UNIVERSITY OF
THESSALY

Department of Computer Science

Professor: George Stamoulis

PhD Thesis

Agricultural Robotics and Automation
Robot Collaboration for Precision Agriculture

---

by

Emmanouil G. Fragkoulopoulos
MSc Computer Science, ESSEX Un, UK

November 18, 2017

Abstract

The need for an increasing agricultural production and a simultaneous decrease in pesticide and other resource usage has led to demand for new methods in agriculture. At the same time, the latest advances in robotics have popularized small autonomous systems, such as Unmanned Aerial Vehicles (UAVs) and robotic rovers. These two seemingly parallel events have contributed in the creation of a new scientific field: *precision agriculture*.

The combination of knowledge from the fields of agriculture, geology, optics, radio communications and robotics has led to new approaches in cultivation, inspection and gathering in farmlands. The most prominent aspect of precision agriculture is the use of automated systems to gather large amounts of data, process them and proceed to localized action.

In this work, the problem of resources overuse, in the form of water and pesticides, is tackled. Image processing methods are used to evaluate the health status of vegetation in great resolution, by means of multi-spectral aerial imaging, captured by a UAV. Health information is combined with positional data from the UAV to produce specific points in the field which need intervention. Path planning and obstacle avoidance methodologies are constructed, which allow a robotic rover, suitable for rough surfaces, to access these points automatically and apply the demanded resource in a localized and economical fashion.

The resulting procedure is tested in the form of a combined system, consisting of a UAV carrying a multi-spectral camera, a robotic rover and a ground control station. Experimental results are presented and discussed upon.

Acknowledgment

Firstly, I would like to express my sincere gratitude to my advisor Prof. Georgios Stamoulis for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank Mr. Stefanakis Dimitrios, CEO of UCAN-DONE for helping me and supporting me with the needed hardware to complete my experiments, also Mr. George Zogopoulos Papaliakos, for helping me with software issues, understanding ROS and the messaging protocols between drones and ground station.

Last but not the least, I would like to thank my family: my wife Aspasia Kotrotsiou, my daugther Sofia-Despina and my son Georgios for supporting me spiritually throughout writing this thesis and my life in general.

Glossary

C++ A modern programming language, characteristic for its low level of abstraction. Due to its very efficient compilers, it can produce very optimized code with fast runtimes, ideal for small-latency environments. 64, 70, 100

GIS A Geographic Information System, commonly referring to software which are able to store, manipulate, analyze and present spatial and geographical data. Commonly used to stitch together aerial photographs of an area, aligning them and referencing them to the coordinate frame. 139

I2C Shorthand for Inter-Integrated Circuit. It is a hardware and software specification for multi-master, multi-slave bus communcation. 29

laser scanner A sensor which is capable of measuring the distance of the nearest obstacle in a range of directions in a 2-dimensional plane. It uses a rotating laser reflected to obstacle surfaces to measure its travel time and estimate said distance. It may be able to identify objects in a full-circle around it, or in a constrained arc. It has a minimum and a maximum measurement range. 16, 79

LiPo Shorthand for Lithium-Polymer battery. An interchangable term with Lithium-Ion (LiIon) batteries, which are the modern choice for storage of electrical energy, thanks to their high power density and high discharge rates, albeit in a more chemically unstable container. 29, 39

MEMS Shorthand for microelectromechanical systems. This is an umbrella term for all modern, sensor and actuator devices of very small size, primarily comprised of semiconductor parts. 29

Mission Planner A Ground Control Station software, suitable for communication and control in systems which use the MAVLink communication protocol. Runs on Windows. 80, 124

multicopter A flying vehicle which uses multiple rotating propellers to produce lift. 21

Multispek A commercial, near-infrared camera model, used in agricultural applications. It consists of a modified GoPro 4 Silver camera, with proprietary firmware and lens. 79

NED The North-East-Down refernce frame. It is a Cartesian 3-dimensional coordinate frame which is used for local navigation near the surface of the Earth. Its x-y plane is tangent to the Earth's surface at the location of interest. 49, 108, 117

Odroid A series of modern, small-size, low-cost, embedded, single-board computers, capable of fast CPU and GPU calculations. 80

OpenCV A modern and popular image processing software library, available in C++ and Python. 86, 95

Pixhawk A flight controller board, designed to host the ArduPilot and PX4 autopilot firmwares. It was designed by Phillip Rowse, in collaboration with 3DRobotics. It is an open-hardware product. When accompanied with suitable peripherals, can serve as an autopilot system for aerial and ground unmanned vehicles. 25, 26, 79

PPM An acronym for Pulse Position Modulation. This is a type of encoding, which allows for multiple PWM channels to be transmitted over one conductor. 27

PWM An acronym for Pulse Width Modulation. This is a type of modulation, which is primarily used by servo motors and Electronic Speed Controllers (ESCs). It carries a single value information, encoded as the high-time of a fixed-frequency pulsetrain. Commonly, the frequency of the pulsetrain is 50Hz and the range of the high-time interval is [1000-2000] milliseconds. vi, 27

Python A modern programming language, with high abstraction level and a remarkably large collection of available libraries. 64, 70, 71, 100, 109

rover An autonomous robotic ground vehicle, which uses wheels or continuous tracks for locomotion. 16, 39, 79

UTM The Universal Transverse Mercator (UTM) is a projection which uses an array of localized 2-dimensional Cartesian coordinate systems to represent locations on the Earth. 49, 109, 117

WGS84 A mathematical representation of the surface of the Earth, in the form of an ellipsoid. Its surface spherical coordinates are the most common format used by GPS systems. 49, 108, 109, 117

Acronyms

ANN Artificial Neural Networks. 3, 4

API Application Programming Interface. 23, 65, 73, 107

CIR Colour Infared. 6

CNC Computer Numberical Control. 42

COTS Commercial Off-The-Shelf. 19, 22, 30, 39

DIY Do-It-Yourself. 26

DSM Digital Surface Model. xii, 6

DSS Decision Support System. 1

ERT Electrical Resistivity Tomography. 2

ESC Electronic Speed Controller. vi, 39

GCS Ground Control Station. 15, 16, 23, 25, 30, 37, 71, 74, 79, 81, 125

GPR Ground Penetrating Radar. 2

GPS Global Positioning System. 1, 29, 42, 72

GUI Graphic User Interface. 125

LAN Local Area Network. 75, 80, 110, 139

LIDAR Light Radar. 6

NDVI Normalized Differential Vegetation Index. ix, 9, 15, 34, 70, 86, 95, 139

NIR Near InfraRed. xii, 4, 9, 34, 70, 100, 127

OS Operating System. 63

PA Precision Agriculture. 1, 2, 6, 8, 9, 10

RC Radio Controlled. 27

ROS Robotics Operating System. 15, 47, 63, 109

RX Receiver. 27, 29

SSCM Site Specific Crop Management. 1

TX Transmitter. 27, 29

UAS Unmanned Aerial System. 139

UAV Unmanned Aerial Vehicle. xii, 1, 2, 6, 8, 9, 10, 12, 15, 16, 19, 39, 79, 123

UGV Unmanned Ground Vechicles. xii, 2, 8, 9, 10

UUV Unmanned Underwater Vechicles. 9

WLAN Wireless Local Area Network. 80

Contents

List of Figures

Chapter 1

Precision Agriculture

Agriculture is a universal and inclusive term which defines the cultivation of plants, animals and fungi's in order to provide resources to asses human needs. Although agriculture is a basic requirement for civilization, nonetheless, non-optimized and brute-force practices can negatively affect the surrounding environment and consequently lead to ecological disasters. In addition, the quadratic growth of human population give rise to an increasing need for more intense and massive farming while repressing the ecological affects of the latter. In order to address this highly important issue, modern technology combined with known mathematical concepts are employed in an effort to optimize and further maximize agricultural production while repressing its ecological effects to a minimum.

The present chapter provides an introduction to the so called Precision Agriculture (PA). Different applications using different platforms and methodologies are outlined in an effort to highlight the effectiveness and the importance of incorporating modern sensing technologies for a better and more efficient agricultural scheme.

1.1 Precision Agriculture

PA, also known as Site Specific Crop Management (SSCM), is a recently developed framework within which, quality factors related to agriculture are precisely observed and analyzed in an effort to maximize productivity while repressing the required resources and the negative ecological outcomes [89]. PA is an inclusive term which holds as special case any technology which aims to observe and accurately map field's parameters in order to allow a site-specific farming strategy.

PA was triggered with the advent of Global Positioning System (GPS) and the trivial access to satellite images. Through these, the spatial variability of field's factors (such as crop yield, topography, organic matter, moisture content) can be accurately mapped and incorporated in a Decision Support System (DSS) specifically defined for an individual field [89]. Moreover, the rapid advancements of UAV led PA to a

new Ira in which high resolution observations can be obtained in a time efficient manner using (relatively) low cost equipment. Fully automatic site-specific robotic swarms, which are designed to optimize the production based on some given constrains (related to logistics and ecology), can potentially revolutionize agriculture and make the latter capable to address the increasing global challenges.

In the following sections, popular applications of PA are briefly outlined and a short introduction is given regarding swarm architectures and real-time collaboration between UAVs and UGVs.

Moisture content estimation

Early work related to PA includes using frequency domain sensors in order to estimate soil's impedance [30]. Subsequently, soil's impedance is correlated with its moisture content and humidity maps are drawn. Apart from frequency domain sensors, the water content can be indirectly estimated using various techniques, among others are, fiber optic sensors [41, 40], ceramic sensor [68] and neutron scattering [77]. As it is mentioned by [77], the methods above are either too expensive or non-practical for usage in the field. To overcome this, a micro controller system which monitors the temperature and the water content of the field is suggested by [77]. In their proposed scheme, the water content is estimated based on the electrical resistivity of the soil which is directly related to its moisture as well as the salinity of the water. Based on these two parameters (i.e. moisture and temperature) a selective irrigation procedure was applied when moisture and temperature reached a predefined threshold [77]. A similar system was designed by [51] in an effort to introduce an optimized framework for farming in India.

Estimated the water distribution within the soil (both horizontally as well as vertically) is a primary goal of the so called near-surface geophysics [26], [64]. Electrical Resistivity Tomography (ERT) and Ground Penetrating Radar (GPR) are two widely used geophysical methods which can indirectly map the subsurface distribution of soil's dielectric properties. In particular, ERT maps soil's electric resistivity [26] while GPR is traditionally used to estimate soil's electric permittivity [18]. Both electric permittivity and electric resistivity are directly related to the moisture content of the soil [22], [33]. Thus, mapping soil's dielectric properties is equivalent to mapping its water content.

Discrimination between weeds and crops

PA has been also successfully applied for separating weeds from crops. Traditionally, weeds can be repressed using herbicides. Although the quality of the latter have been substantially improved over the years, still herbicides can be very costly and in some cases even more expensive than cultivation itself. In addition, herbicides should be used with caution since they are widely considered as dangerous and potentially

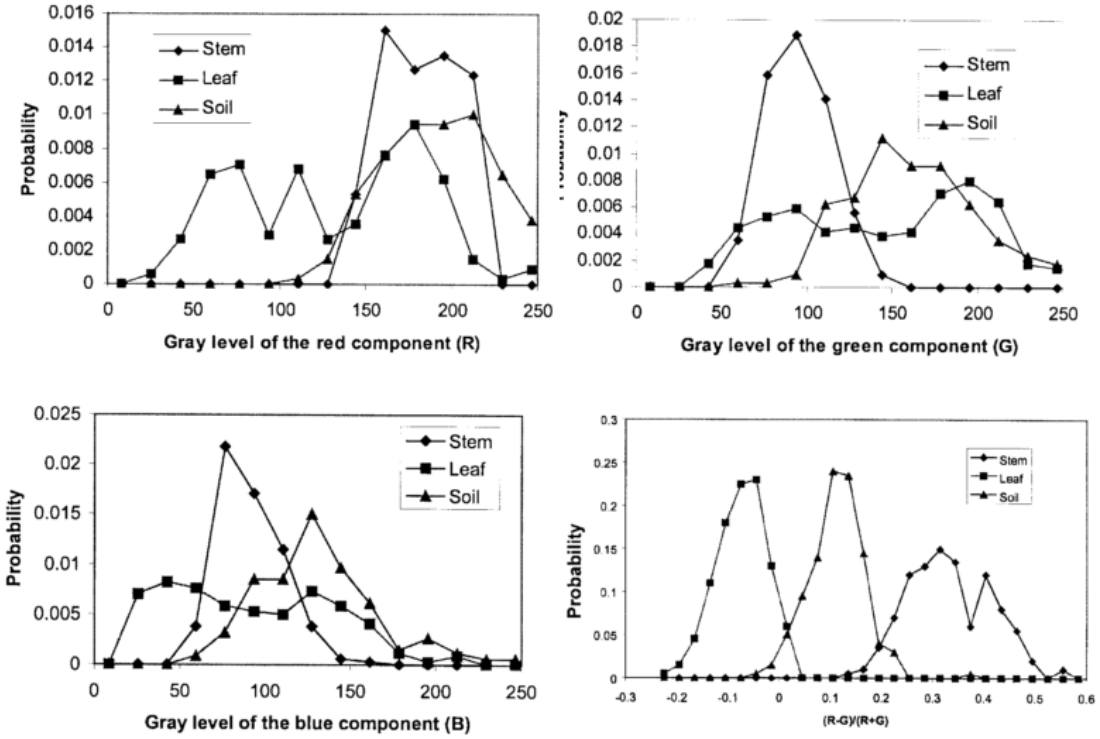

Figure 1.1: Probability of three different colour indexes for steam (weeds), leaf and soil [49].

ecological hazardous materials [27]. From the above, it is evident that separating weeds from crops and applying herbicides accordingly is of a great importance for both ecological and financial reasons.

An early approach for weed detection was suggested by [35]. The main principle of their idea is to detect weeds from aerial pictures based on shape, color and texture features. The main drawback of this approach lies to the fact that different weed species vastly varies in both shape and texture. Cassady et. al. [15] highlighted the importance of color for separating weeds from crops. Zhang et. al.[91] suggested a monochrome based machine vision algorithm in an effort to capture color features associated with weeds. Although laboratory experiments were successful, in real conditions the suggested technique was proven to be unpractical for real-time estimations [49]. In order to reduce the illumination effects and increase the discrimination rate, Mozib et. al. [49] suggested a machine vision framework which is based on a combination of the Red (R), Green (G) and Blue (B) indexes of a picture. They suggested a hybrid color index which is related to R and G

$$F = \frac{R - G}{R + G}. \tag{1.1}$$

The color index described in 1.1, as it is clearly shown in Fig. 1.1, surpasses the

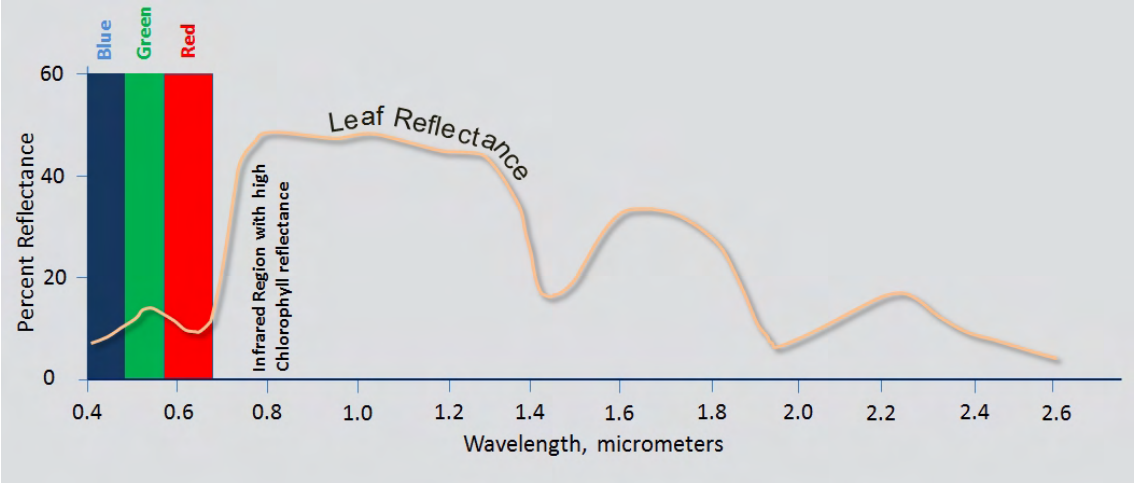

Figure 1.2: Reflectivity of leafs at visible and NIR spectrum (http://www.satpalda.com).

individual R, G, B indices as a parameter for classifying weeds from crops. In the same context, Perez et. al. [57] applied (1.1) and subsequently used classifiers such as k-nearest neighborhood and Bayesian rule in an effort to discriminate weeds from crops based on both their shape and color indexes. The applied classifiers were trained to detect a specific type of weed, namely, dicots. Yang et. al. [87] used a more straight forward approach in which 80 images of weeds and crops were used in order to train an Artificial Neural Networks (ANN). Furthermore the capabilities of the resulting ANN were validated in 30 pictures which they were not part of the training set. Although their results seem promising, nonetheless the method was tested in specimens similar to the ones used during the training process (both training and testing sets were taken from the same field). Thus, this technique should be further tested in different fields and under different conditions in order to further clarify its applicability and limitations.

The previous techniques used aerial pictures taken from commercial cameras based on the R, G, B colour indexes. Apart from that, NIR cameras have been widely used to observe vegetation, since chlorophyll shows a distinctive reflectivity to wavelengths varying from 0.8-1.4 μm (see Fig. 1.2). Gerhards et. al. [31], [32] combined the information from typical RGB and NIR cameras to find distinctive patterns in weed's and crop's spectrum. Furthermore, Tang et. al. [74] and Voix et. al. [78] used NIR images which were subsequently processed in frequency domain using Gabor filters. The resulted images were used to train an ANN designed to discriminate between weeds and crops.

Lee et. al. [42] designed an automatic system for both classification and focused spraying. Their proposed framework can be applied in real time and it is specifically designed for tomato crops. Their method is based on an RGB picture taken from

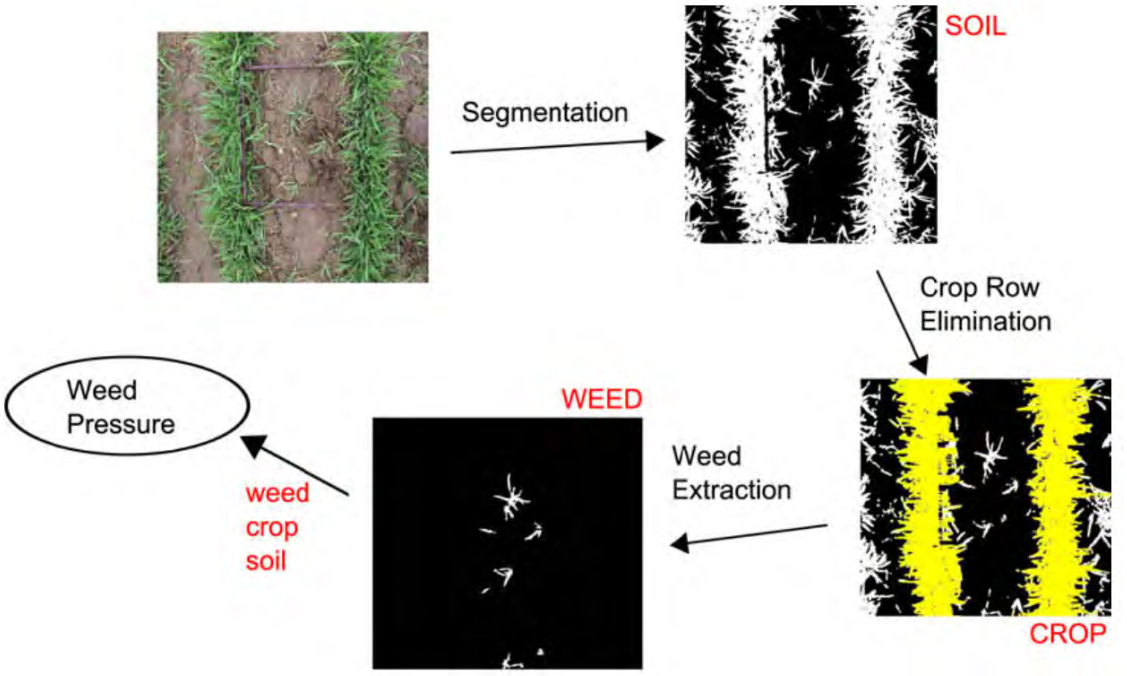

Figure 1.3: The framework suggested by [13] to isolate weeds from crops [13].

a commercial camera that was placed a few meters high from the ground. The subsequent image processing, involves segmentation, binarization and a detection scheme based on structural characteristics, namely, the ratio between heigh and width, main axis of area, perimeter and so on. Burgos-Artizzu et. al. [13] proposed a discrimination scheme based on an RGB camera mounted on the roof of a tractor. The obtained image is firstly transformed using a linear filter ($r = -0.884$, $g = 1.262$, $b = 0.311$) which was proven to perform better compared to the Excess Green coefficients [84, 14]. Subsequently the image is segmented using as threshold the mean value of the intensity of the pixels. From the processed images, the crop rows are specified by using an *AND* technique combining 8 frames. Due to the movement of the camera, the crop rows are further enhanced in comparison to weeds and secondary features. Lastly, the pixels identified as plants which are not laid within the crop rows are classified as weeds.

Burgos-Artizzu et. al. [14] and Xuewen et. al. [85] suggested a more sophisticated processing scheme (not for real-time applications) to isolate weeds from crops. Their method can be divided into the following steps

- Transformed the picture through a linear filter.
- Using a mean threshold to segmented the resulting figure.
- Apply an erosion filter to eliminate isolated black pixels within the crop lines.

Figure 1.4: The DSM constructed by the approach suggested by [88].

- The center of the crop rows is estimated as the column of the figure in which the summation of the pixels intensity reaches a peak maximum.
- An edge detection is applied.
- For every row of the figure, the pixels from the main axis of the crop line till the first edge are defined as crops. The rest are considered to be weeds.

Figure 1.3 illustrates the different steps of the method described above. It is evident that the present technique can not be applied to irregular fields in which crop rows are not trivially identified.

Tree heigh estimation

The height of trees and canopies is of great importance since it can provide essential information regarding ecological, hydro-logical and biophysical processes [69]. It is evident that a direct height measurement for every individual tree would be unpractical and time consuming. To overcome this obstacle, PA approaches have been employed in an effort to measure the heigh of canopies with in a fast and automatic manner. Several examples can be found in the literature of approaches based on photogramic methods, Light Radar (LIDAR), laser scanners and so on [69, 79, 73]. As it is stated by [88], the above techniques require expensive equipment combined with well trained personnel. Zarco-Tejada et. al. [88] used UAVs equipped with a commercial camera modified for Colour Infared (CIR) photography [88]. Using automatic 3D reconstruction techniques, they manage to accurate reconstruct

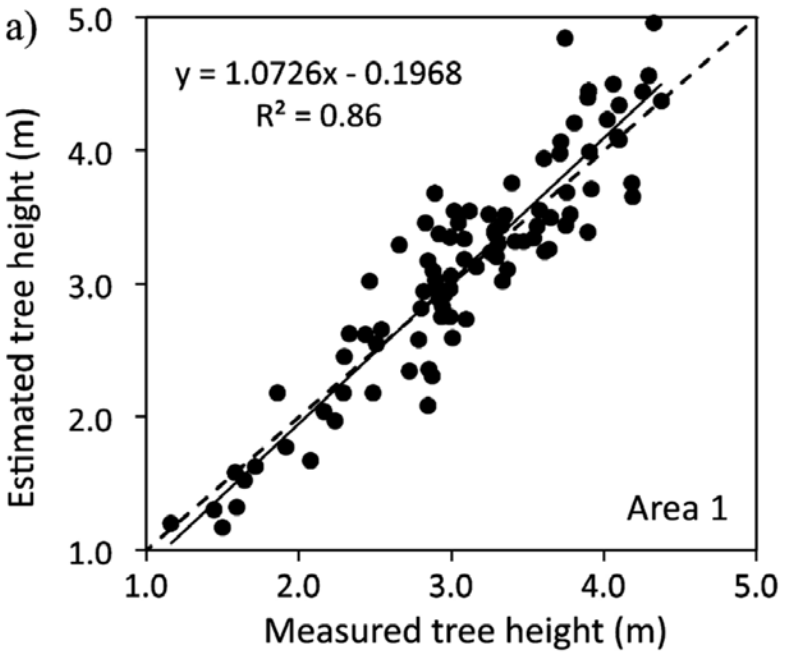

Figure 1.5: Comparison between actual and estimated tree heights using 3D automatic reconstruction techniques on images obtained from UAVs [88].

a DSM of an area planted with olive trees in southern Spain (see Fig. 1.4). In order to validate their results, heights gathered manually from 152 trees were compared with the heights obtained using the automatic 3D reconstruction technique. The actual and the predicted heights are in very good agreement (see Fig. 1.5) which indicates the validity and reliability of their suggested technique [88].

Figure 1.6: A. powered glider, B. powered parachute, C. helicopter, D. fixed wing aircraft, E. Draganflyer X8 quadracopter, F. Aeryon Scout quadracopter [89].

1.2 Collaboration between UAVs and UGVs

UAVs have been widely and successfully applied to PA and manage within a short period of time to revolutionize the field primarily by making high resolution aerial images approachable. Their relatively low cost combined with the fact that UAVs can be trivially equipped with sophisticated sensing tools, make them an attractive choice for addressing problems related to PA. Satellite images, as it is stated by [48], neither can be applied regularly nor meet with the expected resolution needed for PA. Manned airborne platforms although effective, still they remain unpractical due to the high cost of the equipment combined with the operational complexity and the non-trivial delivering system [48, 71]. UAVs can overcome these obstacles by providing a practical and easily accessible platform for PA applications. A vast number of different UAVs have been employed so far,among them are, blimbs [38], balloons [67] and kites [7]. The most widely used UAVs are the remote controlled helicopters and air-planes (see Fig. 1.4) which are recently reach the public domain with their developers trying to appeal to general public as well as specialized scientists [37].

Apart from the applications described earlier in the present chapter, UAVs have been further employed for

Figure 1.7: A swarm architecture designed for UGVs obstacle avoidance based on aerial observations using UAVs [80].

- monitoring fertilizers trials. [90],
- conduct crop scouting and map field tile drainage (using NIR) cameras [90],
- mapping the crop yield variability employing NIR cameras combined with unsupervised classification schemes and principal components analysis [86],
- parameter monitoring based on RGB cameras [24],
- vegetation monitoring and stock valuation [66],
- assessing water stress using various indices like NDVI. [28].

An interesting and more coherent review regarding the UAV systems used in PA can be found in [89].

Collaboration between UAVs and UGVs creates a swarm architecture within which UAVs and UGVs are cooperate in order to assist in situations where humans are

unable to reach [80]. The individual unmanned vehicles are initially independent from each other and can only interact through a cloud environment by exchanging information wireless and in real time [80]. These swarms are called heterogeneous since they consist of both UAVs and UGVs (this definition can be expanded to Unmanned Underwater Vechicles (UUV) as well).

A simple example of heterogeneous architecture is described by [29]. In their work a single UAV is used to map a specific area and detect possible obstacles in order to navigate a UGV accordingly. Chaimowicz et. al. [17] designed a more sophisticated heterogeneous architecture in which a blimp was used to navigate a group of UGVs that was formed using pattern recognition approaches specifically modified to form Gaussian distributions of UGVs on the ground. In the same context, Tanner et. al. [75] proposed a decentralized navigation of UGVs based on information given in real time by a swarm of UAVs. A similar approach was also used by [16] in order to designed a heterogeneous robotic system specifically applied for urban applications and generic localization. Localization as well as feature extraction using heterogeneous robotic swarms was also achieved by [34]. An inclusive and more complete review of heterogeneous architectures can be found by [80] and [23].

The examples above, present (mostly) abstract applications of heterogeneous swarms, and in particular joint navigation and obstacle avoidance [80] (see Fig. 1.7). In the present thesis, a heterogeneous swarm is specifically designed and optimized for PA applications. The two fields, namely heterogeneous robotic systems and PA are merged in an effort to create a new PA framework capable for automatic actions based on precise observations in a practical and trivial manner.

Chapter 2

Robotics Applications on Agriculture

In Chapter 1, the importance of precision agriculture was presented and the benefits of its application on modern crops were laid out. At this point, one should consider the highly technical nature of precision agriculture: Its methods involve the accumulation of large amounts of data, at regular intervals as well as the timely and consistent processing of said data.

Leaving this task to the farmer is against the purpose of this effort. Humans are not reliable when called to perform repetitive tasks, nor are able to process large amount of data with accuracy and speed. On the other hand, computers are known to excel in those areas. Naturally, software might be able to handle the data processing, but cannot go out in the field and collect data samples. This is where robotic platforms can be used, to act as the physical extensions of computing systems.

A robotic system with suitable design is able to carry out these required tasks, complementing a precision agriculture system:

- Field traversing
- Data collection
- Data relay / storage
- Targeted application of treatment
- Constant level of repeat-ability of used methods

Robots have already been introduced in agriculture, in its traditional form. A well-known example are the automated tractors and harvesting machines (Figure 2.1). These robots are built upon a traditional form-factor, implementing an already available functionality, with the difference being that they don't need a human operator

Figure 2.1: An automated tractor

to be occupied with the task of commandeering them constantly which provides considerable benefits. The human is no longer exposed to the danger of labor accidents, predominant in agricultural workflows, and harsh environmental conditions. Moreover, the machine is guaranteed to carry out the harvesting procedure relentlessly without pause with the same level of accuracy and precision. However, it should be noted that usually robotic platforms still need some amount of supervision, to prevent the consequences of malfunctions, mechanical failures and software bugs, as well as periodic maintenance.

Another, more modern approach to introducing robots in agriculture is the use of UAVs for spraying pesticide. Using an unmanned aerial platform for the application of pesticide overcomes problems inherent to the traditional methods of application. Its ability to hover above the crops protects the crops themselves from being trampled by a ground vehicle but also allows the UAV to operate over rough or inaccessible terrain. Moreover, the choice of the helicopter form factor allows it to hover at very low height, ensuring a very focused application of pesticide, a known problem when using airplanes to spray crops: it is difficult to enforce the spread of pesticide withing a very restricted field area and that causes, among others, legal problems with legislation regarding spraying. Finally, as it is to be expected, no aircraft pilot is exposed to labor accidents.

Still, there are currently few to no robotic systems which implement the core tasks of precision agriculture. This is to be expected, since precision agriculture itself is a relatively new sector of research, finding gradually its way to the commercial sector. As its methods are still being refined and tested, there is not a large user base yet to justify investment towards large-scale robotics research and manufacturing for such

Figure 2.2: The RMAX helicopter spraying pesticide

applications. This is expected to change soon, as precision agriculture becomes the de-facto approach for crops supervision and intervention, bringing especially large savings in cost and effort in large-scale cultivation. With that prediction in mind, in this work, an automated, proof-of-concept robotic system is presented, designed to supervise crops, collect relevant data, process them and intervene accordingly.

Chapter 3

Overview of the Proposed System

In this chapter, the overall architecture of a proof-of-concept robotic system which implements methods of precision agriculture is presented. The system is designed to capture the health index of a pre-defined crop field, extract the least healthy locations and intervene by applying water to those areas. At the same time, tight constraints in size, cost and complexity were applied in order to ensure that the system would be flexible and easily deployable, even by a single person.

In order to acquire data on the health status of the vegetation, a multi-spectral camera is used to capture aerial images in the red and near infra-red bands. The camera is carried by a UAV, programmed to fly a mission over the area of interest. Details on multi-spectral imaging and the NDVI index are provided in Chapter 5.

The UAV which is selected for this application has the form of a quad-rotor. The decision on the form of the UAV is well-thought and the reasoning behind it is presented in Chapter 4. The open-source autopilot software Ardupilot and open-hardware autopilot hardware Pixhawk were the enabling technologies behind the UAV solution.

The captured images are downloaded wirelessly and in real-time in a laptop computer in the Ground Control Station (GCS) where they are processed to extract the NDVI of the vegetation under inspection. At the same time, a telemetry stream generated by the UAV is recorded as well, containing position and attitude information. The imaging information is combined with the telemetry stream from the UAV for geolocation so that each data point can be georeferenced in the actual world and addressed with geodetic coordinates. A discussion on frames on reference can be found in Section 7.1.

At the core of the software running in the GCS is the Robotics Operating System (ROS), allowing for the deployment of a modular software solution, spanning the entire network of interconnected systems.

It is of interest to group the unhealthy points to form areas where intervention

material should be applied. This not only prevents the unnecessary waste of excess resources, but also minimizes the time required to apply the material. Thus, the unhealthy points are grouped with clustering algorithms to allow for specific and localized intervention and their coordinates are passed onto the robotic rover.

The rover has information on the crop row geometry in addition to distance measurements from a laser scanner, which allow it to navigate the field safely, avoiding crop rows and other obstacles. This topic is expanded upon in Section 7.3. In this work, the rover is not equipped with any kind of applicator, since this was deemed out of scope and a technical problem of different, separate nature.

The major hardware components of the system are:

1. The UAV used to carry the multi-spectral imager and to perform automated scanning missions over the area of interest
2. The multispectral imaging sensor which captures data in the red and near infra-red spectra, which in turn is then used to extract vegetation health information
3. The GCS in the form of a wireless access point and a laptop computer, for data transfer and processing and vehicle coordination
4. The rover which visits the areas of interest in the field to apply treatment

An abstract diagram of the proposed system configuration can be seen in Figure 3.1.

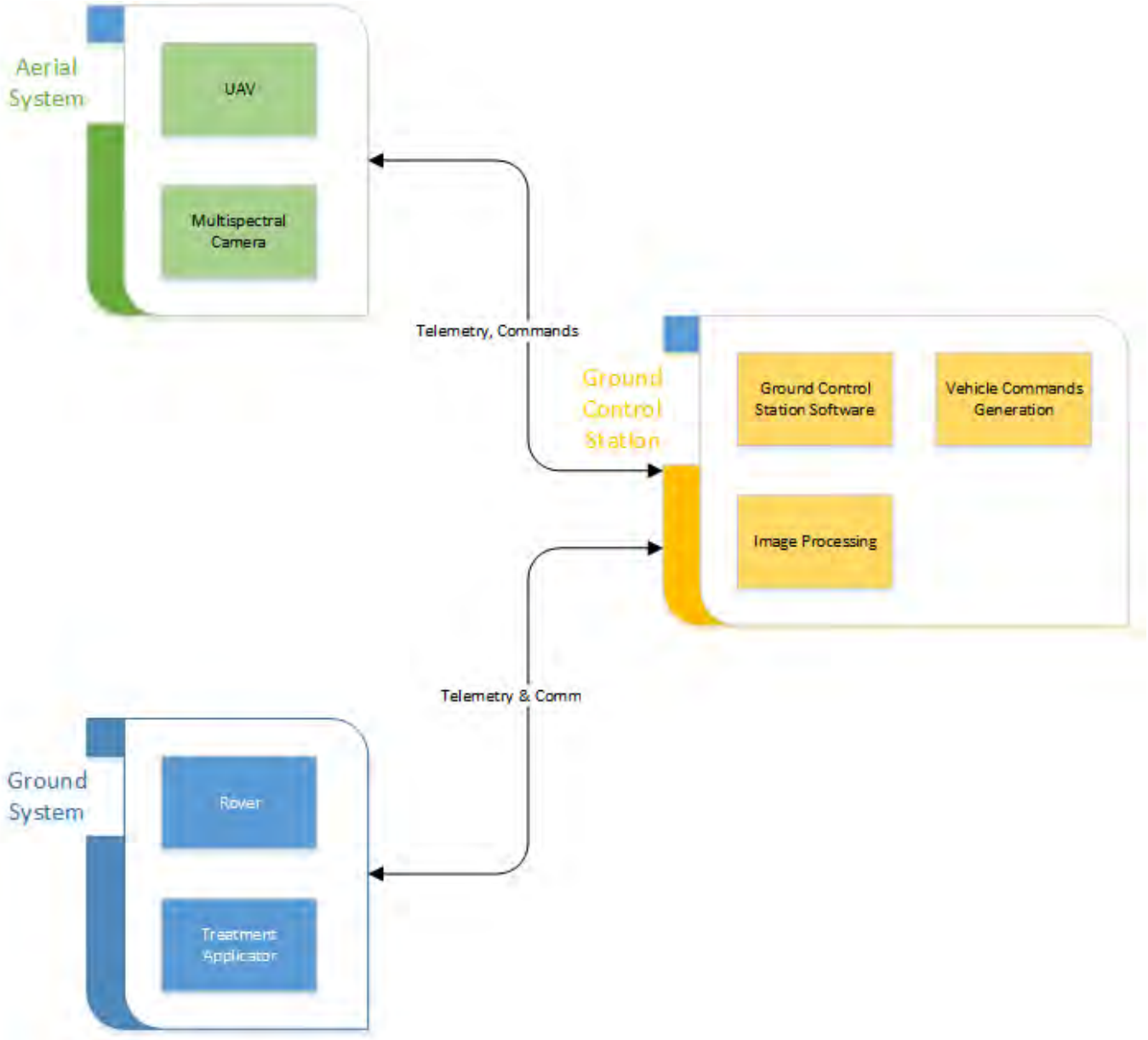

Figure 3.1: Abstract system diagram

Chapter 4

The Unmanned Aerial Vehicle

4.1 On the Choice of the Most Suitable UAV

The core of the precision agriculture application presented in this dissertation is the UAV which is tasked with scanning the crops with a specialized image sensor. Data captured with that sensor can then be analyzed and used to extract metrics, regarding the health and growth of the crops.

Even though the UAV might be the most impressive and flamboyant component of the overall system, one should keep in mind that it is merely a means to an end. The goal is the acquisition and processing of valuable field data, and the means to do it is the capturing system. In fact, the image sensor might arguably be considered a more rare and specialized commodity than the UAV itself: multi-spectral cameras are still very expensive, hard to make and maintain, as well as quite fragile. On the other hand, the UAV revolution is in full bloom and has produced plenty of options, when it comes to lifting and carrying payloads over areas of interest.

Given the above, it might sound as if the selection and operation of a UAV for precision agriculture applications is a non issue. That would be as far from the truth as it can be. Choosing the correct UAV for each application is a major engineering problem which teams and companies face daily. It is even very likely that the most suitable UAV system for an application is a custom-built one, instead of a Commercial Off-The-Shelf (COTS) one.

A realistic approach is to build the UAV around the specified payload, while taking into account the mission requirements. The most crucial decision concerns the airframe type. There are 4 major categories:

- Fixed-wing aircraft
- Helicopters
- Multirotors

- Hybrid air-frames

Fixed-wing aircraft

Commonly known as airplane, this is the most traditional aircraft category. It is well-studied over the span of more than a century and this has led to complete solutions regarding issues on aerodynamics, construction technology, propulsion and control. UAVs which fall under this category also have been studied extensively over the past decades, but room for improvement still exists, with the advance of new materials, structural configurations and control techniques.

Fixed-wing aircraft boast very efficient flight, the fastest flight speed and great payload capacity.

On the downside, they must maintain a minimum airspeed airspeed to keep themselves on the air and require a minimum, relatively clear stretch of ground to land (and possibly take-off, with the exception being the hand-launched vehicles). Parachutes have made an appearance on airplanes, removing the need for a landing strip, but are not applicable in every case; they cushion the fall, rather than nullify it, and thus are not an option in expensive and delicate models. On the smallest scales, airplanes are also very susceptible to wind.

Helicopters

These are the second-oldest motorized, heavier-than-air flying platform. They are also extensively studied and are also quite efficient in flight, albeit not as much as airplanes.

On the bright side, they are exceptionally maneuverable in flight and tolerant of weather adversities. They can take off and land vertically on any terrain and can carry significant payloads.

However, helicopters are notoriously complex in construction and demanding in service and maintenance. They are also very hard to fly, in small scales, and this is all the more true with autopilots. Little to no mature, full-featured autopilots exist for helicopters, whose absence reflects on the importance of the control problem of this highly dynamic, highly non-linear system.

Multicopters

This category covers all vehicles which lift themselves by using multiple rotating propellers, in comparison to a helicopter, which uses a single one (or two, in some exceptions). This is a relatively new category of aircraft, in the sense that it never had a counterpart in manned flight. Even though there have been experimental

Figure 4.1: A multirotor UAV

aircraft in manned aviation in the past, technical difficulties never allowed them to claim their place in the skies. A typical multirotor UAV can be seen in Figure 4.1.

The most significant constraint stems from the physical characteristics of propellers, which exhibit reduced efficiency, the smaller they are. Thus a single, large propeller will always be more efficient when producing lift, compared to multiple, smaller ones. This explains the dominance of the helicopter over the multicopter, in spite of its mechanical complexity.

The advance of technology in the domains of electrical motors, batteries and sensors, have allowed the multirotor to take form as a UAV, where its advantages often outweigh its drawbacks.

It is the simplest to manufacture and maintain of all previous airframes. It consists of a simple geometric design and topology of the motors. Brushless DC motors, mostly used in multirotors are simple to manufacture and virtually maintenance-free. The airframe displays a natural hardware redundancy, in the form of multiple lifting motors. It is impossible to control fully manually, but relatively simple to control with the means of an automatic control system. Naturally, it can hover above a specific location and take off and land vertically.

On the down side, it has very poor efficiency and hence low payload capabilities. Their flight time is equally constrained.

Hybrid UAVs

Aircraft which are a combination of airplanes and multirotors fall under this category. They have fixed wings which produce lift when flying forwards, but they also have motors which point upwards, dedicated for vertical movement. Some configurations use the same motors for vertical and forward propulsion, by placing them on

Figure 4.2: A hybrid UAV

moving supports and rotating them, while transitioning from multirotor to airplane mode and vice versa.

There are some real-world examples of this kind of airframe [81], but hybrid aircraft are mostly met as a UAVs.

These vehicles span the gap between an airplane and a multirotor, trying to combine all of their strong points (long flight times, fast flight, significant payload capacity, hover and vertical flight capability) while leaving out their disadvantages. In practice, they achieve an intermediate performance in all of the aforementioned tasks.

More importantly, there still aren't any proven control algorithms for this type of systems nor any COTS autopilot modules.

Still, this is a very promising category of UAV, which is gaining more ground progressively. A popular hybrid UAV model [8] can be seen in Figure 4.2.

■

Now that the major UAV categories have been presented, the selection of which one should be used for the needs of the current application can be discussed. The target application of the selected vehicle is to carry an multispectral camera (weighing about 100gr) over a field, while following a scanning pattern over it. Since this is a proof-of-concept system, a low-cost, low-maintenance solution is sought. Vertical take off and landing capability is required, so that the system can be deployed in remote fields, far from airstrips. A small flight time (and hence coverage) of a few minutes is adequate for the needs of this work.

All of the above constraints factor towards selecting the multicopter, as the most suitable airframe solution for this application. Indeed, while full-scale and small-scale airplanes are often used in real applications for mapping large portions of land,

multicopter UAVs are also used when there is a need for more detailed scanning over a smaller area.

4.2 On the Choice of the Autopilot Module

Specification of the Autopilot Required Features

Once the aircraft type is selected, the next choice concerns the autopilot system which will be installed in the UAV. The fact that an autopilot providing some level of autonomy will be installed in the aircraft should be taken as a given. The entire premise of a robotic system is for one or more computing systems taking over one or more moving platforms in order to coordinate and automate a task; a human operator is an undesirable link in the chain of a robotic system, because he can often be an unreliable pilot, lacking consistency and sharp timing. Instead, it is much more preferable to assign humans with the task of supervision, where his mature and elaborate high-level reasoning can prove valuable.

There are multiple autopilot systems available in the market currently, suitable for various aircraft types. In this text, only multicopter autopilots will be considered. Their cost varies from under $100 to over several thousand dollars. Their target applications reflect this price range, with the cheaper addressed towards hobby and recreational use while the most expensive marketed towards military and security applications with corresponding specifications and quality control.

The functionality these systems offer also varies. A short list of features commonly found in multicopter autopilots can be seen below, sorted from the most basic to the most exotic:

- attitude stabilization
- altitude control
- position control
- power management
- telemetry feedback
- GCS communication and cooperation
- waypoint mission functionality
- mature telemetry message protocol Application Programming Interface (API)
- integrated video feed
- collision avoidance

- redundant sensor set
- encrypted communication link
- redundant actuator / re-configurable control

The most crucial features that are required for the application involved in this work are full attitude, altitude and position control and waypoint specification, in order to define the mapping mission with accuracy. Also, telemetry feedback and a mature API are needed, for the rest of the robotic system to interface with the UAV in a programmatic way. Collision avoidance and other end-product features are not necessary for the UAV, for this proof-of-concept application.

Candidate Systems

Given the above specifications, we can immediately reject low-end systems, intended solely recreational use, such as the CC3D [59] and the Naze32 [3]. We can also opt out of going for a high-end solution, such as the Piccolo [72] and the Airware systems [9], or other integrated solutions [43].

Basically, a middle ground is sought, occupied by integrated solutions such as the DJI [21] and Mikrocopter [47] systems and the aircraft-agnostic DJI [20], ArduPilot [62], PX4 [62] and Paparazzi [60] autopilot modules.

Delving deeper and examining the features provided by each system individually, currently offered integrated systems (combinging the flying platform and the autopilot module in one product) can be rejected. Those systems are primarily targeted towards a client base which has little technical and piloting knowledge but require a plug-and-play solution for their key application. Such groups are photographers, cinematographers and news reporters. As a result, little to no telemetry information is reported back from these platforms and no API is provided; indeed such provisions would be a waste of resources for the recipient clientele.

Moving on to stand-alone autopilot modules, a product-by-product comparison can be made.

The DJI autopilot modules, with the A2 being the previous model and A3 being its latest evolution, are considered to provide a very robust controller for medium and large airframes. They have high integration with peripherals such as mobile phones and tablets, but do not provide an extended telemetry set nor an API of any kind. DJI has announced a new system providing amenities for programmers and roboticists, but it does not have significant impact yet.

Paparazzi [60] is an open-source software project, intended to offer a sophisticated and free autopilot solution to anyone interested in having one. It boasts a re-configurable module architecture with swappable components and is one of the

oldest projects in its kind. However, it has not gained significant momentum over the years, and as a result doesn't provide two indispensable features: detailed documentation and a proven hardware supporting it.

PX4 [1] is an open-source software autopilot project, maintained by the ETH Zyrich university. It might arguably be the most advanced autopilot software, featuring reconfigurable and swappable software modules, cutting edge control schemes fresh-off the ETH research and a mature telemetry messaging system, doubling as its API. It runs on the Pixhawk hardware (Figure 4.3), which was designed with collaboration with 3DRobotics, a UAV startup company. Sadly, it also has constantly outdated documentation and an unstable GCS.

Finally, the ArduPilot [58] project is one of the most prominent open-source autopilot project in the scene. It shares the Pixhawk as its hardware platform with PX4, but has also been ported to a multitude of other autopilot hardware boards, both in the microcontroller realm (PixRacer, APM2 etc) and the embedded computer realm (Navio, Erle-Brain etc). It also uses the same telemetry messaging system with PX4 (MAVLink [45]), but the similarities stop there. It is led by a group of programmers and engineers from around the world who contribute without profit to the project and, as a result, is developed with usability and functionality in mind. It has extensive documentation and a large community supporting new users. A lot of features have been introduced into it, such as camera gimbal control, support for a variety of sensors, certain redundant subsystems and an extensive, yet still under development API, which may not be as powerful as PX4's, but rich nonetheless. It interfaces best with the Mission Planner GCS software [54], a Windows program. This autopilot system might arguably be the most used in custom-made UAV as of today.

For all of the aforementioned reasons, the ArduPilot system was chosen for the development of this work.

Figure 4.3: The Pixhawk flight controller

4.3 The ArduPilot Ecosystem

Since the ArduPilot system is an integral part of the proposed system, this section is dedicated to its presentation.

Initially, the ArduPilot project was the result of the effort of Chris Anderson's, co-founder of 3DRobotics, to create a global hobbyist community, to which he could later sell the UAV autopilot product which his company was developing. Anderson created the internet blog/forum DIYDrones.com [5] with the intent to gather anyone who was interested in contributing to and utilizing an autopilot system with a dominant open-source and Do-It-Yourself (DIY) character. The household electronics/robotics revolution previously brought by the Arduino [44] microcontroller, led to the project being named "ArduPilot". The experiment was successful and the DIYDrones community grew to a point where other programmers and engineers took major positions in the ArduPilot project. Eventually, as 3DRobotics lost interest in the DIY hobbyist market and decided to stop supporting the ArduPilot project financially, enough momentum had been gained that an independent foundation [55] would be formed to keep the project alive.

ArduPilot is a very mature software project. While it was still hosted in the Arduino platform, it was the largest software project compiled for that board. At one point, it became too cumbersome and limiting to maintain code that would fit into the limited board memory, so the firmware migrated to the then-newly designed Pixhawk board.

Today, ArduPilot is hosted in Github, a hosting service for version control of collaroborative projects, and linked to continuous integration utilities, such as Travis, to facilitate author collaboration. It is broken down into multiple library modules, to make the addition of new code and functionality (such as drivers for new sensors and controllers) easier. It runs on a scheduler coordinating service, in order to ensure a causal runtime logic, in the same working premise as a Real Time Operating System (RTOS).

Multiple vehicle types are supported by ArduPilot. Radio Controlled (RC) airplanes, helicopters and multicopters, cars and boats can be converted onto autonomous systems, by installing an ArduPilot-enabled unit. The corresponding firmware "flavours" of ArduPilot for each platform are named ArduPlane, ArduCopter and ArduRover. The separation of ArduBoat from ArduRover is under consideration and a special firmware for antenna trackers also exists, called Antenna Tracker.

4.4 The Pixhawk Hardware and its Peripherals

As mentioned before, ArduPilot can be hosted on a multitude of boards [61]. However, it is developed with the Pixhawk hardware primarily in mind. For that reason, and because a Pixhawk was used in this work, this board will be the focus of the following section, where an overview of the autopilot hardware will be presented. Still, most hardware alternatives have the same principle of operation and can communicate with roughly the same set of peripherals.

The Pixhawk Board

The Pixhawk hosts a 32-bit ARM Cortex M4 core with FPU, running at 168 Mhz, with 256 KB RAM. It contains 2 MB Flash for permanent memory for storing firmware. Also, it has an additional 32-bit fail-safe co-processor. An interface with an external SD card is also available, for data-logging purposes.

Pixhawk is primarily powered by a 5V power supply, but allows for an extra power input for redundancy. The powering options will be explained later in the section. However, its digital logic is 3.3V but 5V tolerant.

Since it is intended to act as a host device for a multitude of peripherals, it is primarily populated with sockets, accepting external connections. In Figure 4.4 and Figure 4.5 graphic descriptions of the hardware interface can bee seen.

The most common hardware ports are presented below:

- RC Input: This 3-pin port accepts the pilot commands, produced by the handheld Transmitter (TX), in the form of a PPM stream . This port also provides 5V power for the Receiver (RX).

Figure 4.4: Hardware interface of the Pixhawk hardware, top

Figure 4.5: Hardware interface of the Pixhawk hardware, side

- RC Output: Through this 14 channel output array, the control commands towards motors and control surfaces are communicated. They are capable of producing PWM output asynchronously from the main firmware code execution.

- USB: A common USB port, providing power to the board, during setup on the bench. Usually not used after the system is assembled. Firmware upload and update takes place through this port.

- Power Input: This port serves as the main 5V input to the Pixhawk board,

but also has analog voltage and current sensing pins allocated on it, for power monitoring. Usually the Power Module (see next subsection) is connected to this port.

- Serial Ports: Pixhawk hosts 5 TTL serial ports, spread over 4 connectors. These can be used for a multitude of purposes, including telemetry flow, GPS interfacing and also generic communication with optional peripherals.

- I2C Port: Many standard sensors which are used by Pixhawk communicate using the I2C [82] protocol, such as the digital compass. An external bus splitter board is provided, when multiple devices need to be connected simultaneously.

Basic Peripherals

Almost all autopilot systems distribute their hardware over more then one board / enclosure. While this approach may reduce ruggedness and increase complexity, it allows for more setup options in custom-made systems. Additionally, it leaves room for easily upgradable, replaceable peripherals.

The most significant and common peripherals of the Pixhawk are:

- External Digital Compass: A MEMS 3-axis magnetometer, used to measure Earth's magnetic field, for the purpose of orientation. Even though Pixhawk also has an internal magnetometer, that is not preferred, due to the heavy electromagnetic interference commonly found near power systems. Usually sharing the same enclosure with the GPS receiver.

- GPS receiver: A crucial sensor, enabling the autopilot to estimate its absolute position, which is necessary for autonomous missions, comprised of waypoints. Since it relies on direct line of sight with satellites of the GPS constellation (which emit a relatively low-power signal) it cannot function indoors, within intense geological formations (such as canyons), under dense forest canopy or under heavy cloud coverage (e.g. during storms).

  Typically, the precision of a GPS receiver is at the order of 3-5 meters, but more accurate (and expensive) solutions can be employed, which raise the precision up to centimeter-level.

- RX: This is not strictly considered part of the autopilot system, it is required nonetheless, because the UAV pilot needs to be able to issue control or mission commands, using a TX. In some systems this is a required functionality, while in others it is included mainly as a safety, fall-back control channel.

- Power Module: Commonly, the only power storage available on-board a UAV, is a LiPo battery. However, its voltage is typically between 11 and 18V. Since

Pixhawk requires a strictly 5V power input, this devices is tasked with regulating the battery voltage to usable levels, while providing enough current for the autopilot and its most crucial peripherals. It does not have enough amperage to power the servomotors as well.

Commonly also provides voltage and current measurements as well.

Other Compatible Peripherals

The Pixhawk board is able to communicate with and control many more peripheral devices, which enhance its functionality. Below is the list of the most common ones, but their range continuously increases, as support for new devices is added.

- Telemetry Transceiver: Having a live telemetry link with the UAV while it is in the air is considered a crucial feature, both for functionality and safety reasons. Pixhawk is able to communicate with a typical TTL serial radio and establish communication with a GCS, using the MAVLink protocol. Dedicated COTS telemetry devices exist, which packetize information according to MAVLink specification, improving transfer and packet drop rates.

- Rangefinder: So far, none of the aforementioned sensors allows a UAV to know its distance from an object or terrain feature. This information can be provided by a rangefinder sensor. There are many different implementations for such sensors, utilizing different technologies, such as ultrasonic and laser waves. They come in different measuring ranges, from sub-meter to hundreds of meters and have various scan ranges. Others can measure the distance in a very narrow beam in front of them, whereas others provide a full 360 degree distance scan of a 2-D plane.

  Pixhawk is readily compatible with single-beam sensor, either ultrasonic or laser, which uses for obstacle avoidance and height measurement during landing.

- Cameras and Gimbals: Capturing images and video is one of the most prominent applications of UAVs. For that reason, Pixhawk can interface with many camera models, triggering them at specified instances in space or time or controlling their zoom level. Additionally, it can work with stabilization systems and gimbals, to provide pan-, tilt- and yaw- compensated shots, which greatly enhance the comfort of a human observer.

In short, the Pixhawk flight controller is a modern, high-end, hobby-grade autopilot hardware. Thanks to its well-thought design and quality components it has a small footprint and is easy to deploy on custom UAV platforms. It has had great acceptance in amateur UAV systems, but also in small commercial systems, despite the

fact that it does not have aerospace specifications. This goes to show that currently, UAV systems are experiencing a boom in the private sector, who is searching for cheap and efficient solutions. Unfortunately, the current manufacturer of Pixhawk, 3DRobotics has ceased production but it is expected that the design, since it is open, will be adopted by some other manufacturer or be embellished and released in some other, evolved form.

4.5 The IRIS+ Platform

Having expanded upon the UAV platform and autopilot system selection criteria, we can now proceed on the selection of the specific UAS which will carry out the task at hand. We have come down to using a multicopter and the ardupilot/Pixhawk autopilot. What remains, is to combine them into a complete UAS solution.

This integration step may involve building a custom-made multicopter and a proprietary installation solution of the autopilot system. This direction would require careful selection of the UAV hardware components, so that they fit the energy budget of the mission at hand but also so that they can cooperate harmoniously with each other, to provide appropriate lift and endurance. This is not a trivial task and often a lot of components (propellers, motors, batteries) are swapped for others with different specs, in an iterative sizing procedure. Even though guidelines and websites tackling this problem exist, it remains more an art than a closed problem.

Moreover, tuning the control parameters of the autopilot software for a unique UAV platform of unknown properties is also an open problem, involving iterative steps and a lot of trial and error.

While the above steps may be mandatory in edge cases, where very specific requirements must be met, in most cases a ready-made commercial multirotor can fit the mission requirements. Even better, there exist ready-to-fly multirotors, with the autopilot system already embedded. One of these products, and the one that was selected for this work, is the IRIS+ platform, depicted in Figure 4.6.

IRIS+ is a (now discontinued) product of 3DRobotics, consisting of a ready-to-fly quadrotor and a pre-tuned Pixhawk embedded into it. It can achieve about 20 minutes of flight time and has a payload capacity of about 400 g. A telemetry radio is installed inside it and it can carry a 5100 mA h 3-cell battery. It also comes with a pre-bound and configured RC transmitter.

This UAV solution is very popular among robotic amateurs and developers, since it provides a low-cost, yet rugged and well-performing combination of flying platform and autopilot, provided that its specifications fit the application requirements.

Figure 4.6: The IRIS+ quadrotor

Chapter 5

The Image Capture System

In this chapter, a short introduction on multispectral imaging will be provided. The multispectral camera used in this work will be presented and its technical details will be given.

5.1 Multispectral Imaging

Humans are able to perceive through their eyes radiation in a relatively narrow band of the overall spectrum, approximately in the interval of (380nm - 700nm), known as the *visible light* spectrum, corresponding to the violet - red range. Still, radiation outside of this band exists and propagates in the same manner, unbeknownst to us.

Image sensors, commonly known as cameras, are designed to capture frames of a scene, in the same way we perceive it. As a result, most cameras are designed and manufactured to be sensitive to the visible light spectrum and in fact discern radiation in the same manner as the human eye: by categorizing it into three, slightly overlapping ranges. These correspond to the well-known basic colours:

- blue: 400nm-500nm
- green: 450nm-630nm
- red: 500nm-700nm

These ranges are assigned to the three channels of color photography and videography and reproduced by screens and printers.

Still, special sensors can be manufactured, which are sensitive to other radiation ranges, collecting information for uses other than visible image reproduction. Common applications of useful information located outside the visible spectrum can be found in meteorology, astronomy and thermal imaging.

When an image sensor is tuned to capture a small amount of relatively wide radiation band, distinct from each other, we call it a *multispectral imager*. In contrast, hyperspectral imagers capture a large amount of narrow, close to each other bands of radiation.

For the purposes of remote sensing, in the agriculture background, more useful information can be collected from radiation outside of the visible spectrum and specifically from the NIR band.

5.2 The NDVI

Plants have the ability to reflect the infrared light effectively, so as to avoid absorbing this high-energy radiation which would otherwise raise their internal temperature to harmful levels. In contrast, they absorb red light, which is useful for photosynthesis. Healthy, well-watered plants are able to reflect NIR radiation much more effectively than water-deprived, sick plants.

We take advantage of this fact to create imagers which can detect remotely the health status (also known as Normalized Difference Vegetation Index (NDVI)). This kind of cameras usually have a red and green capture channel similar to common cameras, but instead of a blue channel, they have it replaced with the information for the NIR band.

The NDVI index can be calculated as

$$NDVI = \frac{NIR - RED}{NIR + RED}$$

Healthy plants which reflect radiation effectively should present an NDVI is higher than 0.35. Plants over 0.25 are usually sick. Anything below that threshold corresponds to lifeless objects. Compare Figures 5.1 and 5.2; In the first picture, healthy vegetation full of leaves produces high NDVI values, whereas in the second one, where vegetation has dropped its leaves for the winter, areas with low NDVI appear.

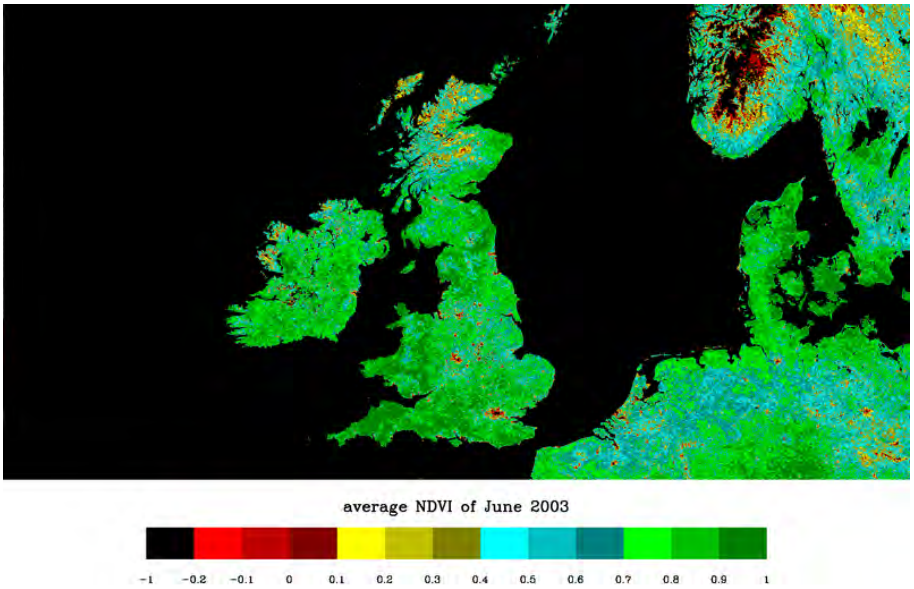

Figure 5.1: NDVI Index of Vegetation in Summer

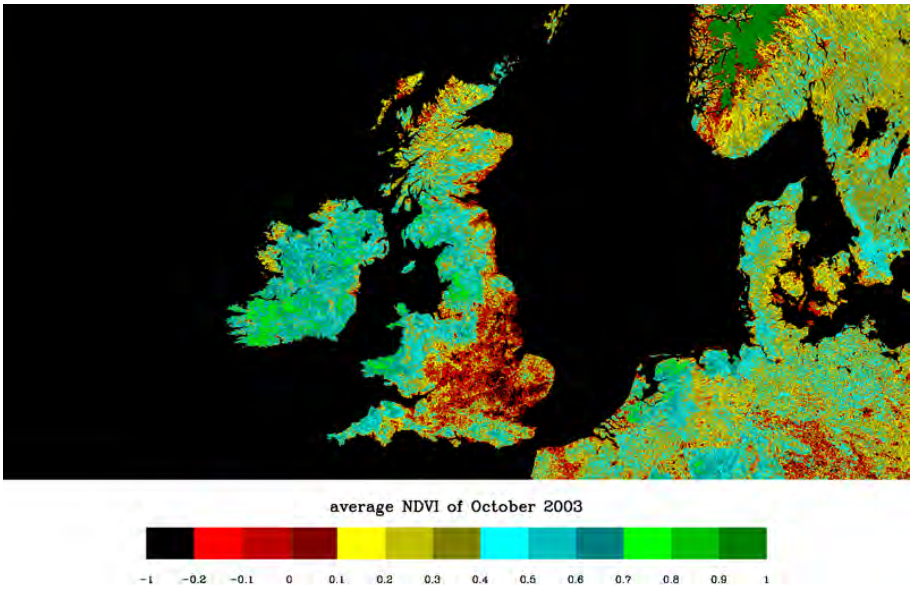

Figure 5.2: NDVI Index of Vegetation in Winter

5.3 The Multispek Camera

Multispectral imagers are not new technology. There are many models available, from experimental, high-end, unique sensors mounted on satellites to hand-held cameras. In general, they are considered an expensive device, available to specialized, professional operators. However, along with the access of a larger user-base to UAVs, some relatively cheap multispectral cameras have been made available.

For this work, the Multispek camera (Figure 5.3, now discontinued) was used. It is based on the popular GoPro Hero 4 Silver video camera, which the Multispek company modified and repurposed for remote sensing applications, by removing the infrared (IR) filter from the lens and changing some firmware parameters.

The most important specifications of the Mutlispec camera are:

- weight: 63 g
- resolution @ 400ft: 3.8cm
- pixel count: 12MP
- image capture rate: 1.4s

Figure 5.3: The Multispek camera

- power consumption: 2.3W

While there are other, equally or more capable cameras for this application available right now, Multispek was a valid option at the time it was made. However, it is not recommended, since it was discovered that it suffers from NIR spillage to its other channels, a defect which other modern cameras in the same price range do not suffer from anymore.

5.4 Connection with Pixhawk

The GoPro Hero 4 host platform has been modified by Multispek, by removing the battery and exposing the shutter trigger and battery contacts to a common 3-pin servo connector.

The camera should be powered from an external 7V-24V DC power source, preferably not from the Pixhawk Power Module, as it does not have the required power to spare.

The trigger pin should be connected to any of the 6 auxiliary output pins of the Pixhawk. These can be configured by the ArduCopter to act as relay pins, for triggering camera shutters. With this method, ArduCopter is able to control the time interval between camera triggers and record their timestamp and location coordinates. Also, GCS software like Mission Planner is able to automatically produce the shutter trigger interval parameter, by taking into account the required overlap of the photos, the flight speed, the flight altitude and the camera lens specifications.

5.5 Communication and Image Transmission

The GoPro Hero 4 device is capable to transmit the captured images and video in real time with various methods.

1. Through a micro-HDMI port
2. Through a hosted WiFi connection, via a proprietary application for Android and iOS devices
3. Through a hosted WiFi connection, via a file server

For this application, the captured images must be transmitted to the GCS computer while the UAV is in flight for off-board processing. Thus, option 1 is outright rejected. Option 2 is rejected as well, because the proprietary application does not allow for custom processing code to be run, and besides, the images must be directed to a laptop computer, not a handheld device (such as a tablet).

Option 3 remains as the only valid alternative. The GoPro raises an otherwise common WiFi network, with a visible SSID and a password requirement, all configurable. Unfortunately, the GoPro does not have the ability to join a pre-existing WiFi network, it can only host one and have other devices join onto it.

The file server is raised at the IP `10.5.5.9:8080`, reachable by any device joined on the network.

Each time an image is captured, it is stored at the directory `videos/DCIM/110GOPRO`. From there, the images can be accessed programmatically, as explained in section 8.11.

As a downside, it should be noted that the range of the WiFi network the GoPro Hero 4 raises is relatively slow, extending only up to a tens of meters. The transfer rate falls proportionally with the signal strength, starting at a few MB/s and dropping as low as a few kB/s or lower as short as a few tens of meters away. This constraint should be taken into account when deploying the system.

Chapter 6

The Rover

The proposed precision agriculture system intervenes to the crops by using a ground-based vehicle, instead of tasking a UAV to carry and operate the corresponding equipment.

This approach has drawbacks and advantages. On the down side, a ground vehicle (commonly referred to as a rover) is limited in its movement by the terrain. Obstacles must be avoided or navigated. Rough terrain may impede or inhibit the path of the rover. The velocity at which the rover can move in a field is also limited. Its size may be constrained by the geometry of the crop rows and the shape of the plants. If the application calls for it, it must be delicate enough not to damage the crops during its mission.

On the up side, a rover is generally capable to lift a much heavier payload than a UAV. In the case of precision agriculture, this usually refers to pesticide or water tanks and sprayers, whose weight is significant. A rover is much more energy-efficient and resistant in harsh environments, if built properly. Also, for the same payload capacity, a rover is much cheaper and easily maintained, compared to a UAV.

The arguments supporting the choice of a rover for the intervention task outweigh those against it, and consequently a rover was used in this work.

6.1 The Rover Platform

Since COTS rover platforms, suitable for robotics applications are much less common, the rover for this work needed to be constructed from the beginning. Considering the heavy-duty nature of the task it had to undertake, a rugged, heavy platform was constructed (Figure 6.1).

The body was constructed out of 3 mm thick sheet metal and welded. Its wedge-like shape allows it to overcome uneven terrain. Its wheels have embedded bearings and provide adequate ground clearance.

Figure 6.1: The rover built for this work

It is powered by a 8200 mA h 3-cell LiPo battery, attached onto two Phoenix ESCs, driving two DC motors. Each motor is a low-RPM, high torque motor and is paired to both wheels of each side of the rover, via a thick timing belt. This allows the vehicle to steer in a tank-like fashion, by using differential wheel speed to accelerate and turn. This configuration removes the need for a mechanical steering system, reducing complexity and adding maneuverability, a highly required feature.

A wooden shelf was installed inside the rover for the ESCs and autopilot to be mounted upon. The battery is mounted with velcro on the side wall, inside the vehicle. The motors are directly mounted on the rover chassis (Figure 6.2). For more details and explanation on the image annotations, see the list below.

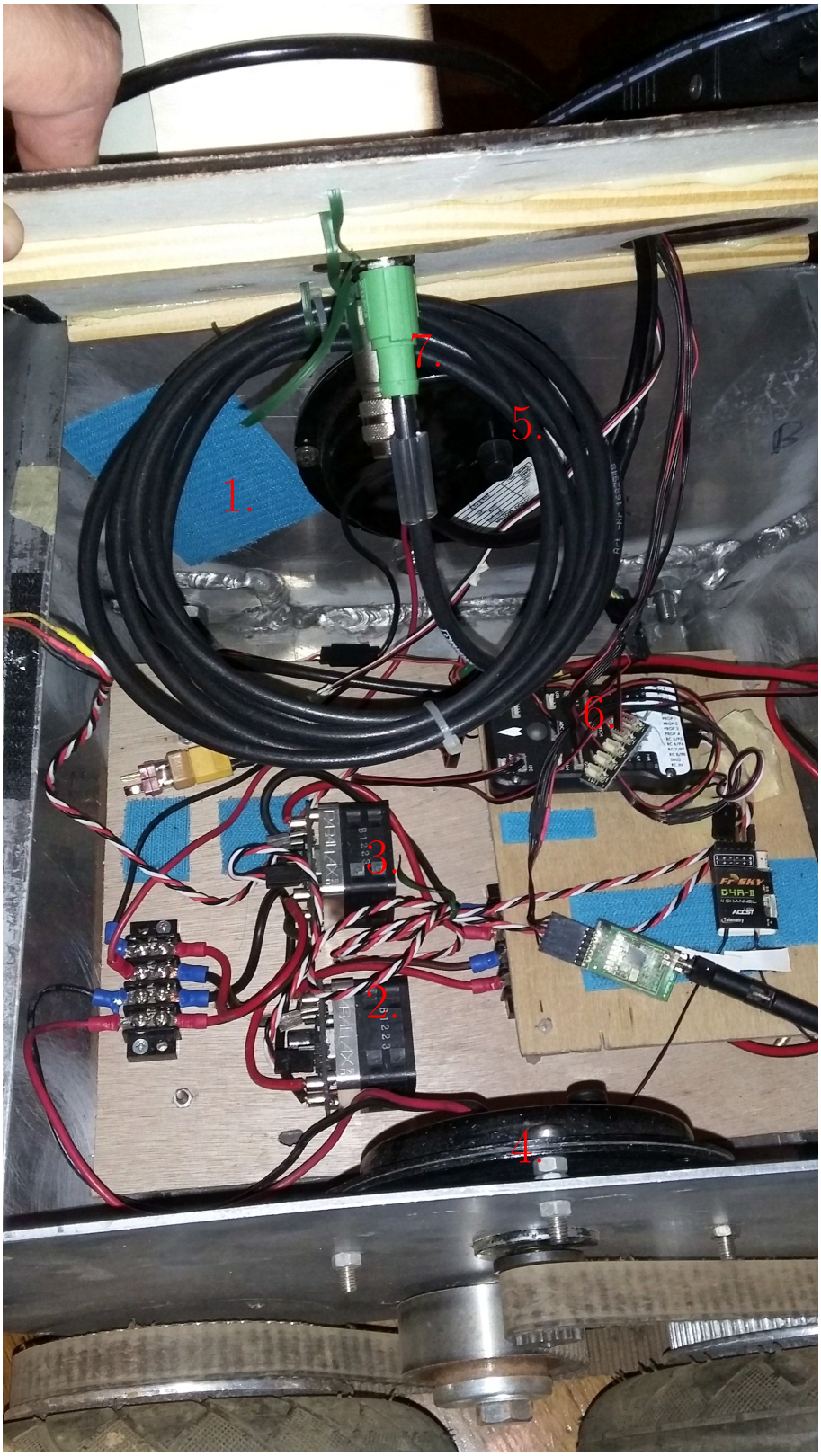

Figure 6.2: Electronics installation inside the rover

Legend for Figure 6.2:

1. Velcro®for battery mounting
2. Right motor driver
3. Left motor driver
4. Right motor
5. Left motor
6. Pixhawk autopilot
7. Laser scanner data and power cables

Some electronic devices need to protrude from the top of the rover. These are the laser scanner (presented in Section 6.3), the GPS receiver, the magnetometer and the WiFi antenna, each one for its own reasons:

- The laser scanner needs to have unobstructed line of sight in front of the vehicle to detect obstacles.
- The GPS receiver must be able to receive signals from as many GPS satellites as possible, and the metal chassis blocks their signal.
- Metal objects create soft magnet interference to magnetometers, and consequently, the magnetometer sensor must be moved as far away as possible from the metal chassis.
- Radio signals are heavily affected and warped by metal objects, thus all antennae must have clear line of sight towards the GCS, and be relatively separated from the chassis.

For the above reasons, another wooden platform was used to host top-level devices. It was designed with CAD software and cut from 4 mm plywood with a Computer Numberical Control (CNC) laser cutter. The overall layout is visible in Figure 6.3. For more details and explanation on the image annotations, see the list below.

Legend for Figure 6.3:

1. Laser scanner
2. GPS receiver with embedded compass
3. Embedded computer

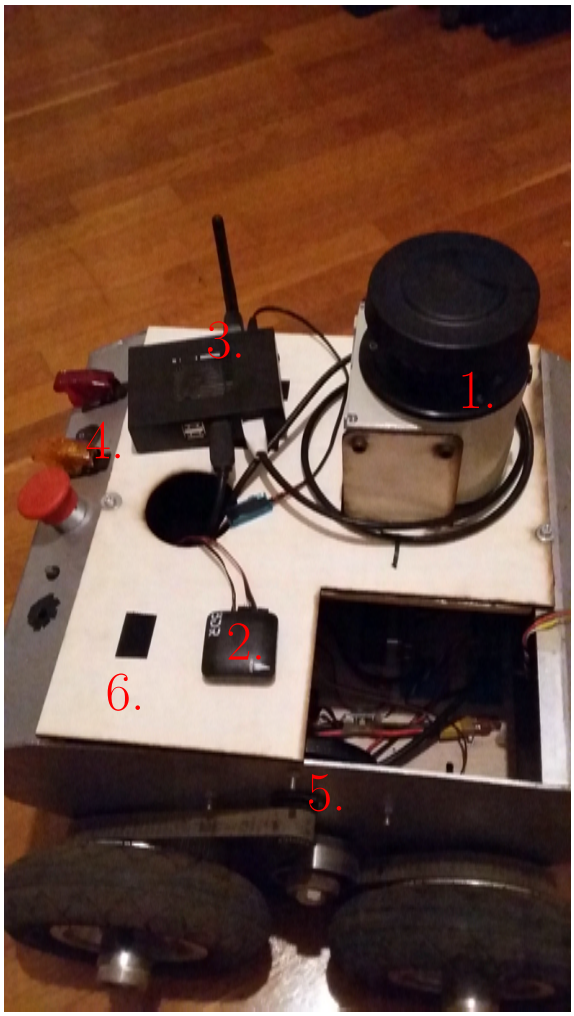

Figure 6.3: Electronics installation on top of the rover

4. Operation switches, from top to bottom: master switch, electronics switch, motors switch, motors kill switch

5. Motor pinion gear, timing belt and tensor

6. Laser-cut, plywood platform

6.2 The Autopilot

The Ardupilot software is capable of commandeering a ground vehicle, in its dedicated firmware form ArduRover. Much of the user interface and functionality is the same as its aerial counterparts, with manual, aided and fully autonomous waypoint missions functionalities being supported.

A Pixhawk autopilot system was thus installed inside the rover. A Power Module was attached to the battery to power the Pixhawk and monitor power consumption.

Since we have expanded upon Ardupilot in the UAV chapter, no further mention will be given in this chapter.

6.3 The Laser Scanner

Obstacle avoidance is a mandatory feature for the rover, for several reasons.

First and foremost, the field geometry can not be realistically mapped and known beforehand. Crop rows and unexpected obstacles can be impassable from the rover, which must have the ability to navigate around them.

Conversely, the the crops themselves must be protected by the heavyweight rover, which must be able to avoid them, so as not to damage them during its mission.

Finally, even if the field is mapped in every detail, the GPS-based location system introduces enough position error to result in undesirable situations: a position estimation error as low as a few decimeters may be enough to make the rover believe it is in the next crop row as the one it actually is in, and result it in trying to reach its next waypoint through a crop row.

For the above reasons, an obstacle avoidance solution must be implemented in the rover. The most common sensor used in this situation is a rangefinder. However, a beam-type sensor is not suitable for this application.

Such sensors are hard-mounted on the chassis and measure the distance of the nearest object only in one direction. If placed facing to the front of the rover (a reasonable assumption), a crop row coming at a shallow angle towards the side of the vehicle may not be detected in time. Furthermore, if the beam was wide enough to account for this problem, the problem of false-positives emerges, if crop rows are too narrow.

Moreover, with a single rangefinder, it would be impossible to discern the direction at which the obstacle is presented; not enough information exists, for the rover to know which direction to turn to to perform avoidance.

To deal with these problems, a laser scanner was installed on top of the rover, facing forwards. Its large scanning angle allows for the detection of objects in a wide range in the direction of motion and its detailed angular resolution makes finding the direction of the row possible.

Regarding its principle of operation, a laser scanner has a rotating head, equipped both with a laser emitter and a sensor sensitive to the wavelength of the laser emitter. While rotating at constant speed, the laser emitter generates laser pulses at regular angular intervals. The sensor receives the reflection of the laser beam and

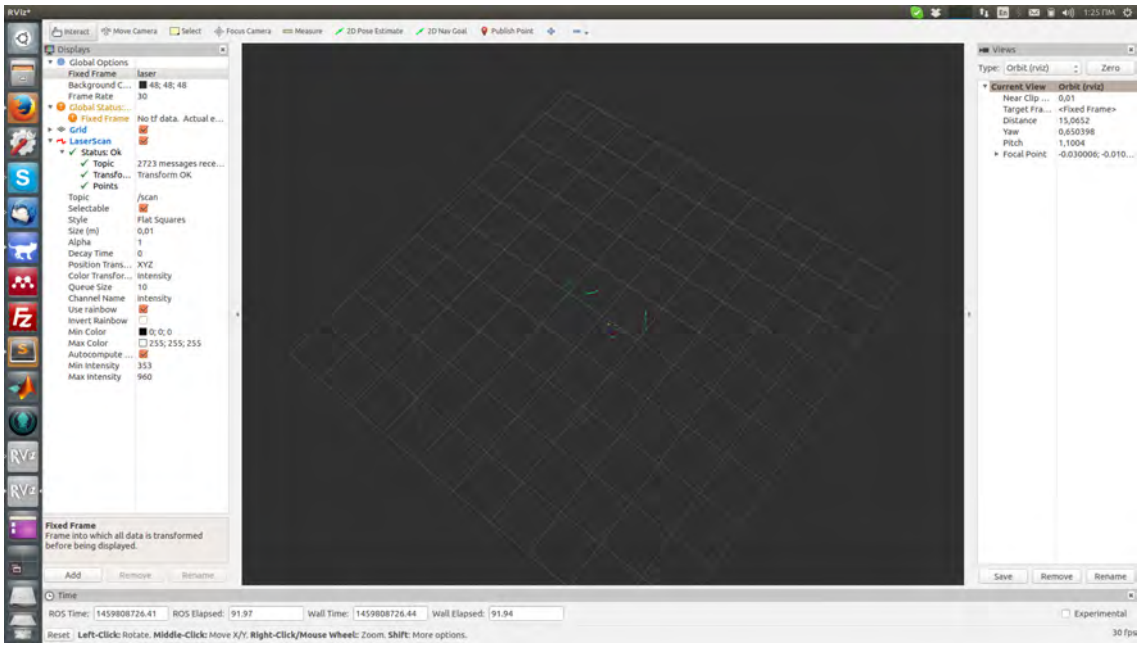

Figure 6.4: A typical laser scan frame

based on the propagation delay, the device estimates its distance from the reflecting object. A visualization of a typical laser scan frame can be seen in Figure 6.4.

The exact model used in this work was a SICK LMS111 [70] (Figure 6.5). It is a high-end, outdoors laser range scanner with the following features:

- Light source : Infrared (905 nm)
- Aperture angle : 270°
- Scanning frequency : 25 Hz
- Angular resolution : 0.5°
- Operating range : 0.5 m - 20 m
- Interfaces : Serial, Ethernet

Laser scanners have high sampling rates and provide large amounts of data. In order to acquire and process it, a potent computing system is required. Pixhawk does not support laser scanners and does not have the necessary computational power to handle them.

For that reason, an external, on-board computer must be installed and used to interface with the laser scanner. This solution can take advantage of the already existing software drivers. This computer must be carried by the rover; the large

Figure 6.5: The SICK LMS1110 laser scanner

amount of data produced by the LMS1110 (about 220 kB/s) would clutter any WiFi link and would make off-board data processing in the GCS impractical.

The on-board computer must be light and energy-efficient, yet powerful enough. Thankfully, such a family of computers exists, commonly referred to as embedded computers.

6.4 The Embedded Computer

Traditionally, robotics applications were targeted towards static robotic manipulators and full-size ground and aerial vehicles. These platforms could allow for large computing systems (even mainframe-sized) with no penalty. However, with the advent of scaled UAVs, a robotics platform with very limited payload headroom, the need for lighter computation units became more imperative than ever.

Embedded computers are gaining ground as the host unit of modern robotics decision centers and algorithmic computational units. New, resource-intensive algorithms such as SLAM and image recognition create a constant demand for embedded computers with ample power and naturally, the market has responded with new products.

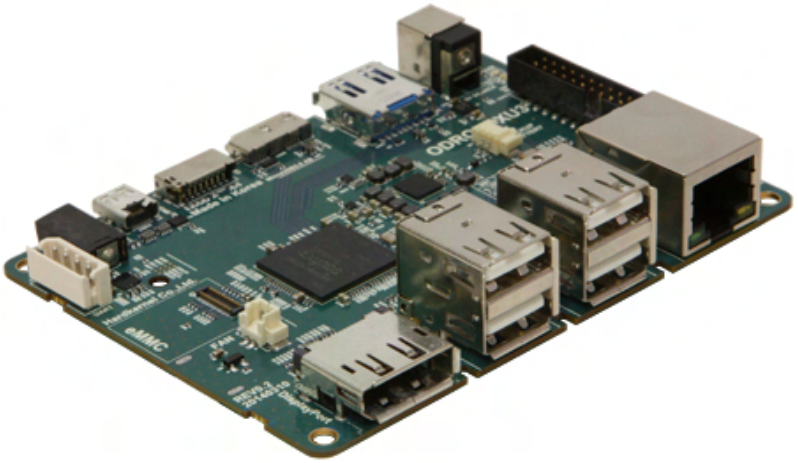

Figure 6.6: The Odroid XU3 Embedded Computer

While there is a multitude of competitive products, such as the Raspberry Pi, the Jetson TX1, the Beaglebone Black etc, we will not delve into a comparative discussion on the topic of embedded computers. This component is just a means to an end; as long as it can carry out the required tasks, it is considered suitable for our application.

Namely, it should be able to host an Ubuntu Linux operating system along with ROS, in order to take advantage of the already existing drivers for the laser scanner.

The Odroid [36] units satisfy our requirements having the advantages of being easy to use, cheap, powerful and with large community support. Specifically, the Odroid XU3 model was used (Figure 6.6).

Among its features, the most crucial ones are:

- Quad-core 2.0GHz processor
- 2GB RAM
- 6 USB ports
- Ethernet port
- 5V/2A power requirement

Apart from the hard requirement for the Ethernet port, to connect the laser scanner, the hardware amenities of the XU3 allow for more functionality to be migrated onboard the rover, making it a more independent unit.

First and foremost, the MAVLink stream form the Pixhawk can be fed to the XU3, through a Serial-to-USB converter cable. From there, off-board control of the rover

can be done on the XU3 with dedicated program threads, and the MAVLink stream can be re-routed to the GCS for visualization and logging purposes.

Functionality can be further expanded by attaching a WiFi USB dongle to the XU3, providing wireless communication capabilities.

Chapter 7

Elements of Theory

Apart from the technical presentation and selection of the hardware components of the system, the mathematical and algorithmic aspects of the system merit their own space. The theoretical support of this work is presented in this chapter, where a separate section is dedicated to discuss the algorithms which enable the system to perform its task.

7.1 Coordinate Frames

Within a robotics system, measurements and quantities are expressed in terms of different reference frames. In order to use these quantities throughout the system and for various calculations, they need to be converted from one frame to another. Thus, proper definition of these *coordinate frames* is important [2].

The most common coordinate frame used to refer to locations at and over the surface of the Earth is WGS84 [50]. It describes Earth as an oblate ellipsoid and thus its coordinates are (ϕ, λ) pairs. ϕ is the *latitude* angle and λ is the longitude angle. These two angles, ranging from -180 to +180 for λ and from -90 to +90 for ϕ, can pinpoint any location on the Earth's surface. Additionally, a third coordinate for height, h, can be used.

The problem with WGS84 coordinates is that they are expressed in degrees, which makes them incompatible with distance measurements on the surface of the Earth. For that reason, the UTM [83] projection has been introduced. This projection splits the longitude of the Earth into 60 zones. Inside each zone, a secant transverse Mercator projection is performed, to map the surface on a Cartesian grid. This grid can be used for distance operations in SI units. Furthermore, the error induced by the projection is less than 1/1000. The conversion between WGS84 and UTM coordinates is possible through operations of medium complexity, but thankfully software libraries which handle the conversion exist [11]. The conversion procedure is has millimeter-level accuracy.

Still, the UTM frame may be an overcomplication when one wants to deal only with the local displacement of a robotic platform, in respect to its initial location. For that purpose, the NED frame is used. NED is shorthand for North-East-Down and is a 3-dimensional Cartesian coordinate system and its origin is usually placed on the start location of the robot. Another valid option is to place the origin at the center of the area the robot is expected to operate. Its x-y plane is horizontal, tangent to the surface of the Earth at the origin. Its x-axis points North and its y-axis points East. As NED is a right-handed axes system, the z-axis positive points down, towards the ceter of the Earth. As a result, above surface heights have negative z-values. Another common name for the NED frame is the *inertial frame*.

The next frame, in the sequence of transformations is the *carried frame* (see Figure 7.1). This has the same orientation as the NED frame, but its origin is placed at the center of mass of the robot, thus the term *carried*. The carried frame is converted to the NED frame by a single translation by the position of the robot.

Another important frame is the *body frame*. Its x-axis is aligned with the longitudinal axis of the robot, which for aircraft it usually coincides with the forward direction. Its y-axis extends to the right of the robot and its z-axis to the bottom of the robot, completing a right-hand system. As the robot turns and banks, so does the body frame. This rotational transformation between the carried frame and the body frame can be expressed as 3 separate rotations.

There are many conventions for the description of this rotation sequence but the one used most often in robotics and especially in UAV applications are the Euler angles with the Tait-Bryan convention. The 3 Euler angles are roll, pitch and yaw, represented with the triplet (ϕ, θ, ψ). The Tait-Bryan convention dictates that in order to rotate from the carried frame to the body frame, we must do, in that order:

1. Rotate about the z-axis of the carried frame by ψ, until the x-axis is in the same plane as the body-frame x-axis, resulting in a new, intermediate frame.

2. Rotate about the y-axis of this intermediate frame by θ, until the x-axis coincides with the body-frame x-axis, resulting in a new, intermediate frame.

3. Rotate about the x-axis of this intermediate frame by ϕ, until the y-axis coincides with the body-frame y-axis.

Those three angles can be used to construct a rotation matrix, which is defined as

$$\mathbf{R}_v^b = \begin{bmatrix} c_\theta c_\psi & c_\theta s_\psi & -s_\theta \\ s_\phi s_\theta c_\psi - c_\phi s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & s_\phi c_\theta \\ c_\phi s_\theta c_\psi + s_\phi s_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi & c_\phi c_\theta \end{bmatrix} \tag{7.1}$$

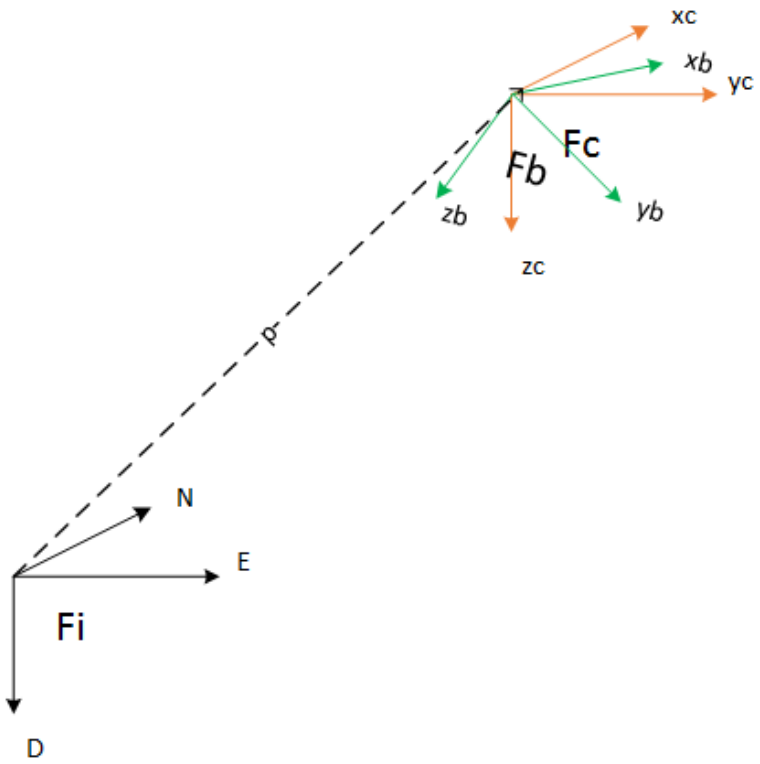

Figure 7.1: The NED, carried and body frames

Where

$$s_x = \sin(x)$$
$$c_x = \cos(x)$$

$\mathbf{R}_v^b$ can be used to convert vector quantities from the inertial (or carried) frame to the body frame and vice versa. For example, the gravity vector, which in the inertial frame is usually expressed with only a vertical component

$$\mathbf{F}_{g,i} = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{7.2}$$

is "felt" from on-board the UAV as

$$\mathbf{F}_{g,b} = \mathbf{R}_v^b \mathbf{F}_{g,i} \tag{7.3}$$

$$= \begin{bmatrix} c_\theta c_\psi & c_\theta s_\psi & -s_\theta \\ s_\phi s_\theta c_\psi - c_\phi s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & s_\phi c_\theta \\ c_\phi s_\theta c_\psi + s_\phi s_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi & c_\phi c_\theta \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{7.4}$$

$$= mg \begin{bmatrix} -s_\theta \\ s_\phi c\phi \\ c_\phi c_\theta \end{bmatrix} \tag{7.5}$$

Conversely, if we want to convert a vector from the body frame to the inertial frame, we can multiply it with the transpose of the rotation matrix:

$$\mathbf{p}_i = \mathbf{R}_v^{b\intercal} \mathbf{p}_b = \mathbf{R}_b^v \mathbf{p}_b \tag{7.6}$$

There are other methods of representing rotations,such as quaternions [65], but rotation matrices built by Euler angles are very common. Rotations allow us to capture vector measurements taken from on-board a robot, such as accelerometer and laser scanner readings, and convert them to the inertial frame, for use in various algorithms. Common applications are orientation estimation and localization.

Since camera readings and image recognition also play a major role in this work, we will introduce two more frames of reference. See Figure 7.2 for visual comparison.

The camera frame has its origin at the camera sensor, and its z-axis outwards from the camera lens. The x- and y- axis usually are placed to the right and downward respectively. Since in our application the camera is mounted on the bottom of the UAV facing downward, the Euler angles from the body frame to the camera frame are

$$\mathcal{E}_{cam}^b = (0, 0, 90) \tag{7.7}$$

Finally, the image frame is a 2-dimensional frame, corresponding to the projection of the environment on the camera sensor as a 2-dimensional plane. The image frame is set in front of the camera origin by the focal distance f. The x- and y-coordinates describe the positions of objects as they are captured on the image.

The objects coordinates are scaled by their distance to the image frame. A point with coordinates

$$p_{cam} = (x_p, y_p, z_p)$$

in the camera frame, will be imprinted on the image frame at the coordinates

$$p_i = \left(\frac{f}{z_p} x_p, \frac{f}{z_p} x_p \right) \tag{7.8}$$

Conversely, if we know the distance of the object to the camera and its position in the image frame, we can invert the projection to find its location in the camera frame.

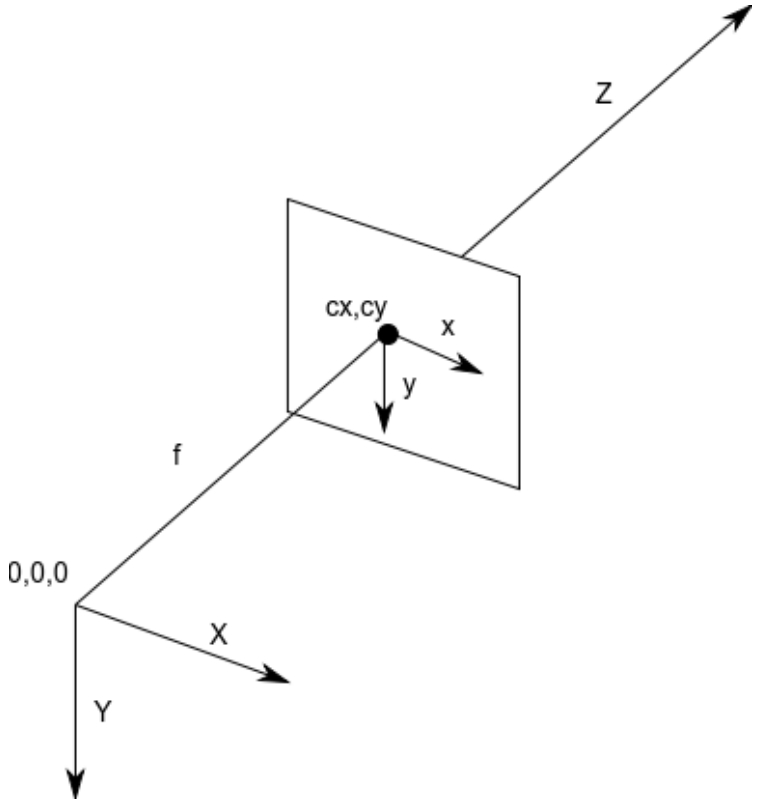

Figure 7.2: The camera and image frames

7.2 Rover Path Generation

Part of the end goal of this work is to enable a rover to navigate through a field with bush vegetation planted in crop rows and visit specific points inside of it. As is discussed in Section 6.3, the rover must not collide with the vegetation but rather navigate around it. This is especially important when the rover needs to visit a point in a different crop row than the one it currently is.

It is assumed that the vegetation forms impassable rows. This is a common occurrence, for example in vineyards. As an example, see Figure 7.3, where plants form parallel crop rows (line R-R'). We also assume that the field contains at least 2 corridors (lines C1-C1' and C2-C2') which are larger than the space between the crop rows.

With this geometrical information, it is possible to specify a path from the current rover location to the target location, by adding two intermediate waypoints located on a corridor. Essentially, the rover will exit a crop row, turn on a corridor and re-enter the field at the desired row.

However, as is the case of the rover constructed in the premises of this work, it is not guaranteed that the rover will be able to turn around its body-z axis. This ability

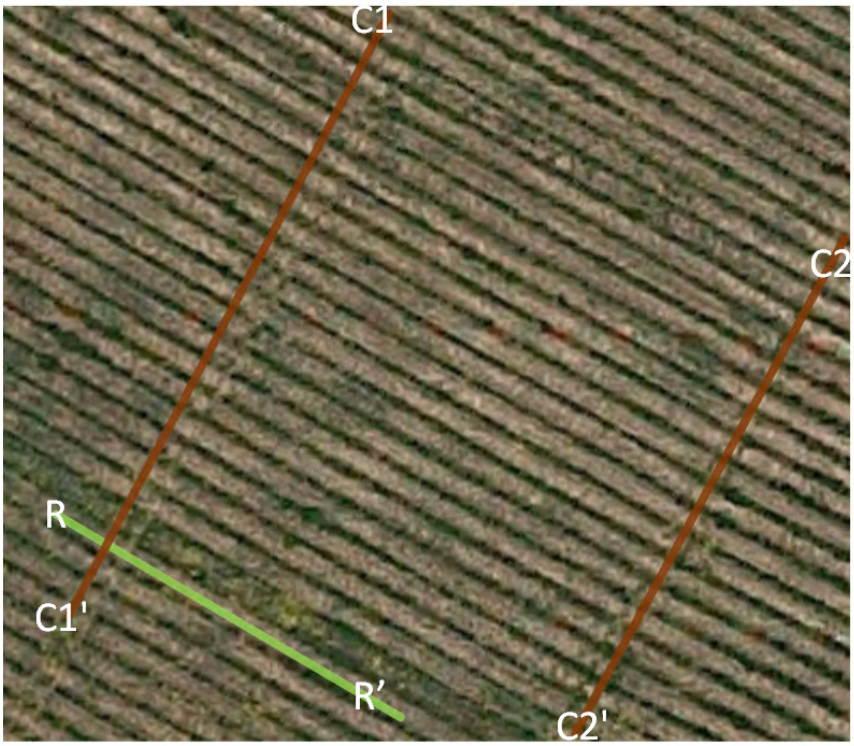

Figure 7.3: A Typical Crop Field Geometry

depends on the locomotion system configuration as well as the motor power and ground friction. This imposes another constraint: The rover must exit a crop row towards the direction it is currently facing. See Figure 7.4 for more details.

We examine two cases: One where the rover starts facing generally northwards (S) and one where the rover starts facing southwards (S'). In both cases, the target location is G.

In the first case, the rover must continue up to point E, reaching the corridor, turn to F, reach it and then turn again in the crop row to reach G.

In the second case, the steps are similar, but the points E and F are replaced with their counterparts E' and F', situated in the corridor on the other side of the field.

It is clear that a detailed strategy must be defined for the generation of the intermediate points E and F.

We start with the assumption that we know the coordinates of points A, B and C. A and B are situated on different corridors and on the ends of the same crop row. C is situated on the same corridor as B. Let us define the coordinates of A, B and C as:

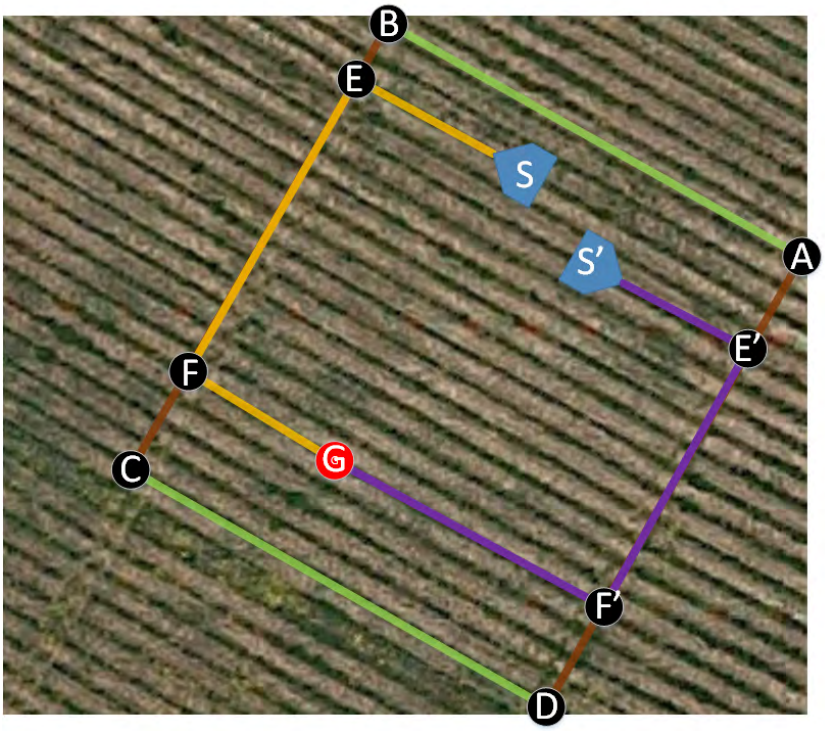

Figure 7.4: The Proposed Path Geometry

$$\mathbf{A} = (A_{lon}, A_{lat}) \tag{7.9}$$
$$\mathbf{B} = (B_{lon}, B_{lat}) \tag{7.10}$$
$$\mathbf{C} = (C_{lon}, C_{lat}) \tag{7.11}$$

Working on the NED frame and using the above information, we can define, on the 2-D horizontal plane, lines describing the slope of the crop rows and the lines of the two corridors.

The line AB has the equation

$$y = l_{AB}x + c_{AB} \tag{7.12}$$

where:

$$l_{AB} = (B_{lat} - A_{lat})/(B_{lon} - A_{lon}) \tag{7.13}$$

and

$$c_{AB} = A_{lat} - l_{AB}A_{lon} \tag{7.14}$$

Similarly the line BC is

$$y = l_{BC}x + c_{BC} \tag{7.15}$$

with:

$$l_{BC} = (C_{lat} - B_{lat})/(C_{lon} - B_{lon}) \tag{7.16}$$

and

$$c_{BC} = B_{lat} - l_{BC} B_{lon} \tag{7.17}$$

We can also easily define the line AD as:

$$y = l_{AD} x + c_{AD} \tag{7.18}$$

with

$$l_{AD} = l_{BC} \tag{7.19}$$

and

$$c_{AD} = A_{lat} - l_{AD} A_{lon} \tag{7.20}$$

We also define another two auxiliary lines, parallel to AB, one crossing point S (or S') and another crossing G.

The first one has the equation

$$y = l_S x + c_S \tag{7.21}$$

where:

$$l_S = l_{AB} \tag{7.22}$$

and

$$c_S = S_{lat} - l_S S_{lon} \tag{7.23}$$

and the second is

$$y = l_G x + c_G \tag{7.24}$$

where:

$$l_G = l_{AB} \tag{7.25}$$

and

$$c_G = G_{lat} - l_G G_{lon} \tag{7.26}$$

We can calculate E as the point where line BC and the one crossing S intersect:

$$E_{lon} = (c_{BC} - c_S)/(l_S - l_{BC}) \tag{7.27}$$

$$E_{lat} = l_S E_{lon} + c_S \tag{7.28}$$

and F as the point where line BC and the one crossing G intersect:

$$F_{lon} = (c_{BC} - c_G)/(l_G - l_{BC}) \tag{7.29}$$

$$F_{lat} = l_G F_{lon} + c_G \tag{7.30}$$

Conversely, points E' and F' can by found as the intersections of line AD with the lines crossing S and G:

$$E'_{lon} = (c_{AD} - c_S)/(l_S - l_{AD}) \tag{7.31}$$
$$E'_{lat} = l_S E'_{lon} + c_S \tag{7.32}$$

$$F'_{lon} = (c_{AD} - c_G)/(l_G - l_{AD}) \tag{7.33}$$
$$F'_{lat} = l_G F'_{lon} + c_G \tag{7.34}$$

The decision on whether the path S-E-F-G or S-E'-F'-G should be generated can be made based on the heading of the rover, which is available information. If the rover is heading northwards, the first path is generated and followed. Otherwise, the second one is followed.

7.3 Obstacle Avoidance

The importance of collision avoidance for this specific robotic application has already been mentioned in previous sections. The rover must not under any circumstances collide with the surrounding vegetation. One way of achieving this would be to have an exact, accurate map of the crop row formation as well as knowledge of the precise position of the rover. With this information, a path planning algorithm would be able to produce a collision-free path, by having access to a high-level of detail of the problem inputs. After such a path is generated, it should be accurate enough to be collision-free. Essentially, this is an open-loop approach to the problem.

However, there are several problems with this approach. First and foremost, such a high-level of detail in the depiction of the field would require an enormous amount of effort from human operators, with the related cost. Still, the captured information would quickly go out-of-date, in a dynamic environment such as a field with live plantation.

Moreover, in order to access location information for the rover with centimeter-level accuracy, expensive hardware infrastructure is required; the standard GPS receivers provide an accuracy of only a few meters.

Thus, it is evident that this approach to path planning would quickly fail, due to the high degree of both process and measurement uncertainty.

Sensor Description

A closed-loop method is required for robust obstacle avoidance. To this end, a planar laser scanner sensor is used in order to provide real-time readings of the surrounding obstacles with centimeter-level accuracy. Without getting into technical details in

this section, a laser scanner sensor is able to detect the distance of the nearest obstacle in each angular segment around it, over an angular range. Characteristics of performance for this type of devices are the scan angle range, the angular resolution and the distance measurement range.

An example is given for the used SICK LMS111 sensor, so as to better explain the nature of each specification. An angular range of 270 degrees would signify that the sensor can detect obstacles as far as 135 degrees to the left of it and 135 degrees to the right of it. It would be blind to any obstacles outside this range.

The angular resolution specification reflects the angular step between successive distance measurements and is an index of distinguish-ability for the sensor. A 0.5 degree resolution would mean that the sensor takes a new distance measurement every 0.5 degree. In conjunction with the angular range specifications, it is calculated that at each scan frame, the sensor takes 540 distance measurements.

Finally, the distance range expresses the limits of the distance measurement, which often has a minimum and maximum value. Our sensor may have a 0.5m - 20m measurement range.

Mathematically, the measurement vector for each scan can be annotated as

$$d(\phi, t) \tag{7.35}$$

with

$$\phi = n \cdot d\phi \tag{7.36}$$

$$n \in [\phi_{min}/d\phi, \phi_{max}/d\phi] \subset \mathcal{Z} \tag{7.37}$$

$$d(\phi) \in [d_{min}, d_{max}] \tag{7.38}$$

Equation 7.36 reflects the angular resolution of the senor, with n being the discretization index and $d\phi$ being the angular resolution. Equation 7.37 reflects the angular limits of the sensor. Equation 7.38 reflects the measurement range of the sensor.

For the SICK LMS111 sensor it would be $d\phi = 0.5degree$,
$(\phi_{min}, \phi_{max}) = (-135degrees, 135degrees)$ and $(d_{min}, d_{max}) = (0.5m, 20m)$.

The interaction of the sensor with the environment can be seen in Figure 7.5, where the rover is placed between two crop rows, depicted as green rectangles. The angular range of the sensor is depicted as an incomplete circle centered on the sensor, with a radius equal to the maximum measurement range. A number of distance measurements are drawn as dashed red lines, ending on the nearest obstacle or at the end of the range.

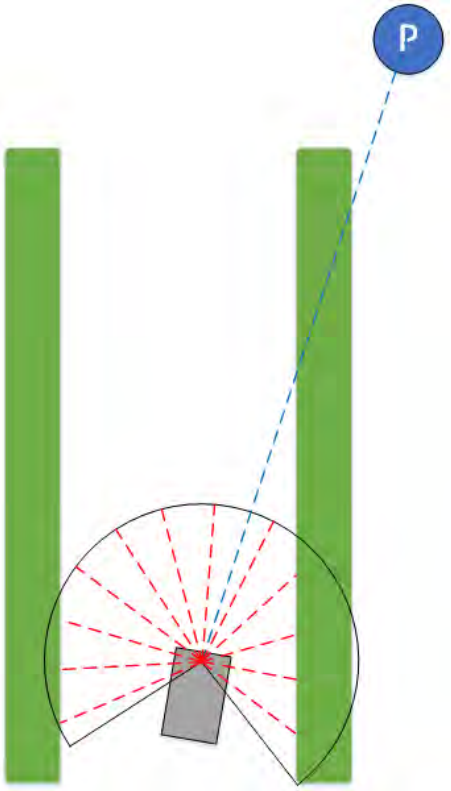

Figure 7.5: The obstacle avoidance and navigation problem geometry

Vector field approach

Given the available information of the desired target position and the obstacle positions with respect to the rover, we must devise a control strategy for path generation. Among many candidate solutions, a simple and tested method is that of the vector field. First, we introduce the *potential fields* as a path planning approach, which, for each point in space, constructs a reference vector which the robot should align itself with during its motion. In this manner, by dictating the direction of the robot's motion, the potential field can shape the path it will follow (see Figure 7.6) (Image taken from [12]).

It should be noted that potential fields usually have two defining functions: one which produces the vector norm at any location and another for the vector direction. The vector norm can be used to create a reference velocity for the trajectory. However, in this work, the velocity control is chosen to be performed by the low-level autopilot, not the trajectory manager. Thus, vector fields are used instead, which are different from potential fields in that they do not necessarily represent the gradient of a potential. Rather, the vector simply indicates a desired direction of travel [52].

At each point, the reference vector direction can be constructed by taking into account various conditions which affect the resulting trajectory. Each condition results

to a component vector field, which can be summed up with others to produce the final field.

The most common component is the attraction field, created by the target waypoint $\mathbf{p} = (x_p, y_p)$, the destination. The corresponding vector field at any location $\mathbf{r} = (x, y)$ should construct an attractive field towards the destination. Usually, a constant-norm vector is produced, which points towards the destination:

$$\mathbf{V}_p(\mathbf{r}) = k_p \frac{\mathbf{p} - \mathbf{r}}{|\mathbf{p} - \mathbf{r}|} \tag{7.39}$$

or in Cartesian form

$$\begin{bmatrix} V_{p,x} \\ V_{p,y} \end{bmatrix} = \frac{k_p}{\sqrt{(p_x - r_x)^2 + (p_y - r_y)^2}} \begin{bmatrix} p_x - r_x \\ p_y - r_y \end{bmatrix} \tag{7.40}$$

To accomplish obstacle avoidance, another vector field is created, which points away from known obstacles at positions $\mathbf{o}_i = (x_{o,i}, y_{o,i})$. Its magnitude should be greater, as the robot approaches the obstacle, to ensure that the repulsive contribution will become strong enough to overcome the attractive force towards the destination. Usually, an inverse distance law is used:

$$\mathbf{V}_{o,i}(\mathbf{r}) = k_o \frac{\mathbf{r} - \mathbf{o}_i}{|\mathbf{o}_i - \mathbf{r}|} \tag{7.41}$$

The dividing distance could very well be raised to some power, to make the increasing force effect steeper, with decreasing distance from the obstacles.

The sum

$$\mathbf{V}(\mathbf{r}) = \mathbf{V}_p(\mathbf{r}) + \sum_i \mathbf{V}_{o,i} \tag{7.42}$$

produces the overall reference direction. As discussed above, the magnitude of this vector is disregarded and the velocity control is left to the low-level autopilot.

Constants k_p and k_o should be tuned appropriately, so that the correct balance between attraction and repulsion is achieved.

In Figure 7.6 it is evident how the destination point produces a field component which draws the trajectory close to it, especially in locations far from the obstacle (black circle). Close to the obstacle, the repulsive field component is so strong that overpowers the attractive force.

In Figure 7.7, the application of the vector field methodology for a given rover position is visualized. The blue attraction vector points towards the destination point, annotated with a blue "P" point. If only this component contributed to the reference heading, then the rover would collide to the crop row.

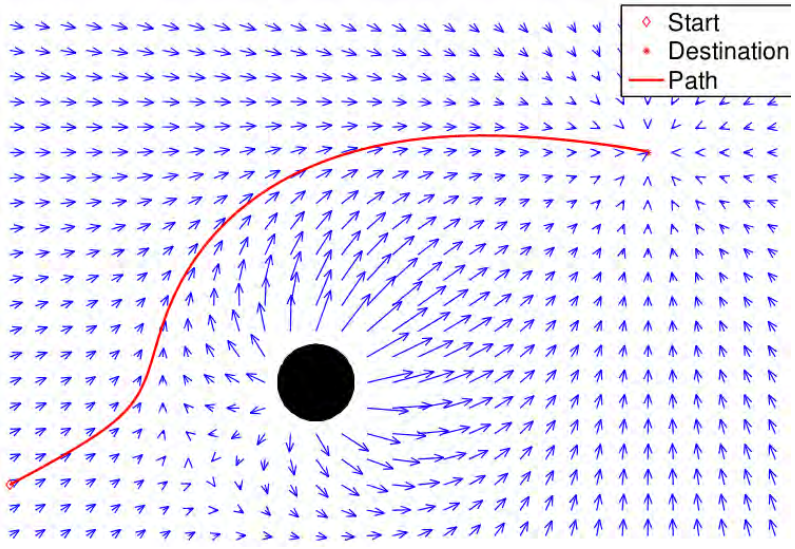

Figure 7.6: An example vector field

In this work, the repulsion vector used is

$$\mathbf{V}_{o,\phi}(\mathbf{r}) = k_o \frac{\mathbf{R}_i^b \begin{bmatrix} -cos\phi \\ -sin\phi \end{bmatrix}}{d(\phi)^2} \tag{7.43}$$

A first-pass value assignment for the vector constants could be:

$$k_p = 1 \tag{7.44}$$

$$k_o = \frac{1}{250} \tag{7.45}$$

The various object distance measurements create the repulsive vectors $\mathbf{v}_{o,i}$, each denoted as a red, dashed arrow. Their sum is the red, solid arrow.

The blue and red components are summed to the green overall vector field reference heading. The reference vector is the result of the operation

$$\mathbf{V}(\mathbf{r}) = \mathbf{V}_p(\mathbf{r}) + \sum_{\phi} \mathbf{V}_{o,\phi} \tag{7.46}$$

Along the heading of $\mathbf{V}(\mathbf{r})$, a tracking point can be placed (annotated as a green "T" circle) at a pre-defined distance and communicated to the low-level autopilot. The autopilot will try to direct the rover towards it.

Mathematically, the position of the tracking point could be expressed as

$$\mathbf{T}(\mathbf{r}) = L \frac{\mathbf{V}(\mathbf{r})}{|\mathbf{V}(\mathbf{r})|} \tag{7.47}$$

where L is the lead distance of the tracking point from the vehicle.

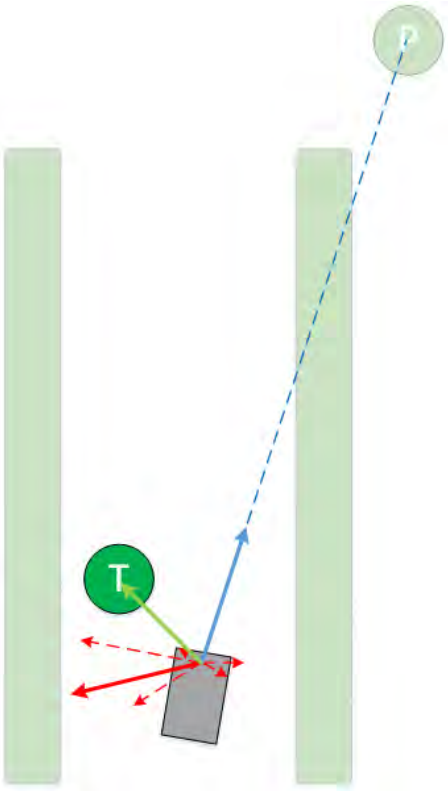

Figure 7.7: Vector field synthesis

Chapter 8

Software Components

Modern robotic systems tend to be large and complex, spanning multiple environments and networks. They take advantage of knowledge from multidisciplinary fields and make use of a diverse set of computational tools. This is a natural effect of the fact that as robotics evolves and is called to solve more intricate problems, robotic systems become larger and more involved, in order to face new problems and challenges.

Similarly to the programmers, who maintain software libraries and build upon them, roboticists often try to create hardware and software modules. These modules implement specific, well defined functionalities and can be combined relatively easily to construct a robotic system of larger scale. This alleviates the burden of re-solving every robotics problem from roboticists, each time they start building a new system.

It is considerably harder to standardize hardware components and device drivers, because they reflect the unique aspects of each project and device. For example, it is impractical to re-purpose a rover design which was used for defusing bombs to be part of a precision agriculture system; The latter application calls for a cost-efficient, modular solution, while the first requires a robust and very precise platform.

Still, on the software side of things, there has been significant progress in the development of software modules for robotics applications.

8.1 The Robotics Operating System (ROS)

Perhaps the most significant contribution towards the standardization of robotics constructs is ROS [63]. ROS is a meta operating system which provides utilities for executing and managing multiple user-written programs as well as establishing a pipe-based communication system among them, using TCP. Furthermore, it facilitates the compilation and linking of source code and libraries, under its build system, called *catkin*.

ROS is basically intended to support the Ubuntu Linux Operating System (OS), but there is also limited support for Windows and some embedded system architectures.

Roscore

Each time ROS executables need to be run on a computer, one ROS program must be run before all. This program is called the `roscore`. Only one `roscore` can be running at each time in any computer. `roscore` is tasked with arbitering all subsequently launched ROS programs (called *nodes*), monitoring their execution and organizing their inter-communication.

Nodes

The user can write ROS programs (usually referred to as *nodes*) in Python and C++, taking advantage of both languages' strong points.

Python nodes can benefit from the vast number of libraries (also known as modules), suitable for scientific calculations in a high-level abstraction. Notable examples are the *numpy* module for matrix manipulation and the *openCV* library for image processing.

C++ nodes can be compiled very efficiently to produce fast code, ideal for implementing device drivers or writing low-latency loops. Fast processing of the large amount of data produced by a laser scanner or execution of fast control loops are some examples where C++ would be a good choice of programming language.

It is recommended to split the functionality of a robotics project into multiple nodes, so that each one contains a stand-alone body of code, which can run on its own thread and exist separately of other nodes.

Topics

ROS uses a pipe-based system to arbiter communications between nodes. Each pipe is called a *topic* and must contain only one type of variable structure. Variable structure types are called *messages* is ROS terminology. There are stock message definitions for the most common data structures, but custom messages can be described as well. Each node can either subscribe on one or more topics and/or publish onto them.

Pipes are queue-based, buffered structures, meaning that a nodes can publish more than one message, filling the pipe gradually, before another node starts popping messages from that pipe and processes them.

Topics are created by objects belonging to nodes, called *publishers*. Publishers are created with a new topic name and a topic type as arguments. They inform the

roscore of the creation of a new topic and interface with him for the exchange of messages. They also provide methods for message publishing.

Conversely, objects for subscribing to topics (i.e. receiving messages) exist, called *subscribers*. These are also assigned on nodes and given a topic to monitor and a callback function as arguments. Each time a new message appears on a topic, the subscriber will pop it and pass it on the callback function.

It should be noted that for C++ compiled nodes, subscriber objects only monitor topics for new messages during specific events during the node execution. That is, they are not asynchronous threads but run when `ros::spinOnce()` or `ros::spin()` functions are called.

Python nodes have dedicated threads for subscribers.

8.2 ROS Node/Topic Example Network

A simple ROS node and topic network can be seen in Figure 8.1. The image is a snapshot of the ROS network state at the time it was taken. Two nodes (executable programs) are seen running, depicted as ovals, `/turtlesim` and `/draw_square`. Each has its own, unique name. In this case, both start with a slash, but namespaces are supported, allowing for the same leaf name to be used under different prefixes.

Two topics are also observed in the form of directed lines, `/turtle1/pose` and `/turtle1/command_velocity`. `turtle1` is considered to be a namespace for these topics in this case. The direction of the arrows specifies the publisher/subscriber relatation between the topics and the nodes.

In this example, `pose` is published by `/turtlesim` and subscribed by `/draw_square` while `command_velocity` is published by `/draw_square` and subscribed by `turtlesim`.

Parameter Server

Commonly, coded functionality needs to act/operate based on the value of one or more parameters. Controller gains, IP addresses and delay intervals are some common examples where the behavior of a program depends on values which may need to change from execution to execution, based on the needs of the user. Compiled ROS programs cannot have their execution parameters changed, unless another re-compilation is issued. This is an unstable and time-consuming process.

ROS provides a convenient solution for this problem: the *parameter server*. Before the execution of a node, one or more parameters of specific value can be loaded to a dedicated node/server (always under the roscore supervision) represented by a name-value pair. These parameters can then be accessed during run-time by the nodes, through a dedicated API.

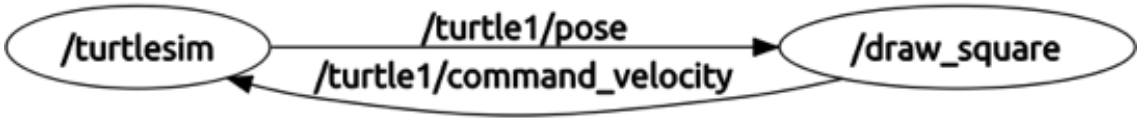

Figure 8.1: A simple ROS node and topic network

Multiple parameters can be loaded from text files, described in `.yaml` format.

Packages

The source files for the nodes, the custom message definitions, makefiles and other project-related files are organized in *packages*. Each package is a collection of files, usually referring to a well-defined program or functionality. For example, the source code required to operate a robotic arm can be enclosed in a package, from the device drivers to the high-level trajectory planning and computer vision programs.

The ROS build system, *catkin* parses each package and builds it, producing executable files. Packages can have interdependencies and provide libraries, to be used by other project-packages.

We usually say that an executable node belongs to a specific package.

File System

Each ROS installation has a standard file structure. The system files are automatically placed by the installer under the `/opt/ros/<ros-version>` directory as well as any other packages downloaded from the official package server.

However, the user files should be created and placed into the *workspace*. The workspace is a specifically declared folder under the user directory tree, where custom, proprietary applications are developed. It contains two major folders, `src` and `devel`.

The user can create a new folder under the `src` directory for each new package he wants to develop. Each package folder creates source code, scripts, launch files, package manifests and other assets, corresponding to that specific package.

Each time the compile system provided by the ROS installation is invoked, it scans the whole `src` directory tree and compiles all the packages it finds. The executable files that are created are placed under the `devel` folder, from where the `roscore` launches them.

Currently, the compile system used by ROS is called `catkin`.

Launch Files

Nodes can be launched via commands in a bash terminal, but in large robotics projects, this can quickly become cumbersome and error prone. ROS defines *Launch Files*, XML structures which belong to packages and describe an execution order and the arguments of multiple nodes, as well as topic renames and parameter loading.

The user can offload the node launching to a launch file and only issue its execution instead, alleviating most of the launching burden.

Launch files can also perform basic conditional block execution and include other node files, allowing for modular and flexible programs.

8.3 The multimaster-fkie Package

Often robotic systems need to span multiple computers and computing units, in a networked structure. Centralized systems with peripheral agents and decentralized systems fit this description and are very common in all but the most rudimentary projects. Thus, there is the need for co-operating nodes across different computers.

Let's assume that a networked system consists of two computers, A and B, linked with a WiFi connection through a router. Each node needs a roscore to operate under. One option is to launch roscore in A and declare in B that all nodes should be assigned to the roscore of A, which is supported and a viable option. However, in the case where the communication between A and B is intermittently lost, all nodes of B will hang during the communication outage and may crash. This is very unwanted behavior, especially in fast-moving robots which interact with the environment.

A solution to this problem is to launch a roscore in each A and B, each with its own nodes. However, this complicates communication between nodes of A and B, because standard ROS protocol does not allow for shared topics between roscores. The developer could try to write his own TCP, UDP, serial or other communication protocols, but this is the exact burden one is trying to avoid, by using ROS and the topic system.

multimaster_fkie [76] is a collection of 3rd party packages which offer exactly this functionality. It consists of two main nodes.

Running `rosrun master_discovery_fkie master_discover` under a roscore will raise a node which will look for and find other roscores in other computers in the same network. The nodes and topics of these roscores are identified.

Running `rosrun master_sync_fkie master_sync` subsequently allows for synchronization of topics and parameters between roscores. The topics of one roscore are available to the nodes of the other and vice versa.

This functionality really unlocks a lot of potential for distributed systems across networks. Issues with packet drops and communication delays still exist, but at least each computer is now able to run stand-alone code and maintain minimum failsafe functionality for that event.

8.4 The Image Transport Package

Typically, message packets are defined either by standard ROS types or by user definition files, in a similar manner as e.g. C language structures. Then topics of appropriate size can host them and they can be encoded and decoded by publishers and subscribers. All kinds of message types can be exchanged between nodes this way, from simple integer numbers to complex GPS and odometry information with dozens of fields.

However, there is a notable exception: images. Images have very large packet sizes which make their exchange difficult in networked systems, especially since multiple instances of each image are created, one for each node subscribed to its topic.

For that reason, the *Image Transport* [46] package was created, which alleviates some of the burden of image exchange from roscore, by using a more efficient exchange protocol, with less impact on the system memory.

On the downside, it requires the use of its own publisher and subscriber objects and the conversion of image message types to its own type.

8.5 The Image_Proc Package and Image Rectification

Every camera introduces distortion to the captured image, due to lens imperfections and misalignment. This is mostly visible as a "bending" of straight lines the further they are from the center of the image (see Figure 8.2) and is a problem for computer vision applications which rely on geometric measurements upon the image coordinates.

Thankfully, calibration and rectification of the camera distortion is possible and in ROS it can be done using the *image_pipeline* package [56] and its sub-packages.

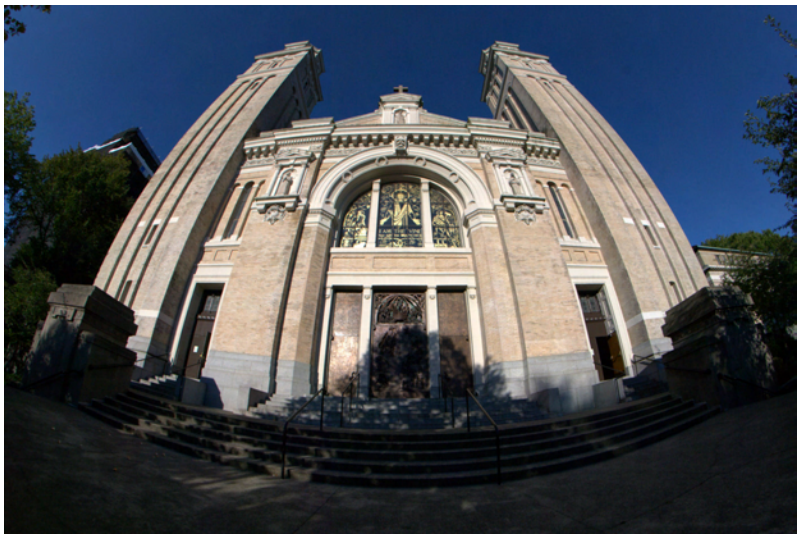

Figure 8.2: A demonstration of lens distortion

Figure 8.3: The lens calibration chess pattern

A method for camera calibration is provided by the `camera_calibration` package. The user prints a pre-defined chess pattern (Figure 8.3) and places it in front of the camera, in various positions, distances and orientations. The calibration algorithm compares the visible result with the expected correct image projection and constructs the distortion coefficients matrix.

Once the distortion coefficients are calculated, the `image_proc` node can be run, which handles image rectification. With its default arguments, the `image_proc` node subscribes to the `image_raw` topic for the image feed and to the `camera_info` topic for the camera calibration data and produces the `image_rect_color` topic with the

rectified image.

8.6 The OpenCV Library

Computer vision is by now a vital part of many robotics applications. In fact, it is so wide-spread that it has its own term: robotic vision. Robotic vision is the discipline which is involved with enabling robots to capture photographs or videos of their surroundings, process them in search of specific features and geometries and interpret them, in such a way that the robot will be able to gain information about its surroundings. This information can be used for navigation, obstacle avoidance, object recognition, grasping and manipulation of objects and implicit communication.

Generally, the development of computer vision algorithms is a scientific field on itself and it would be near impossible to incorporate any vision functionality if robotics applications if one had to program it from the beginning.

Thankfully, as is usually is the case, a revolutionary software project has emerged, solving exactly the problem of computer vision as a part of a robotics system: the *OpenCV* library [39].

OpenCV comes as a C++ or Python library and is very feature-rich. From colour-masking to edge and feature detection, it provides a large set of software tools, which enable roboticists incorporate robotic vision functionality into their projects.

ROS has a dedicated set of packages for OpenCV, which are frequently updated.

In this work, OpenCV functions are used extensively during the processing phase of the NIR images, to extract the NDVI of the crop under survey.

8.7 The MAVLink Protocol

When software systems with modular software are deployed, especially in distributed systems, there is the need for standardized communication protocols between the constituting parts. Not only does a communication protocol allow for robust communication between the system parts, contribute to a streamlined developing phase and successful transfer of required information, but it provides important indirect advantages to the system.

With a carefully designed, detailed and preemptive communication protocol, a system can be expanded with new modules and parts by using the same protocol. As long as all system parts "speak the same language", they can cooperate with each other.

Equally important, the communication protocol is the system part which ensures the integrity of communications and their security. Methods for identifying a broken

or otherwise corrupted message, such as checksums are specified by the communication protocol. Packetization and reconstruction lie under the same protocol. Also, encryption methods are topics of the communication protocol as well.

Traditionally, for as long as UAVs were considered solely military technology, aviation, military-grade communication protocols were used for UAV-GCS communication. These usually come with extensive security requirements, are cumbersome to program and integrate into a new project and are very difficult to extend, in order to incorporate new functionality.

As UAVs have become more accessible by the research community, other communication protocols are emerging and are commonly used in new projects and products. One of them, perhaps the most known, is *MAVLink* [45].

Taken from the webpage of QGroundControl, a GCS software developed by ETH Zyrich, follows the definition of another piece of software developed by the same laboratory, MAVLink:

```
MAVLink is a very lightweight, header-only message marshaling
library for micro air vehicles
```

At its core, MAVLink is nothing more than a C or Python library, which can be imported into any UAV or GCS communications module. It defines standard message types for common UAV-related operations, such as the waypoint or mission definition, upload and download, telemetry information, command relaying and health monitoring.

It also provides standard methods for encoding and decoding the aforementioned information with software functions and offers a packetization strategy to take advantage of communication systems of specific packet size, increasing bandwidth.

MAVLink leaves room for user-defined messages and is constantly evolving, through user feedback. Currently, the stable version is 1.0 but 2.0 is on the works.

Each MAVLink packet has a specific structure, but not fixed length. Its components are visualized in Figure 8.4, where each byte is represented by a square.

1. Start byte (0xFF)
2. Payload length, used for packet decoding, since its length is not fixed
3. Packet sequence ID, to detect packet loss
4. System ID, for packet addressing in multi-system networks
5. Component ID, for packet addressing in multi-component systems
6. Message ID, corresponding to a predefined message set and characterizing the meaning of the payload

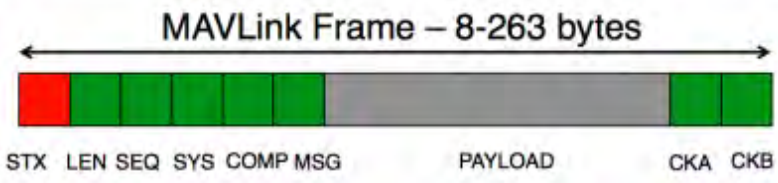

Figure 8.4: The components of a MAVLink packet

7. Data bytes

8. Two checksum bytes for packet integrity verification

ArduPilot and PX4 both use MAVLink as their communication protocol for autopilot-GCS communication, but in theory MAVLink also supports multi-vehicle networks and hardware sub-module addressing. Other autopilot projects use MAVLink as their communication protocol, exactly because it is robust, functional and stand-alone.

8.8 The mavros Package

Even though MAVLink C and Python libraries are provided, there is no officially supported method to interpret a MAVLink packet into a ROS topic. This functionality is highly desirable, when one wants to interface an ArduPilot autopilot with a ROS system.

In order to achieve that goal, a dedicated node would have to be programmed, which would take advantage of the MAVLink libraries and translate the message traffic incoming from the UAV into ROS topic of different types and vice versa. Thankfully, this program already exists and is incorporated in the *mavros* package [25].

The mavros node can be run alongside other ROS nodes, intercepting MAVLink traffic between the autopilot and GCS. Information such as GPS coordinates, orientation, altitude and many more are exposed as ROS topics in real time. Conversely, mission commands, flight mode issuing and control surface commands can be sent as topics to mavros and be encoded as MAVLink packets, sent to the UAV.

Unfortunately, mavros is designed with the PX4 firmware in mind, which uses a slightly more specialized subset of the MAVLink dialect. As a result, not all functionality advertised and documented by mavros is operational when communicating with ArduPilot systems, thus the developer should be aware of that.

In this work, mavros is used to capture telemetry information of the UAV during the scanning of the crops phase. The position and orientation of the UAV are recorded. Specifically these topics are used:

- `/mavros/global\_position/global` containing the timestamped GPS coordinates of the UAV

- `/tf` where mavros publishes the transformation from the body frame to the NED frame

8.9 The dronekit Library

Another piece of software that can be used to intercept, decode and encode MAVLink packets is *dronekit* [4]. It comes with APIs for Android, Python and cloud applications, but does not support ROS topics. Still, it can be used in a ROS node as any other Python library.

dronekit is much more feature-rich compared to mavros, when it comes to issuing commands to the autopilot and triggering based on events. It also encodes information in a more user-friendly way, since it abstracts a lot of functionality under Python classes and methods.

A vehicle object is the core of dronekit functionality, which is instantiated with the source address of the MAVLink stream.
`vehicle = dronekit.connect(<mavlink_src_address)`
The MAVlink source address can be a serial COM port, a USB port, a TCP or a UDP IP address.

The autopilot mission can be downloaded as a mission object, edited and reuploaded.
```
mission = vehicle.commands
mission.clear()
mission.add(waypoint)
mission.upload()
```

The location of the vehicle can be queried at any time through the location member, in various frames of reference.
```
vehicle.location.global_frame
vehicle.location.local_frame
```

The same holds for the vehicle orientation
`vehicle.heading`

Finally, the mode of the autopilot can be set by
`vehicle.mode = dronekit.VehicleMode(<modeName>)`
Common modes offer manual, guided and automatic control.

In this work, dronekit is ran in a ROS node on the Odroid carried by the rover, in order to monitor the state of the rover and issue waypoint commands for obstacle avoidance, according to the readings of the laser scanner.

8.10 The MAVProxy GCS

Throughout this work, there is often the need to re-direct and send MAVLink packet streams to one or more recipients. The telemetry information from the rover must reach both the Odroid on-board companion computer, but the GCS laptop as well, for the system operator to monitor it.

Moreover, the transport layer over which they are sent may need to be switched, according to the channel they are sent through. Packets travel over an 118kbaud/s, 8N1, TTL serial connection while going from the Pixhawk to the Odroid, but as a high-speed Ethernet UDP stream from the Odroid to the GCS laptop.

The stream duplication and type conversion functionalities are not trivial at all; a bad implementation of such functionality may lead to high packet loss, latency and low bandwidth. Thankfully, there exists a software which can handle this type of operations very efficiently and effectively: The *MAVProxy* GCS software [10].

MAVProxy was originally built as a GCS software, with all the traditional functionality of telemetry display, mission upload and monitor and teleoperation, by one of the main contributors and authors of ArduPilot, Andrew Tridgell. It was built to have a very small resource impact on the running OS, and for that reason it is a good choice for small, low-power, embedded computers. It uses a module-based logic, in order to keep only the vital functionality active at any moment. In fact, at its core, MAVProxy can run only as a console-based program, with no GUI at all.

Because it incorporates functionality to redirect and convert MAVLink streams, in a level that almost no other GCS software can offer right now, it is often used solely as a MAVLink handling software. This is exactly how it is used in this work as well.

Specifically, an instance of MAVProxy runs on the Odroid of the rover to re-direct the Pixhawk stream to the GCS laptop and convert it from TTL serial to UDP protocol. Another instance runs on the GCS laptop to duplicate the UAV Pixhawk telemetry stream and provide it to both the GCS software and the ROS nodes who need it.

For example, the initialization command run on the Odroid, which raises MAVProxy and routes the MAVLink streams is

```
mavproxy.py --master=/tty/USB0 --baudrate=115200
     --out=udp:127.0.0.1:14550
     --out=udp:127.0.0.1:14551
     --out=udp:192.168.1.201:14551
```

The `master` argument lets MAVProxy know at which port the source of the MAVLink stream sends the telemetry stream. Multiple ports can be declared in the case where multiple, redundant links are used. In our case, a USB port is used.

The `baudrate` argument accompanies all serial port declarations and sets the baud rate of the communication.

With the `out` argument, the ports to which the MAVLink stream should be forwarded are declared. As is the case, the `udp:IP:port` format is valid as well.

8.11 The wget Program

As it was previously mentioned, the photographs taken from the Multispek camera are stored within its SD card and are available through the file server hosted by the camera at its IP address.

These images need to be accessed through the established Local Area Network (LAN); removing the SD card after each flight is not only cumbersome and prone to accidents, but most importantly breaks the continuum of operations which ideally should happen in real time.

Naturally, the act of downloading the images should not be assigned to a human through the GUI of a web browser. This is an unnecessary workload with a repetitive character, which needs to be carried out during the whole flight of the UAV. All of these characteristics make the automation of this task a very lucrative option.

The *wget* program for Linux [53] is a very suitable candidate: It is part of the GNU project and advertises exactly the required functionality: retrieving files using HTTP, HTTPS or FTP. It can be passed a number of option flags, which control and embellish its operation.

In our case, we will be using the following command to retrieve pictures from Multispek:

```
wget -r -nd -np -N -A .JPG http://10.5.5.9.8080/videos/DCIM/110GOPRO/
```

Let us examine each option flag separately, in an effort to understand the exact function of the command:

- -r: Enables the recursive parsing of the directory. Used in case subfolders with images are created under the directory of interest.
- -nd: When parsing and retrieving recursively, do not mirror the directory structure. Instead copy all files in a single directory.
- -np: Do not parse parent directories. Only traveling the directory tree downwards is allowed.
- -N: Turn on timestamping. This option makes wget download a file only if it is a newer version of the already copied one. In this way, an already downloaded

file will not be downloaded again, cluttering the bandwidth of the wireless link.

- -A .JPG: Create an accept list, which filters which file types are downloaded. In this case, only JPG files are downloaded.
- The target address: This is the last argument, which states which network address will the wget program access and copy from.

8.12 k-means Algorithm and Library

In the premises of this work, the many thousands, if not millions of data points, corresponding to each image pixel are captured during an imaging mission. Each is evaluated to obtain vegetation health data which will later be used to decide whether this specific point needs to be issued with intervention material (pesticides or water). However, it is evident that pixel-by-pixel application is infeasible.

A strategy for grouping the datapoints is sought, so that representative locations can be extracted. Application of intervention material in these locations should cover as many of the datapoints of interest as possible.

To that goal, the *k-means* algorithm, stemming from the signal processing and data mining scientific fields is employed. Given a set of datapoints and a number of required results, the algorithm returns a number of *centroids*, points which correspond to an equal number of groups (clusters), to which the datapoints can be categorized. Each datapoint is said to belong to the cluster of a centroid if that centroid is the one that is closest to the datapoint. A visualization of the clustering procedure can be seen in Figure 8.5.

The implementation of the k-means algorithm used in this work was taken from the SciPy Python library, specifically the *kmeans2* algorithm [19].

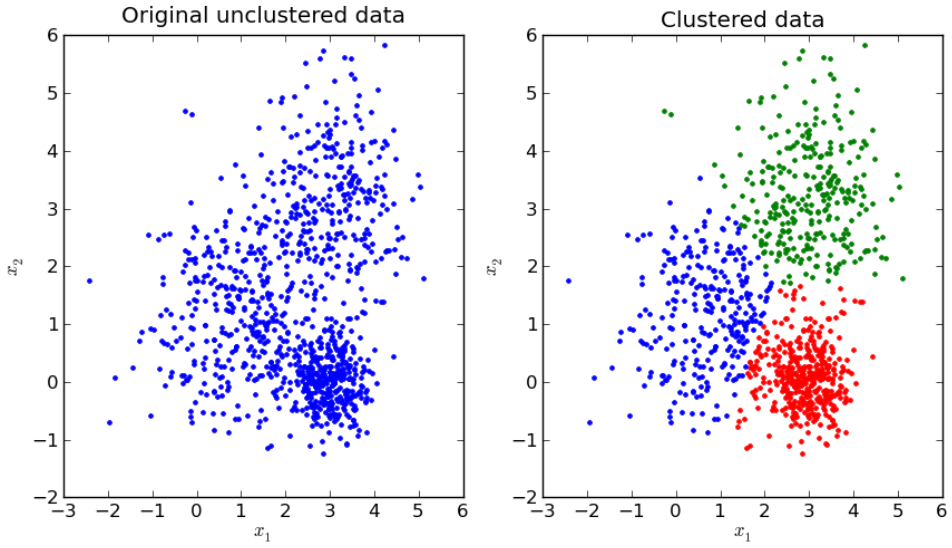

Figure 8.5: Clustering with the k-means algorithm

Chapter 9

System Integration

In this chapter, the details of how individual components of the system are combined to form a unified system will be laid out. Furthermore, the novel software solutions which were employed for the solution of the problem at hand will be presented in detail.

9.1 Network Configuration

The proposed system is composed by separate, spatially distributed systems, which still need to communicate with each other in real-time. Hence, there is a need for a data network which spans the entire system and allows the exchange of data between the system components.

There are several constraints, imposed by the technology of the employed devices, which need to be taken into account during the network design.

First and foremost, as expected, both the UAV and the rover need to travel long distances in outdoor environments. It is impossible to maintain a wired connection through a tether to the GCS. A wireless connection is the only option in both cases.

The Pixhawk situated in the UAV can communicate its telemetry information with its wireless serial module. The ground-end of this serial link is a transceiver with a USB interface, which needs to be plugged onto the GCS.

The Multispek camera can output its captured images in two ways. The first one is to use its mini HDMI port, which constitutes a wired solution and thus is deemed unusable. The second is to use the WiFi network of the camera and the corresponding file server. It should be emphasized that GoPro 4 (and hence Multispek as well) is able to only raise their own WiFi network, upon which it acts as host and not as a client. Thus, it is impossible to attach a Multispek onto an existing WiFi network.

The SICK LMS111 laser scanner offers either a serial or an Ethernet output. It is convenient to use the Ethernet interface, since it offers higher transfer rates and is

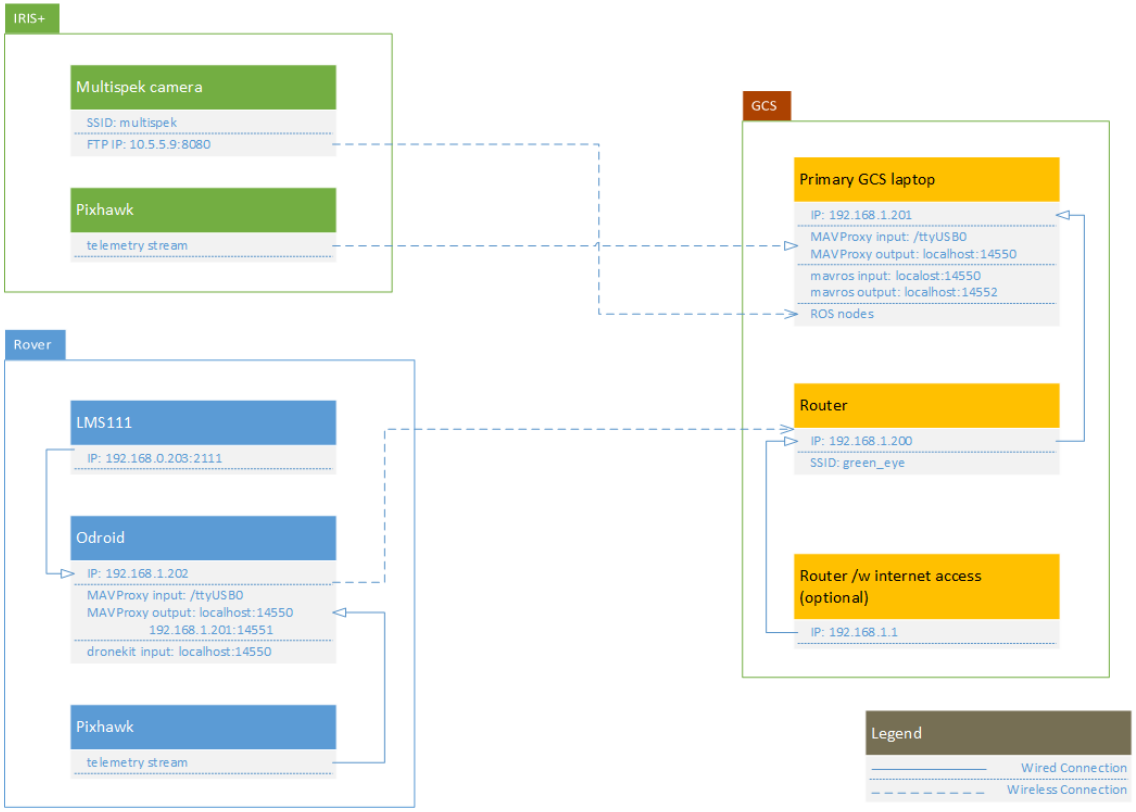

Figure 9.1: System network configuration

easily usable with modern computing systems.

The Pixhawk situated in the rover can communicate its telemetry information and receive commands either by contacting via a wireless serial module (as in the case of the UAV unit) or by directing the data stream towards the on-board Odroid unit.

The Odroid itself can communicate either through its USB ports, its ethernet port or by using a USB WiFi adapter.

While many GCS software can be used to setup the UAV mission, Mission Planner is a popular one and arguably the most mature and feature-rich. However, it is only available as a Windows distribution.

Taking into account all of the above constraints, the proposed system network is presented in Figure 9.1.

An Ethernet wireless router/switch is tasked with hosting the LAN/Wireless Local Area Network (WLAN), where the GCS laptop, the Mission Planner laptop and the rover Odroid are connected, allowing all of them to communicate with each other. Static IP addressing is selected, in order to automate the procedure of connection and discovering for each computer.

The reserved IPs of the network components are:

- router/switch: 192.168.1.201
- GCS laptop: 192.168.1.201
- Mission Planner laptop: 192.168.1.204
- Odroid: 192.168.1.202
- Laser scanner: 192.168.0.203

Starting from the UAV, the wireless serial transceiver pair is used to control and monitor the Pixhawk. The ground-based transceiver is plugged in a USB port on the GCS laptop. The over-the-air data transfer rate is set to 57600 baud, as per default.

The GCS laptop is connected to the LAN and a MAVProxy instance duplicates and forwards the MAVLink stream to multiple consumers in UDP protocol:

- The Mission Planner laptop at port 14550, where the mission of the UAV can be composed and uploaded and initiated
- The local mavros node listening at localhost:14550

The GCS laptop is also logged in the wireless network hosted by the Multispek, using its wireless adaptor, so that it can download the captured images. Also it is connected to the LAN via an Ethernet cable.

By default, the image server of the Multispek camera can be reached at the address 10.5.5.9:8080.

The Pixhawk placed in the rover is connected to the Odroid using a USB-to-Serial adaptor, with a data transfer rate of 115200 baud. On the Odroid, a MAVProxy instance forwards the MAVLink stream to multiple consumers in UDP protocol:

- localhost:14550 for local use
- The GCS laptop at port 14551

The laser scanner is connected to the Odroid using its Ethernet interface. A separate subnet is dedicated for this connection, since it was deemed impractical to bridge the wired and wireless network adapters of the Odroid. The subnet 192.168.0.X was used for this connection and the port 2111 or 2112 for communication, as per defaults.

9.2 Flow Diagram

Figure 9.2 contains an abstract representation of the overall information flow and the procedure flowchart. The steps in green refer primarily to the UAV system, those in yellow to the GCS and those in blue to the rover. In general, the software in the three different platforms can run independently and does not need synchronization.

At a glance and following the information flow, the UAV is the first system component to start its operation, by beginning its mission (top left start node). During its flight, it will capture images at predetermined points, defined during the mission setup (node below). Images will be triggered for as long as the mission goes on. After it is completed, the UAV can return home and land, regardless of the evolution of the other software threads.

The image processing software, running on the GCS laptop can be started before the UAV takes off (e.g. during system setup), seen as the yellow start node. Each time a new image is taken, the *wget* program, which polls the camera for new files, will download it on the laptop (node below). A software hook which monitors the image folder will trigger on each new image download and begin another iteration on the loop of the yellow branch.

At first, the new image filename will be published by a ROS node on a topic ("Publish new image filename"). The filename message will trigger the callback of the next ROS node, which will read the image, convert it into a ROS image message and push it down another topic ("Read image and publish image message").

Another node will read the image message and process it, in order to extract the NDVI values out of it, as well as locate the image areas which correspond to sick vegetation and isolate them ("Calculate image NDVI").

The next node ("Point of interest geolocation") apart from the image message will also receive the UAV position information, via telemetry. Combining this information, a large array storing the coordinates of all affected vegetation points will be aggregated. Until the mission is over, this array will continue to grow.

Upon the end of the mission, the coordinate array will be processed, and a few points of primary interest will be extracted. These points need to be visited by the rover and are thus communicated to it ("Publish waypoints to rover").

The rover software can also be started asynchronously, e.g. during system setup ("Rover mission start"). As long as no waypoints are received from the GCS, no operation will be carried out.

When the waypoints are received, each one will be examined in order and a path towards it which respects the crop field geometry will be generated ("Generate path to next waypoint"). Distance data produced by the laser scanner will be read and

the path will be adjusted in real time to ensure obstacle avoidance. When all of the waypoints are visited, the mission of the rover is complete.

The launch files (see Subsection 8.2) of the GCS laptop and the rover ROS systems reflect this architecture.

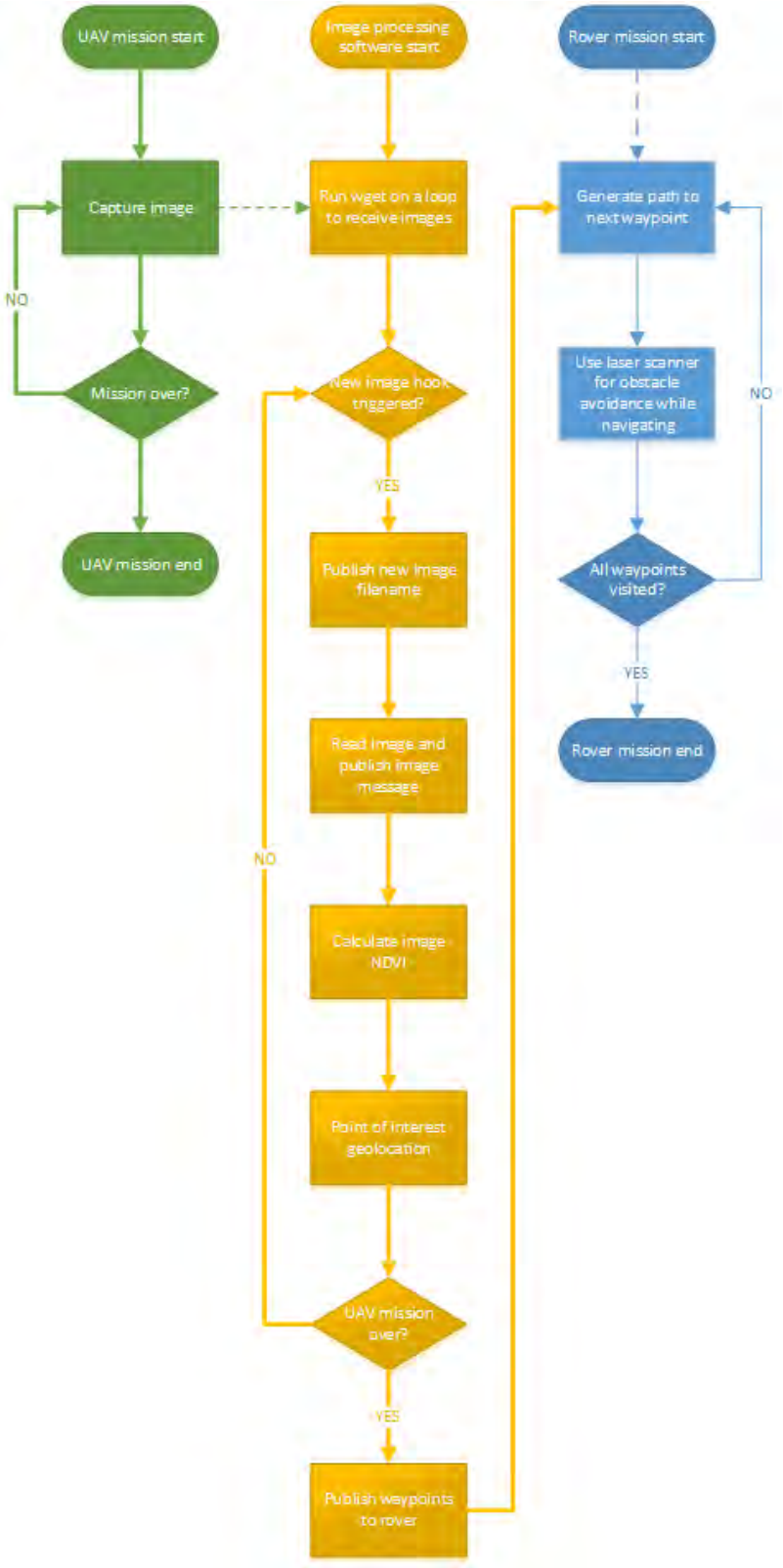

Figure 9.2: System software flow diagram

9.3 Launch Files

This is the launch file responsible for bringing up the ROS nodes on the GCS laptop.

```
1  <launch>
2
3    <!-- Launch the mavros nodes, which handle communication with the
         UAV -->
4    <include file="$(find␣green_eye)/launch/mavros_ge.launch"/>
5
6    <!-- Load the camera calibration -->
7    <group ns="calibration">
8      <rosparam command="load" file="/home/george/GOPRO/chessboard/
           calibrationdata/cal.yml"/>
9    </group>
10
11   <!-- Load program parameters -->
12   <rosparam command="load" file="$(find␣green_eye)/data/ge_params.
         yaml"/>
13
14   <!-- Launch the file monitoring node -->
15   <node pkg="green_eye" name="file_monitor" type="file_monitor"
         output="screen"/>
16
17   <!-- Launch the image publisher node, which converts the image file
          to a ROS topic -->
18   <node pkg="green_eye" name="image_publisher" type="image_publisher"
          output="screen"/>
19
20     <!-- Launch the image processing node -->
21   <node pkg="green_eye" name="image_processor" type="image_processor"
          output="screen">
22     <remap from="/camera/image_rect_color" to="/camera/image_raw"/>
23   </node>
24
25   <node pkg="green_eye" name="geolocation" type="geolocation_python"
         output="screen"/>
26
27 </launch>
```

Initially, `mavros` is raised, to convert telemetry from the UAV into ROS topics (line 4).

The pre-determined camera calibration parameters are loaded into the parameter

server from the stored YAML file (line 7-9).

The rest of the program parameters, such as the sick plants thresholds for the NDVI processing are loaded into the parameter server (line 12).

The `wget` program is launched from outside the ROS environment and is thus not included in this launch file.

The `file_monitor` responsible for detecting new file , `image_publisher` responsible for creating the image topic and `image_processor` responsible for NDVI processing are raised next (lines 15-23).

Finally, the Python `geolocation` node is brought up, to convert the points of interest from the captured pictures into waypoints for the rover to visit.

The rover launch file is also provided below

```
1 <launch>
2
3   <include file="$(find␣green_eye)/launch/lms111.launch" />
4
5   <rosparam command="load" file="$(find␣green_eye)/data/gh_params.
       yaml" />
6
7   <node pkg="green_eye" name="path_planner" type="gh_path_planner.py"
        output="screen" />
8
9 </launch>
```

Initially, it loads a custom-written launch file to initialize the laser scanner parameters and start the laser scanner driver node. This launch file is provided by the device driver ROS package.

Afterwards, it loads the parameters related to the path planner functionality.

Finally, it raises the main path planner node, where the target reception, dronekit interfacing, path generation and laser scanner interfacing functionality takes place.

9.4 File Monitoring

As presented in Section 8.11, the *wget* program is used in this work to download the NDVI images from the camera to the GCS laptop in real-time. New image files are created in a pre-specified directory. The GCS software needs to be alerted on the appearance of new images in order to read them later and convert them in a file format usable by OpenCV.

A dedicated ROS node was created for this task, namely watch for the creation of new images in a prespecified folder and send their file path down a ROS topic, on which another node tasked with image conversion is subscribed. The corresponding code is presented below and explained in detail.

The primary technology enabling this node is the `inotify.h` library, which creates hooks for file creation.

```
#include <ros/ros.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/inotify.h>
#include <signal.h>

#include <std_msgs/String.h>

#define EVENT_SIZE ( sizeof (struct inotify_event) )
#define BUF_LEN   ( 1024 * ( EVENT_SIZE + 16 ) )

int fd;
int wd;

////////////////////
// SIGIN handler //
////////////////////
void sigintHandler(int sig)
{
  // remove the hook and delete the event
  ( void ) inotify_rm_watch( fd, wd );
  ( void ) close( fd );
  ros::shutdown();
}

////////////////////
// Main Function //
////////////////////
int main( int argc, char **argv )
{
  ros::init(argc, argv, "file_monitor", ros::init_options::
     NoSigintHandler);
  ros::NodeHandle n;
  ros::Publisher pub = n.advertise<std_msgs::String>("/image_latest"
```

```
    ,100);
36  signal(SIGINT,sigintHandler);
37
38  ROS_INFO("file_monitor␣node␣up");
39
40  ros::WallDuration(3).sleep(); //wait for other nodes to get raised
41
42  int length, i;
43
44  char buffer[BUF_LEN];
45  std_msgs::String msg;
46  std::stringstream ss;
47
48  // Create the event
49  fd = inotify_init();
50
51  if ( fd < 0 ) {
52    perror( "inotify_init" );
53  }
54
55  // Add the hook to watch a folder for new images
56  wd = inotify_add_watch( fd, "/home/user/catkin_ws/src/green_eye/
       data/images", IN_CREATE);
57
58  // Loop and check for new files
59  while (ros::ok())
60  {
61
62    // Blocking-read for the new file
63    length = read( fd, buffer, BUF_LEN );
64
65    // Handle bad return values
66    if ( length < 0 ) {
67      ROS_ERROR("Bad␣length␣in␣file_monitor");
68      ros::shutdown();
69    }
70
71    i = 0;
72    while ( i < length ) {
73      struct inotify_event *event = ( struct inotify_event * ) &
           buffer[ i ];
74      if ( event->len ) {
```

```
75        if ( event->mask & IN_CREATE ) {
76          if ( event->mask & IN_ISDIR ) {
77            ROS_ERROR("Unexpected␣directory␣creation");
78          }
79          else {
80            ROS_INFO("The␣file␣%s␣was␣created", event->name);
81            // Convert from char to std_msgs::String
82            ss.str(std::string());
83            ss << event->name;
84            msg.data = ss.str();
85            // and publish
86            pub.publish(msg);
87          }
88        }
89        else if ( event->mask & IN_DELETE ) {
90          ROS_ERROR("Invalid␣event␣type");
91          ros::shutdown();
92        }
93      }
94      i += EVENT_SIZE + event->len;
95    }
96  }
97
98  ROS_INFO("file_monitor␣closing");
99
100 return 0;
101}
```

In lines 31-40, the node is initialized. It is given the name `file_monitor`, the node-handle is created and a topic publisher is assigned on it. The publisher produces the `/image_latest` topic of `std_msgs:String` type.

In lines 48-56 the hook is created and it is given a specific folder to watch. The last parameter of `inotify_add_watch` is `IN_CREATE`, which corresponds to file creation. Other options exist, such as file deletion or modification, but they are not of interest to us.

The while-loop in line 59 runs the following code perpetually, until roscore dies for any reason (exits due to an exception or closed by the user), which is monitored by the function `ros::ok()`.

The program waits in line 63 with a blocking read on the `fd` buffer, until a hook fills it with a struct related to the `inotify.h` library event. Once that happens,

the content is copied on the `buffer` buffer, for further processing. Specifically, this information is loaded in the `event` structure.

If the new file created is a folder (line 76), then an error is returned, since we do not expect any folders to be created inside our watch folder, only image files. Otherwise, we capture the filename and path provided in the `event->name` field and publish it over the `\image_latest` topic by using the `pub` publisher (lines 79-87).

With the new image file information pushed in the topic, another node will handle its reading and processing.

As a side note, notice that we have disabled the default SIGINT function of the node (line 33) and replaced it with a callback function (line 36). This is because by default the node wouldn't also delete the watched folder hook on shutdown and a specialized function is used to do that first instead, and then close the node (lines 20-26).

9.5 Publishing the Image Topic

After the new image filename and path have been published onto the `\image_latest` topic by the `file_monitor` node, the `image_publisher` node steps in to read that image and convert it into suitable message types for further processing. The corresponding code can be seen below and it is further explained afterwards.

```
1  #include <ros/ros.h>
2  #include <ros/console.h>
3  #include <image_transport/image_transport.h>
4  #include <opencv2/highgui/highgui.hpp>
5  #include <opencv2/contrib/contrib.hpp>
6  #include <cv_bridge/cv_bridge.h>
7
8  #include <cstdlib>
9  #include <cstdio>
10
11 #include <std_msgs/String.h>
12 #include <marti_visualization_msgs/TexturedMarker.h>
13 #include <geometry_msgs/Pose.h>
14 #include <tf/transform_listener.h>
15
16
17 class ImgPub {
18
19 public:
20
```

```
21  ros::Subscriber sub;
22  image_transport::Publisher pub_img_raw;
23  ros::Publisher pub_cfg;
24  sensor_msgs::CameraInfo cfg;
25
26  ros::Publisher pub_texmark;
27  marti_visualization_msgs::TexturedMarker texturedmarker;
28  geometry_msgs::Pose camera_pose;
29
30  tf::TransformListener listener;
31  tf::StampedTransform transform;
32
33  ////////////////////////////
34  // Initialize class members //
35  ////////////////////////////
36  ImgPub(ros::NodeHandle n) {
37    // Subscriber to the latest image name topic
38    sub = n.subscribe("/image_latest",100, &ImgPub::imageCallback,
        this);
39
40    image_transport::ImageTransport it(n); // create the image
        transport handle
41
42    pub_img_raw = it.advertise("/camera/image_raw",1); // create the
        raw image topic
43    pub_cfg = n.advertise<sensor_msgs::CameraInfo>("/camera/
        camera_info",1); // create the camera_info topic
44    pub_texmark = n.advertise<marti_visualization_msgs::
        TexturedMarker>("/camera/mapviz_image",1); // create the
        mapviz TexturedMarker topic
45
46    // Read the camera_info data from the parameter server and create
         the CameraInfo message
47    XmlRpc::XmlRpcValue list;
48    int i;
49
50    cfg.header.frame_id = "camera";
51    double temp;
52    ros::param::get("/calibration/image_height", temp);
53    cfg.height = (uint)temp;
54    ros::param::get("/calibration/image_width", temp);
55    cfg.width = (uint)temp;
```

```
56    ros::param::get("/calibration/distortion_model", cfg.
          distortion_model);
57    ros::param::get("/calibration/distortion_coefficients/data", list
          );
58    cfg.D.clear();
59    for (i=0;i<5;i++) {
60      cfg.D.push_back((double)list[i]);
61    }
62    ros::param::get("/calibration/camera_matrix/data", list);
63    for (i=0;i<9;i++) {
64      cfg.K[i] = list[i];
65    }
66    ros::param::get("/calibration/rectification_matrix/data", list);
67    for (i=0;i<9;i++) {
68      cfg.R[i] = list[i];
69    }
70    ros::param::get("/calibration/projection_matrix/data", list);
71    for (i=0;i<12;i++) {
72      cfg.P[i] = list[i];
73    }
74
75    // Fill the TexturedMarker structure
76    texturedmarker.header.frame_id = "map";
77    texturedmarker.action = texturedmarker.ADD; //add
78    texturedmarker.ns = "";
79    texturedmarker.id = 0;
80    texturedmarker.lifetime = ros::Duration(300);
81    texturedmarker.resolution = 0.02;
82    texturedmarker.alpha = 1.0;
83
84    // Publish the camera info message for Geolocation to record it
85    pub_cfg.publish(cfg);
86
87  }
88
89  ////////////////
90  // Destructor //
91  ////////////////
92  ~ImgPub()
93  {}
94
95
```

```
96  ////////////////////////////////
97  //Process new image callback //
98  ////////////////////////////////
99  void imageCallback(const std_msgs::String img_name)
100 {
101   ROS_DEBUG("Callback to publish image %s", img_name.data.c_str());
          // Debug output
102   std::stringstream ss;
103   ss.str(std::string());
104
105   // Creating the image pathname
106   ss << "/home/user/catkin_ws/src/green_eye/data/images/";
107   ss << img_name.data;
108
109   // Read the image file to an openCV matrix
110   ROS_INFO("Reading %s", ss.str().c_str());
111   cv::Mat image = cv::imread(ss.str(), CV_LOAD_IMAGE_COLOR);
112
113   ros::Time stamp = ros::Time::now(); // capture the current
         timestamp
114
115   // stamp and publish the image topic
116   ROS_DEBUG("Converting cv_img to ros_img");
117   sensor_msgs::ImagePtr msg = cv_bridge::CvImage(std_msgs::Header()
         , "bgr8", image).toImageMsg();
118   msg->header.stamp = stamp;
119   msg->header.frame_id = "camera";
120   pub_img_raw.publish(msg);
121
122   // stamp and publish the camera_info
123   ROS_DEBUG("Publishing camera_info");
124   cfg.header.stamp = stamp;
125   pub_cfg.publish(cfg);
126   ROS_DEBUG("Done publishing raw images");
127
128   // Calculate the camera pose
129   ROS_DEBUG("Calculating the camera pose");
130   try{
131     listener.lookupTransform("map", "camera", ros::Time(0),
           transform);
132     camera_pose.position.x = transform.getOrigin().getX();
133     camera_pose.position.y = transform.getOrigin().getY();
```

```
134       camera_pose.position.z = transform.getOrigin().getZ();
135       camera_pose.orientation.x = transform.getRotation().getX();
136       camera_pose.orientation.y = transform.getRotation().getY();
137       camera_pose.orientation.z = transform.getRotation().getZ();
138       camera_pose.orientation.w = transform.getRotation().getW();
139
140       // stamp and publish the TexturedMarker topic
141       ROS_DEBUG("Publishing␣mapviz␣TexturedMarker");
142       texturedmarker.header.stamp = stamp;
143       texturedmarker.image = *msg;
144       texturedmarker.pose = camera_pose;
145       pub_texmark.publish(texturedmarker);
146       texturedmarker.id = texturedmarker.id+1;
147       }
148     catch (tf::TransformException &ex) {
149       ROS_ERROR("%s",ex.what());
150     }
151
152   }
153
154 };
155
156 ///////////////
157 //Main function
158 ///////////////
159 int main(int argc, char **argv)
160 {
161   ros::init(argc, argv, "image_publisher"); // initialize the ros
        node
162   ros::NodeHandle nh; // create the nodehandle
163
164   // Raising verbosity level to DEBUG
165   if( ros::console::set_logger_level(ROSCONSOLE_DEFAULT_NAME, ros::
        console::levels::Debug) ) {
166      ros::console::notifyLoggerLevelsChanged();
167   }
168
169   ImgPub image_publisher(nh);
170
171   //wait for other nodes to get raised
172   ros::Duration(1).sleep();
173
```

```
174   ROS_INFO("image_publisher␣node␣up");
175
176   // Spin waiting for the callback
177   while (ros::ok())
178   {
179     ros::spin();
180   }
181
182   return 0;
183 }
```

The node is initialized in lines 159-167 and a new object which handles all of the image reading functionality is created (line 169). This is an object of the `ImgPub` class, defined in the same file. Then the node falls into a spin (line 179) to capture subsequent incoming topic traffic.

The `ImgPub` class encompasses all the functionality of this node. This is a common practice in ROS node implementations: Start a node which creates only one class object and let that object handle all of the functionality for that node.

On initialization of the object, a new subscriber on the `/image_latest` topic is created (line 38) as well as publishers for the outputs of this node: the `/camera/image_raw` and `/camera/camera_info` topics. On the former, the raw image message will be published, wereas on the latter, important information regarding the camera calibration will be pushed, accompanying each image.

Notice that the publisher of the camera message is a method of the `image_transport` (*it*) class, instead of a typical nodehandle. This is done for performance and compatibility reasons, as explained in Section 8.4.

In lines 46-73 the structure regarding the camera calibration is initialized and populated with contents already uploaded in the parameter server beforehand the execution of the current node.

Next, the code for the subscriber callback function is defined (lines 99-152). Each time it is called, the new image filename is read (line 107) and the complete image path is constructed (lines 103-107). The image is read and stored in the OpenCV `cv::Mat` structure (lines 110-111). The image type is specified as a 3-channel image with the argument `CV_LOAD_IMAGE_COLOR`. `cv::Mat` is a generic and useful variable type which allows for the storage of images of variable size and interfaces well with most functions and methods of OpenCV.

Afterwards, the image array is converted into an image message, timestamped and published (lines 116-120), along with its camera calibration information (lines 123-126).

9.6 Image Processing for NDVI Extraction

This node comes functionally after `image_publisher`, capturing the raw image data and performing the NDVI operation onto it. Its code is presented below and explained aftwards.

```
#include <ros/ros.h>
#include <ros/console.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/contrib/contrib.hpp>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>

#include <cstdlib>
#include <cstdio>

#include <std_msgs/String.h>

ros::Publisher pub;
ros::Subscriber sub;
image_transport::Publisher pub_poi;
image_transport::Subscriber sub_img_rect;

static const std::string OPENCV_WINDOW = "Vigor␣index";

////////////////////////////////
//Process new image callback //
////////////////////////////////
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
  ROS_DEBUG("Entering␣image␣processing␣callback");
  cv_bridge::CvImagePtr cv_ptr; // Create a pointer to an openCV
     image

  ROS_DEBUG("Copying␣image␣for␣local␣processing");
  cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::
     BGR8); // Convert the ROS image message to a openCV image
  ROS_DEBUG("Pasting␣image␣to␣a␣cv::Mat␣object");
  cv::Mat image = cv_ptr->image; // acquire the image pointer

  // split the image channels to apply NDVI calculation
  ROS_DEBUG("Splitting␣the␣image␣channels");
```

```
36  cv::Mat channels[3];
37  cv::split(image,channels);
38  cv::Mat bi, gi, ri;
39  bi = channels[2];
40  gi = channels[1];
41  ri = channels[0];
42
43  // Convert to float for float calculations
44  ROS_DEBUG("Converting␣integer␣channels␣to␣float");
45  cv::Mat bf(bi.rows, bi.cols, CV_32FC1);
46  cv::Mat gf(gi.rows, gi.cols, CV_32FC1);
47  cv::Mat rf(ri.rows, ri.cols, CV_32FC1);
48  bi.convertTo(bf,CV_32FC1);
49  gi.convertTo(gf,CV_32FC1);
50  ri.convertTo(rf,CV_32FC1);
51
52  // Create the NDVI arrays, as per e.g. the bf size
53  cv::Mat ndvi(bf.rows, bf.cols, CV_32FC1); // The NDVI image array,
       -1 - +1 range
54  cv::Mat ndvi_thresh(bf.rows, bf.cols, CV_32FC1); // The thresholded
        NDVI image array, {0,1} range
55  cv::Mat ndvi_disp(bf.rows, bf.cols, CV_32FC1); // The NDVI image
       array, displayable float range
56
57  // Perform the NDVI calculation
58  ROS_DEBUG("Performing␣NDVI␣calculaiton");
59  ndvi = (2*bf - rf) / (rf); // as suggested by Multispek crew
60
61  // Create a displayable NDVI image
62  //
63  ndvi_disp = ndvi;
64
65  // Perform thresholding operation
66  ROS_DEBUG("Thresholding␣the␣result");
67  // Define the sick vegetation thresholds
68  double thresh_low, thresh_high;
69  ros::param::getCached("/ndvi_thresh_min", thresh_low);
70  ros::param::getCached("/ndvi_thresh_max", thresh_high);
71  cv::Mat poi;
72  cv::inRange(ndvi, thresh_low, thresh_high, poi);
73
74  // Communicate the image higher and lower values for debugging
```

```
75   double min, max;
76   cv::minMaxLoc(ndvi, &min, &max);
77   std::cout << "ndvi␣min/max:␣" << min << "␣" << max << std::endl;
78   cv::minMaxLoc(poi, &min, &max);
79   std::cout << "poi␣min/max:␣" << min << "␣" << max << std::endl;
80
81   try{
82   cv::imshow(OPENCV_WINDOW, ri); // Display an image to the debug
        window
83   // cv::imshow(OPENCV_WINDOW, ndvi_disp); // Display an image to the
         debug window
84   // cv::imshow(OPENCV_WINDOW, poi); // Display an image to the debug
         window
85   }
86   catch(...) {ROS_DEBUG("imshow␣failed");
87
88   }
89   // stamp and publish the image topic
90   ROS_DEBUG("Converting␣cv_img␣to␣ros_img");
91   cv_bridge::CvImage img_out;
92   img_out.header = std_msgs::Header();
93   img_out.encoding = sensor_msgs::image_encodings::MONO8;
94   // img_out.image = ndvi;
95   img_out.image = poi;
96   sensor_msgs::ImagePtr img_out_ptr = img_out.toImageMsg();
97
98   img_out_ptr->header.stamp = msg->header.stamp;
99   img_out_ptr->header.frame_id = "camera";
100  pub_poi.publish(img_out_ptr);
101
102 }
103
104 ///////////////
105 //Main function
106 ///////////////
107 int main(int argc, char **argv)
108 {
109   ros::init(argc, argv, "image_processor"); // initialize the ros
        node
110   ros::NodeHandle n;
111
112   // Raising verbosity level to DEBUG
```

```
113   if( ros::console::set_logger_level(ROSCONSOLE_DEFAULT_NAME, ros::
          console::levels::Debug) ) {
114      ros::console::notifyLoggerLevelsChanged();
115   }
116
117   image_transport::ImageTransport it(n);
118   cv::namedWindow(OPENCV_WINDOW, CV_WINDOW_NORMAL);
119   cv::startWindowThread();
120
121   sub_img_rect = it.subscribe("/camera/image_rect_color", 10,
          imageCallback);
122   pub_poi = it.advertise("/camera/ndvi",1);
123
124   ROS_INFO("image_processor␣node␣up");
125
126   while (ros::ok())
127   {
128     ros::spin();
129   }
130
131   cv::destroyWindow(OPENCV_WINDOW);
132
133   return 0;
134 }
```

The node is initialized in lines 107-115. In lines 117 the image transport handle is created and in lines 118-119 an OpenCV image window thread is started. This can be used for visualization of the OpenCV arrays during the node runtime, for supervision or debugging purposes. In line 121 the subscription on the `/camera_rect_color` is initiated and in line 122 the resulting image is published on `/camera/ndvi`. Even though no previous node has been publishing the `/camera_rect_color` topic, this does not pose a problem. We use this chance to introduce the concept of topic name re-naming capabilities of ROS.

For the launch of the `image_publisher` node, we execute the following code:

```
<node pkg="green_eye" name="image_processor" type="image_processor"
output="screen"
<remap from="/camera/image_rect_color" to="/camera/image_raw"/>
</node>
```

This code re-maps every instance of the `/camera/image_rect_color` topic in the `image_publisher` code to the `/camera/image_raw` topic. This is another great feature ROS provides to aid code modularity and reusability.

The image processing takes place in the `imageCallback` function. In lines 27-32 the input image is converted from the `image_transport` type to the OpenCV `cv::Mat` type, using the `cv_bridge::toCvCopy` method.

Afterwards, the 3 channels (red, green and NIR) are split and allocated into separate variables (lines 35-41). These channels are of integer type but we are interested in performing float calculations on top of them, so they are converted into float type (lines 44-50). Other arrays of the same size and type are created to store intermediate and final results (lines 53-55).

The core NDVI calculation takes place in line 59. Notice that it differs from what was presented in Section 5.2. This is because of the NIR band spillage on the red channel. To compensate for this, the NIR channel is subtracted from the red channel, to obtain the "true red" channel. In turn, this is the channel used for NDVI calculations. While this might seem inappropriate, it is the technical solution proposed by the camera manufacturers, nonetheless.

Since we are interested in isolating the features corresponding to sick vegetation from the NDVI-extracted image, we proceed with thresholding the result (lines 66-72). The threshold value are not fixed, but read from the parameter server. The result is an image populated only in the value range of interest. This functions as an "image mask" which covers the vegetation of interest.

9.7 Point of Interest Geolocation

After the NDVI information has been extracted from the image, the next course of action is to isolate single points of interest and correlate them to the position of the UAV. This task is performed by the following node, whose code is presented and explained in detail below. Note that this node is written in Python, in contrast to the previous nodes which were written in C++. Python was selected for this node because it provided some useful high-level functions and methods.

```
1  #!/usr/bin/env python
2
3  import rospy
4  import numpy as np
5  import sys
6  import tf
7  from cv_bridge import CvBridge, CvBridgeError
8  from scipy.cluster.vq import kmeans2
9  import image_geometry
10 import utm
11
12 # Import required messages
```

```
from rosgraph_msgs.msg import Clock
from sensor_msgs.msg import Image
from sensor_msgs.msg import CameraInfo
from sensor_msgs.msg import NavSatFix
import geometry_msgs.msg
from visualization_msgs.msg import Marker, MarkerArray

class Geolocator:

    def __init__(self):
        self.sub_img = rospy.Subscriber('/camera/ndvi', Image, self.
            append_data, queue_size=50)
        self.sub_cfg = rospy.Subscriber('/camera/camera_info',
            CameraInfo, self.get_camera_info, queue_size=10)
        self.sub_gps = rospy.Subscriber('/mavros/global_position/
            global', NavSatFix, self.get_wgs84, queue_size=10)
        self.pub_marker = rospy.Publisher('/target/marker', Marker,
            queue_size=10000)
        self.pub_GPS = rospy.Publisher('/target/GPS', NavSatFix,
            queue_size=10)
        self.pub_pois = rospy.Publisher('/target/points',
            geometry_msgs.msg.PointStamped, queue_size=10000)

        self.bridge = CvBridge()
        self.cfg = CameraInfo()
        self.camera_model = image_geometry.PinholeCameraModel()
        self.N = None
        self.E = None
        self.alt = 0
        self.max_alt = 0
        self.listener = tf.TransformListener()
        self.done = False
        self.points = np.empty(shape = (0,2), dtype='float64') #
            Points of interest container
        self.cur_point = geometry_msgs.msg.PointStamped() # Temporary
             point variable
        self.cur_point.header.frame_id = 'camera'
        self.count = 0
        self.temp_array = np.empty(shape = (3000,4000) , dtype='uint8
            ')
        self.centroids = None
        self.home = NavSatFix()
```

```
46
47          # Build target markers array
48          self.target = Marker()
49          self.targets = MarkerArray()
50          self.target.header.frame_id = '/map_true'
51          self.target.id = 0
52          self.target.type = Marker.CYLINDER
53          self.target.action = Marker.ADD
54          self.target.pose.orientation.x = 0
55          self.target.pose.orientation.y = 0
56          self.target.pose.orientation.z = 0
57          self.target.pose.orientation.w = 1
58          self.target.scale.x = 1
59          self.target.scale.y = 1
60          self.target.scale.z = 1
61          self.target.color.r = 1
62          self.target.color.g = 0
63          self.target.color.b = 0
64          self.target.color.a = 1
65          self.target.lifetime = rospy.Duration(10)
66
67      def get_camera_info(self, cfg_msg):
68          rospy.logdebug('storing␣CameraInfo␣structure')
69          self.cfg = cfg_msg;
70          self.camera_model.fromCameraInfo(cfg_msg)
71
72      def get_wgs84(self, gps_msg):
73          self.latitude = gps_msg.latitude
74          self.longitude = gps_msg.longitude
75
76      def append_data(self, image):
77          rospy.logdebug('geolocation␣new␣image␣callback')
78          try:
79              (trans,rot) = self.listener.lookupTransform('map', '/
                    camera', rospy.Time(0))
80          except (tf.LookupException, tf.ConnectivityException, tf.
                ExtrapolationException):
81              pass
82          # parse image
83          try:
84            cv_image = self.bridge.imgmsg_to_cv2(image, "mono8")
85          except CvBridgeError as e:
```

```
86          print(e)
87
88
89      rospy.logdebug('Rectifying image')
90      self.camera_model.rectifyImage(cv_image, cv_image)
91      rospy.logdebug('Converting to numpy array')
92      temp_array = np.asarray(cv_image)
93      temp_array = np.squeeze(temp_array)
94      rospy.logdebug('Calculting non-zero elements')
95      temp_points = np.transpose(np.nonzero(temp_array))
96      rospy.logdebug('Current image has %d nonzero elements' %
           temp_points.shape[0] )
97      projected_points = np.empty([3000*4000,3], dtype='float64')
98      counter = 0
99      skipper = 0
100     for i,j in temp_points:
101         if (skipper % 100 == 0): # run operations every 100
               coordinate pairs
102             temp_point = projected_points[counter,:] # For each
                   point in the image frame
103             temp_point_tup = self.camera_model.projectPixelTo3dRay
                   ((i,j)) # Convert it to the camera frame
104             temp_point = np.empty([1,3], dtype='float64')
105             temp_point[0,0] = temp_point_tup[0]
106             temp_point[0,1] = temp_point_tup[1]
107             temp_point[0,2] = temp_point_tup[2]
108             temp_point = temp_point/temp_point[0,2] # Normalize it
                    for unit height
109             temp_point = -temp_point * self.alt # And scale it up
                   by the vehicle height
110
111             # Find the camera-to-map transformation
112             temp_point_stamped = geometry_msgs.msg.PointStamped()
113             temp_point_stamped.header.frame_id = '/camera'
114             temp_point_stamped.header.stamp = rospy.Time(0)
115             temp_point_stamped.point.x = temp_point[0,0]
116             temp_point_stamped.point.y = temp_point[0,1]
117             temp_point_stamped.point.z = temp_point[0,2]
118
119             temp_point_stamped_NED = self.listener.transformPoint(
                   '/map_true', temp_point_stamped)
120
```

```
121                 projected_points[counter,0] = temp_point_stamped_NED.
                        point.x
122                 projected_points[counter,1] = temp_point_stamped_NED.
                        point.y
123                 projected_points[counter,2] = temp_point_stamped_NED.
                        point.z
124
125                 self.pub_pois.publish(temp_point_stamped_NED)
126
127                 self.target.id = self.target.id + 1
128                 self.target.pose.position.x = temp_point_stamped_NED.
                        point.x
129                 self.target.pose.position.y = temp_point_stamped_NED.
                        point.y
130                 self.target.header.stamp = rospy.Time(0)
131
132                 counter += 1
133             skipper += 1
134
135         # Add the points of this image to the points container (it is
                in NED frame)
136         self.points = np.vstack((self.points, projected_points[0:
                counter,0:2]))
137         (rows, cols) = self.points.shape
138         rospy.logdebug('Points␣gathered:␣%d␣x␣%d', rows, cols)
139         # rospy.logdebug('First point coordinates in camera frame: (%
                f, %f, %f)' % (temp_point[0,0], temp_point[0,1],
                temp_point[0,2]))
140         rospy.logdebug('First␣NED␣point␣in␣current␣image:␣(%f,␣%f,␣%f
                )' % (projected_points[0,0], projected_points[0,1],
                projected_points[0,2]))
141
142         self.find_centroids()
143
144         time_now = rospy.Time.now()
145         self.target.header.stamp = time_now
146         self.target.header.seq = 1
147
148     def find_centroids(self):
149         rospy.logdebug('entered␣find_centroids')
150         (self.centroids, labels) = kmeans2(self.points, 3)
151         rospy.logdebug("""Centroid coordinates:
```

```
152                         %f, %f
153                         %f, %f
154                         %f, %f""" % (self.centroids[0,0], self.
                                centroids[0,1], self.centroids[1,0], self.
                                centroids[1,1], self.centroids[2,0], self.
                                centroids[2,1]))
155         (rows, cols) = self.centroids.shape
156         rospy.logdebug('Finished␣centroids␣calculation.␣Found␣%d␣x␣%d
                ␣centroids', rows, cols)
157 
158     def checkDone(self):
159         try:
160             rospy.logdebug('Checking␣current␣altitude')
161             (trans,rot) = self.listener.lookupTransform('map', '/
                    camera', rospy.Time(0))
162             self.N = trans[0]
163             self.E = trans[1]
164             self.alt = trans[2]
165             rospy.logdebug('Current␣altitude:␣%f,␣Maximum␣altitude:␣%f
                    ' % (self.alt, self.max_alt))
166             if self.alt > self.max_alt:
167                 self.max_alt = self.alt
168             # Check if mission is done
169             if self.alt < (0.5*self.max_alt):
170                 if not(self.done):
171                     self.done = True # This means we're coming down
172                     self.find_centroids()
173                     time_now = rospy.Time.now()
174                     self.targets.markers[0].header.stamp = time_now
175                     self.targets.markers[0].pose.position.x = self.
                            centroids[0,0]
176                     self.targets.markers[0].pose.position.y = self.
                            centroids[0,1]
177                     self.targets.markers[1].header.stamp = time_now
178                     self.targets.markers[1].pose.position.x = self.
                            centroids[1,0]
179                     self.targets.markers[1].pose.position.y = self.
                            centroids[1,1]
180                     self.targets.markers[2].header.stamp = time_now
181                     self.targets.markers[2].pose.position.x = self.
                            centroids[2,0]
182                     self.targets.markers[2].pose.position.y = self.
```

```
                    centroids[2,1]
183             self.pub_marker.publish(self.targets) # Fails with
                    AttributeError: 'list' object has no attribute 'x'
184
185             gps_point = NavSatFix()
186             gps_point.header.frame_id = '/wgs84'
187             gps_point.header.stamp = rospy.Time.now()
188             gps_point.status.status = 0
189             (utm_lat, utm_lon, num, let) = utm.from_latlon(self.
                    home.latitude, self.home.longitude)
190             (lat, lon) = utm.to_latlon(utm_lat+self.centroids
                    [0,0], utm_lon+self.centroids[0,1], 34, 'S')
191             gps_point.latitude = lat
192             gps_point.longitude = lon
193             self.pub_GPS.publish(gps_point)
194             gps_point.header.stamp = rospy.Time.now()
195             (utm_lat, utm_lon, num, let) = utm.from_latlon(self.
                    home.latitude, self.home.longitude)
196             (lat, lon) = utm.to_latlon(utm_lat+self.centroids
                    [1,0], utm_lon+self.centroids[1,1], 34, 'S')
197             gps_point.latitude = lat
198             gps_point.longitude = lon
199             self.pub_GPS.publish(gps_point)
200             gps_point.header.stamp = rospy.Time.now()
201             (utm_lat, utm_lon, num, let) = utm.from_latlon(self.
                    home.latitude, self.home.longitude)
202             (lat, lon) = utm.to_latlon(utm_lat+self.centroids
                    [2,0], utm_lon+self.centroids[2,1], 34, 'S')
203             gps_point.latitude = lat
204             gps_point.longitude = lon
205             self.pub_GPS.publish(gps_point)
206         except (tf.LookupException, tf.ConnectivityException, tf.
                ExtrapolationException):
207             rospy.logdebug('failed␣to␣lookup␣transform')
208             pass
209
210 # Main function
211 if __name__=='__main__':
212     try:
213         rospy.init_node('geolocation', log_level=rospy.DEBUG)
214         G = Geolocator()
215         rate = rospy.Rate(1)
```

```
216         rospy.loginfo('geolocation Python node up')
217         while not rospy.is_shutdown():
218             G.checkDone()
219             rate.sleep()
220     except rospy.ROSInterruptException:
221         pass
```

The main node initialization code resides at the end, in lines 211-221. A `Geolocator` class object is created in line 214 (defined previously in the file) and left working. Once a second (line 215) its method `checkDone` is run to decide if the UAV has landed and the results can be gathered (line 218).

The `Geolocator` object is initialized in lines 22-65. It subscribes to the `/camera/ndvi` and `/camera/camera_info` topics and also to the `/mavros/global_position/global` topic, produced by `mavros` (see Section 8.8) (lines 23-25) and advertizes the `/target/marker` and `/target/GPS` topics, carrying the target information in local and GPS frame format.

Various variable initializations ensue, with the most important one being that of the `points` array (line 39). This holds the coordinates of every field point under examination. Also important is the `PinholeCameraModel` object (line 32), which allows us to rectify the image and convert from image to camera coordinates and vice-versa.

The `targets` variable array with length 3 will carry the three most suffering spots of vegetation for the entire survey and will also be used for visualization of the result in an external viewer (lines 48-65).

The callback function `get_camera_info` (lines 67-70) for the topic `/camera`... `/camera_info` simply stores the topic message (the camera calibration information) in a local structure.

The callback function `get_wgs84` (lines 72-74) for the topic `/mavros/`... `global_position/global` stores locally the UAV coordinates at each time instance in real-time.

The callback function `append_data` for the topic `/camera/ndvi` is responsible for the important task of the geolocation of the points of interest for each incoming picture. Every time a new ndvi image is available, first and foremost, the transformation for the camera frame to the NED frame is requested from the TF tree (line 79).

ROS has a dedicated mechanism for aiding the developers with coordinate frame transformations problems, called the *TF Tree*, where TF stands for "transformation". ROS has special API commands which the developers can use to inform it about the transformations between various coordinate frames. If the transformations refer to successive frames, then ROS can organize the frames to a tree structure, on which all

frames are placed and a path exists between any two frames. Then, the developer can query the transformation between any frame pair. This greatly alleviates a significant computational burden from the developer, since he no longer needs to write tedious transformation operations each time a transformation is needed. Instead, he only needs to update the TF tree regularly with single transformation steps.

In our case, `mavros` already populates the TF tree with transformations from WGS84, to NED, to the body frame from the UAV telemetry, which we can then easily use.

The image message is also converted to an OpenCV structure (lines 83-86).

The image is rectified (line 90) and converted to a numpy array (line 91-93). The *numpy* library offers a lot of high-level mathematical operations, methods and functions and is well-suited for the needs of this node.

Afterwards, the indices (coordinates in the image frame) of all the non-zero elements of the image are stored in the `temp_points` array. Recall that these correspond to the affected vegetation areas that were left out after the image masking operation.

Image pixels will now be scanned to reveal the most affected point of this image. Because the image resolution is very high (4000*3000) pixels, scanning and processing every pixel would take a very long time and break the real-time nature of the node. Instead, a balance between speed and resolution was found at scanning one every 100 image pixels (line 101).

For each scanned pixel, an array holding its normalized coordinates in the camera frame was created and filled (lines 102-103). Then, the coordinates were scaled up by the UAV above-ground altitude (line 109).

The conversion of the point coordinates are transformed into the NED frame (here referred to as `map_true`). The `transformPoint` method of the ROS transform listener is used. A `PointStamped` object is created and loaded with the coordinates (lines 112-117) and the result is returned into the object `temp_point_stamped_NED`.

The NED coordinates for each point are stored in the `points` array, along with other pixels' coordinates from previous image callbacks (line 136). They will be examined as a total after the end of the inspection flight.

The `checkDone` function (lines 158-208), run once a second, compares the state of the UAV against a mission end condition. In this case, the condition is whether the current altitude is less than half of the maximum altitude recorded during the flight (line 169), signifying the landing phase of the mission.

When the condition is met, the `find_centroids` function is run (line 172) on the `points` array. This function, defined in lines 148-156 uses the `kmeans2` algorithm from the `scipy.cluster.vq` library [19] at its core. Given a n-dimensional distribution of points (2-dimensional in our case) it returns k points (3 in our case, but this can vary) which stand as the centroids of the distribution. In other words, the

points are lumped together into 3 separate groups and the "centers" of those groups are given.

`kmeans2` allows us to group together many small points with poor NDVI index and treat them simultaneously, by intervening at central points, with maximum proximity to the surrounding distribution, without having to visit every single point, a potentially very time- and resource-consuming task.

The 3 centroids are stored in the `targets.markers` structure (lines 174-182). However, they are represented in the NED frame, which is not convenient for communicating them to the robotic system and is dependent upon the origin location of the UAV. The absolute WGS84 system will be used instead.

To convert from NED to WGS84, we use the `utm` Python library [11]. This is convenient because from a local standpoint, UTM coordinates [83] form a cartesian grid, in contrast to the WGS84, which are in degrees and in spherical coordinates around the Earth. Thus, they enable us to add the vectors of the home position and the centroids.

Summarily, the procedure for each centroid is:

1. Convert home position from WGS84 to UTM (line 189)
2. Vector add the local centroid coordinates (line 190)
3. Convert the result back to WGS84 (line 190)

Finally, the coordinates of the 3 target points are transmitted via the `/target/GPS` topic to the rover, so that it can proceed with the intervention (line 205).

9.8 Generation of Rover Path

The functionality related to the path planning and obstacle avoidance for the rover is performed onboard the rover, at the Odroid embedded computer. The option of sending all telemetry information back to the GCS laptop and processing it there was considered but rejected for a few reasons:

- The autonomy of the rover would be significantly reduced, being constantly dependent on the communication quality with the GCS.
- The required bandwidth to transmit the laser scanner readings is very high, cluttering the communication link.
- Either another wireless link hardware pair would be required to transmit the telemetry information from the Pixhawk to the GCS or an onbard embedded

computer would need to be installed anyway to convert and transmit the information over WiFi.

Thus, another ROS installation was included on the onbard Odroid and the following Python script was included in it. Its details are explained subsequently. The reasoning behind the functionality of this script is explained in Section 7.2.

It should be noted that both the Odroid and the GCS laptop are connected onto the same LAN and thus the multimaster_fkie (see Section 8.3) ROS package was used for the sharing of topics between to separate ROS installations.

The primary task of this script is three-fold:

1. Receive waypoints from the GCS in the form of GPS coordinates, which the rover should visit.
2. Create intermediate points, consisting a path which allows the rover to navigate the crop rows, according to the reasoning presented in Section 7.2.
3. Use laser scanner readings to avoid collisions with vegetation.

The last point will be expanded upon on the next section.

```
1  #!/usr/bin/env python
2
3  import rospy
4  # Import DroneKit-Python
5  import dronekit
6  from pymavlink import mavutil
7  import numpy as np
8
9  # Import required messages
10 from rosgraph_msgs.msg import Clock
11 from sensor_msgs.msg import NavSatFix
12 import geometry_msgs.msg
13 from visualization_msgs.msg import Marker, MarkerArray
14
15 class path_planner:
16
17   def __init__(self):
18     self.sub_gps = rospy.Subscriber('/target/GPS', NavSatFix, self.
         add_waypoint, queue_size=10)
19     self.rover = dronekit.connect('udp:127.0.0.1:14550')
20     rospy.sleep(3)
```

```
    self.mission = self.rover.commands
    self.mission.clear()
    self.mission.upload()
    self.wp_index = None
    self.wp_queue = []

    self.turnArray = 3*[None]
    self.turnIndex = None

  def add_waypoint(self,gps_point):
    self.wp_queue.append(gps_point)
    print "Added a new waypoint to the queue"

  def waypoint_manager(self):
    if self.wp_index == None: # If we don't have any active waypoints
        yet
      if len(self.wp_queue)==0: # If we don't have any waypoints
        print "No waypoints received"
        return
      else: # If we have our first waypoint, route it
        self.wp_index = 0;
        print "Routing first waypoint"
        self.draw_path(self.wp_queue[0])
        return
    elif self.distance_to_current_waypoint() < 2: # If we have
       reached the current waypoint
      if len(self.wp_queue) > self.wp_index+1 : # If there is another
          waypoint available
        print "Routing next waypoint"
        self.wp_index = self.wp_index+1
        self.draw_path(self.wp_queue[self.wp_index])
        return
      else:
        print "Routed last waypoint"
    else: # If we have at least one waypoint but haven't reached it
       yet
      print "Rover on its way to next_waypoint"
      self.navigate() # Navigate towards next turn
      return

  def distance_to_current_waypoint(self):
    """
```

```
59      Gets distance in metres to the current waypoint.
60      It returns None for the first waypoint (Home location).
61      """
62      current_wp=self.wp_queue[self.wp_index]
63      targetWaypointLocation=dronekit.LocationGlobalRelative(current_wp
            .latitude,current_wp.longitude,current_wp.altitude)
64      distancetopoint = self.get_distance_metres(self.rover.location.
            global_frame, targetWaypointLocation)
65      print "Distance␣to␣next␣waypoint:␣%f" % distancetopoint
66      return distancetopoint
67
68    def get_distance_metres(self,aLocation1, aLocation2):
69      """
70      Returns the ground distance in metres between two ‘LocationGlobal
            ‘ or ‘LocationGlobalRelative‘ objects.
71
72      This method is an approximation, and will not be accurate over
            large distances and close to the
73      earth’s poles. It comes from the ArduPilot test code:
74      https://github.com/diydrones/ardupilot/blob/master/Tools/autotest
            /common.py
75      """
76      dlat = aLocation2.lat - aLocation1.lat
77      dlong = aLocation2.lon - aLocation1.lon
78      return np.sqrt((dlat*dlat) + (dlong*dlong)) * 1.113195e5
79
80    # Called on a new target
81    # Sets up a new mission which respects the row heading
82    def draw_path(self,gps_point):
83      # Enter hold mode until the new mission is created
84      self.rover.mode = dronekit.VehicleMode("HOLD")
85
86      # Clear the existing mission
87      print "Clearing␣mission"
88      self.mission.download()
89      self.mission.wait_ready()
90      home = self.rover.home_location
91      self.mission.clear()
92
93      ptA = dronekit.LocationGlobalRelative(rospy.get_param("/lat_a"),
            rospy.get_param("/lon_a"), 0)
94      ptB = dronekit.LocationGlobalRelative(rospy.get_param("/lat_b"),
```

```
        rospy.get_param("/lon_b"), 0)
95      ptC = dronekit.LocationGlobalRelative(rospy.get_param("/lat_c"),
        rospy.get_param("/lon_c"), 0)
96      ptG = dronekit.LocationGlobal(gps_point.latitude, gps_point.
        longitude, gps_point.altitude)
97
98      ptS = self.rover.location.global_frame
99
100     # Turn 1
101     l_ab = (ptB.lat-ptA.lat)/(ptB.lon-ptA.lon)
102     c_ab = ptA.lat - l_ab*ptA.lon
103     l_bc = (ptC.lat-ptB.lat)/(ptC.lon-ptB.lon)
104     c_bc = ptB.lat - l_bc*ptB.lon
105     c_ad = ptA.lat - l_bc*ptA.lon
106
107     # Check if vehicle is heading Northwards or Southwards
108     curr_heading = self.rover.heading
109     print curr_heading
110     if curr_heading>180:
111       curr_heading = curr_heading - 360
112
113     if np.fabs(curr_heading)<90: #vehicle faces Northwards
114       print "heading␣north"
115       l_s = l_ab
116       c_s = ptS.lat - l_s*ptS.lon
117
118       ptE_lon = (c_bc-c_s)/(l_s-l_bc)
119       ptE_lat = l_s*ptE_lon + c_s
120
121     # Turn 2
122       l_g = l_ab
123       c_g = ptG.lat - l_g*ptG.lon
124
125       ptF_lon = (c_bc-c_g)/(l_g-l_bc)
126       ptF_lat = l_g*ptF_lon + c_g
127
128     else:
129       print "heading␣south"
130       l_s = l_ab
131       c_s = ptS.lat - l_s*ptS.lon
132
133       ptE_lon = (c_ad-c_s)/(l_s-l_bc)
```

```
      ptE_lat = l_s*ptE_lon + c_s

    # Turn 2
      l_g = l_ab
      c_g = ptG.lat - l_g*ptG.lon

      ptF_lon = (c_ad-c_g)/(l_g-l_bc)
      ptF_lat = l_g*ptF_lon + c_g


    ptE = dronekit.LocationGlobalRelative(ptE_lat, ptE_lon, 0)
    ptF = dronekit.LocationGlobalRelative(ptF_lat, ptF_lon, 0)

    print "Uploading new mission"
    self.mission.add(dronekit.Command(0,0,0, mavutil.mavlink.
       MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.
       MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0,home.lat, home.lon, 0))
    self.mission.add(dronekit.Command(0,0,0, mavutil.mavlink.
       MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.
       MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0,ptE.lat, ptE.lon, 0))
    self.mission.add(dronekit.Command(0,0,0, mavutil.mavlink.
       MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.
       MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0,ptF.lat, ptF.lon, 0))
    self.mission.add(dronekit.Command(0,0,0, mavutil.mavlink.
       MAV_FRAME_GLOBAL_RELATIVE_ALT, mavutil.mavlink.
       MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0,ptG.lat, ptG.lon, 0))

    # Verify that mission has 1+3 waypoints
    items = self.mission.count
    print "Constructed a mission with %i items" % (items)
    if items<4:
      print "Failed to produce a mission with 4 items"
    self.mission.upload()

    self.mission.download()
    self.mission.wait_ready()
    items = self.mission.count
    print "The vehicle has a mission with %i items" % (items)
    if items<4:
      print "Failed to acquire a mission with 4 items. Re-uploading"
      self.draw_path(gps_point)

```

```
168     self.turnArray[0] = ptE
169     self.turnArray[1] = ptF
170     self.turnArray[2] = ptG
171     self.turnIndex = 0
172
173   def navigate(self):
174     targetPoint=self.turnArray[self.turnIndex]
175     distancetopoint = self.get_distance_metres(self.rover.location.
          global_frame, targetPoint)
176     if distancetopoint < 2:
177       self.turnIndex = self.turnIndex + 1
178     if self.turnIndex = 3:
179       self.turnIndex = 2
180     self.rover.simple_goto(self.get_tracking_point(targetPoint))
181
182   def get_tracking_point(self, targetPoint):
183     # [To be described separately]
184
185
186 # Main function
187 if __name__=='__main__':
188   try:
189     rospy.init_node('gh_path_planner', log_level=rospy.DEBUG)
190     P = path_planner()
191     r = rospy.Rate(0.5) # 0.5Hz
192     rospy.loginfo('gh_path_planner␣Python␣node␣up')
193     while not rospy.is_shutdown():
194       P.waypoint_manager()
195       r.sleep()
196   except rospy.ROSInterruptException:
197     pass
```

Starting from the end of the code and from the main function (lines 186-197) the node is initialized with the name `gh_path_planner` (line 189) and a `path_planner` object is created (line 190). The object class encompasses all of the functionality of the code and will be explained afterwards. With a rate of 0.5Hz (line 191, 195) the method `waypoint_manager` of the class is called to perform the functionality of the script (line 194).

The class `path_planner` functions primarily through the method `waypoint_manager`, which stores the waypoints and keeps track of the already visited ones. It uses the `draw_path` method to generate the intermediate paths around the crop rows and the `navigate` method to perform low-level control over the steering of the rover.

During the initialization of the class object (lines 17-28), a subscriber on the `/target/GPS` topic is declared which listens for new target waypoints from the GCS. The corresponding callback function `add_waypoint` populates a related array (lines 30-32).

In line 19, the first usage of the `dronekit` module is met (see Section 8.9): a connection object is created which listens to the source address of the MAVLink stream (see Section 8.7), in this case specified at `udp:127.0.0.1:14550`. All read and write operations regarding the rover will take place through this `rover` object.

Another important data structure is the `mission` object (line 21) which is used to read and write waypoints to the Pixhawk autopilot. Even though in this application the rover is steered by the script and not from the Pixhawk guidance algorithm, it is useful as it can store the rover intended path and then be displayed at a GCS.

The `wp_queue` (line 25) list contains the target GPS coordinates and is populated by the callback method `add_waypoint`. The active, targeted waypoint is denoted at any time by the index variable `wp_index` (line 24).

Each time the `waypoint_manager` method is called, it initially checks against the `wp_queue` variable value (line 35).

If it is `None`, then no waypoint has been set as active yet; the mission is just starting. If `wp_queue` contains any target waypoints, then the first one is set as active and a new route is constructed to visit it (lines 36-43).

In the case where `wp_index` is not `None`, it means that some waypoint is currently active. First, it is checked if the rover has reached it, by comparing its distance from it (line 44), using the method `distance_to_current_waypoint`.

`distance_to_current_waypoint` compares the current position of the rover (acquired in line 65 with `rover` member `location.global_frame`) with the target waypoint, by using the method `get_distance_meters`. For the needs of the comparison, the ROS message type `NavSatFix` is converted to the `dronekit`-compatible `LocationGlobalRelative` type (lines 62-63). Note that the comparison of the coordinates is done using a norm-2 distance, suitable for a Cartesian plane. Even though the coordinates are ellipsoid, in the local tangeant NED frame the approximation is acceptable and the error is very small.

Continuing with the logic of `waypoint_manager`, if the distance to the current target waypoint is adequately small, the next waypoint is routed and `wp_index` is increased (lines 44-49). Otherwise, the current target waypoint remains unchanged and the steering algorithm is called, in the form of the method `navigate` (lines 52-55).

The draw_path Method

This method, which creates the crop row go-around path from the rover current location to the current target waypoint uses the reasoning and geometric calculations from Section 7.2.

Initially, the current mission loaded on the rover is downloaded and cleared (lines 88-89,91). At the end of this method, the new path will be uploaded to the Pixhawk. This is done solely for visualization purposes in our case, but there is always the option to let Pixhawk take over and guide the rover through the mission.

The `home` location refers to the point where the Pixhawk of the rover was first switched on. This serves as the origin of the local NED frame and is used in the conversions between UTM and WGS84 coordinates.

Points A, B and C are pre-specified from the system operator and loaded as parameters. They are retrieved from the parameter server in lines 93-95. The target location (point G) is constructed in line 96. The current location of the rover (point S) is read at line 98.

In order to construct point F, the equations of lines AB, BC and AD are first constructed, in lines 101-105. Afterwards, based on the current heading of the rover (line 108-111) point E is either placed on line BC (lines 113-119) or on line AD (lines 129-134).

Point F is similarly placed on either line in the cross-section with line crossing G, based on the same check, on lines 122-126 for the first case and on lines 137-141 for the second.

Finally, the two constructed points are converted both in `dronekit` format (lines 144-145) and also on a local array, `turnArray`, to be used by `navigate` (lines 168-171).

On lines 147-151 new mission object is created and on lines 154-166 it is uploaded to Pixhawk. The additional consistency checks are used for the case where the upload procedure is interrupted by intermittent communications. If an invalid number of waypoints is detected, then the mission is re-uploaded.

The navigate method

It is during the transitions from points S to E to F to G that we require a navigation scheme aided by the laser scanner measurements. The corresponding navigation routine, `navigate` is called upon in line 54. Its implementation is in lines 173-180. Essentially, the method works similarly to the waypoint visiting logic, holding a point index `turnIndex` for the currently active goal-point.

The coordinates of the current target point are passed onto the method `get_tracking_point` which returns the current tracking point. The tracking point

serves as a direction indicator, a point which is constantly updated, leading the rover and pulling it away from obstacles and towards the next point, as presented in Section 7.3. It is this tracking point that the Pixhawk is commanded to drive the rover to, at each time. The `dronekit simple_goto` method implements the low-level rover movement.

The logic for the generation of the tracking point, as well as the handling of the laser scanner data, as implemented by `get_tracking_point`, will be discussed in the next section.

9.9 Interfacing with the Laser Scanner and Obstacle Avoidance

Finally, the source code for the interfacing with the laser scanner and the obstacle avoidance functionality is presented below. This is an extension to the main Python script that was presented in the previous Section.

Initially, some additional declarations and variable initialization take place in the `__init__` function of the `path_planner` class.

```
1  def __init__(self):
2
3    #[...]
4
5    self.sub_laser = rospy.Subscriber('/laser/scan', LaserScan, self.
        update_scan, queue_size=10)
6
7    # [...]
8
9    self.kp = rospy.get_param("/laser/kp")
10   self.ko = rospy.get_param("/laser/ko")
11   self.dphi = rospy.get_param("/laser/dphi") # in radians
12   self.span = rospy.get_param("/laser/span") # in radians
13   self.ranges = [None]
```

A new subscriber is built for the topic where the laser scanner publishes its measurement messages (line 5). The variable parameters which contribute to the algorithm of obstacle avoidance are initialized and take values from the parameter server (lines 9-11). The distance measurement array $d(\phi)$ is initialized as well, in line 13 and will contain the latest distance measurement data.

The subscriber callback function, `update_scan` simply copies the `laserScan` message data to the local variable, as seen below.

```
  # Update the laser scanner ranges array
  def update_scan(self,laserScan):
    self.ranges = laserScan.ranges
```

Next, the `get_tracking_point` function code itself is presented.

```
  def get_tracking_point(self, targetPoint):
    if self.ranges[0] = None: # No laserScan message received yet
      return self.rover.location.global_frame # Send the current
         location to keep rover stationary
    else:
      curr_loc = self.rover.location.global_frame # Get current
         location and convert it to UTM
      (point_utm_lat, point_utm_lon, num, let) = utm.from_latlon(
         targetPoint.latitude, targetPoint.longitude)
      (rover_utm_lat, rover_utm_lon, num, let) = utm.from_latlon(
         curr_loc.lat, curr_loc.lon)

      # Construct the attractor vector
      point_distance = np.sqrt((point_utm_lat - rover_utm_lat)*(
         point_utm_lat - rover_utm_lat) + (point_utm_lon -
         rover_utm_lon)*(point_utm_lon - rover_utm_lon))
      attraction_NED_n = self.kp*(point_utm_lat - rover_utm_lat)/
         point_distance
      attraction_NED_e = self.kp*(point_utm_lon - rover_utm_lon)/
         point_distance

      # Construct the repulsor vector in the body frame
      nSamples = len(targetPoint.ranges)
      repulsion_NED_x = 0
      repulsion_NED_y = 0
      for i in xrange(nSamples):
        phi = i*self.dphi - self.span/2
        d = targetPoint.ranges[i]
        repulsion_body_x = repulsion_NED_x - self.ko*np.cos(phi)/(d*d
           )
        repulsion_body_y = repulsion_NED_y - self.ko*np.sin(phi)/(d*d
           )

      # Construct the repulsor vector in the NED frame
      heading = self.rover.heading * np.pi / 180
      repulsor_NED_n = np.cos(heading)*repulsion_body_x - np.sin(
```

```
            heading)*repulsion_body_y
27      repulsor_NED_e = np.sin(heading)*repulsion_body_x + np.cos(
            heading)*repulsion_body_y
28
29      # Construct tracking point in the NED frame
30      delta_NED_n = attraction_NED_n + repulsor_NED_n
31      delta_NED_e = attraction_NED_e + repulsor_NED_e
32
33      delta_NED_n_normalized = delta_NED_n / np.sqrt(delta_NED_n*
            delta_NED_n + delta_NED_e*delta_NED_e)
34      delta_NED_e_normalized = delta_NED_e / np.sqrt(delta_NED_n*
            delta_NED_n + delta_NED_e*delta_NED_e)
35
36      # Convert tracking point in UTM
37      tracking_utm_lat = rover_utm_lat + delta_NED_n_normalized
38      tracking_utm_lon = rover_utm_lon + delta_NED_e_normalized
39
40      # Convert tracking point to WGS84
41      (tracking_wgs_lat, tracking_wgs_lon) = utm.to_latlon(
            tracking_utm_lat, tracking_utm_lon, num, let)
42
43      return dronekit.LocationGlobalRelative(tracking_wgs_lat,
            tracking_wgs_lon, 0)
```

Initially, a validity check on the range data takes place in lines 2-3. If, for any reason the `ranges` array has not been filled with data yet, then the position which the rover currently holds will be returned, so that the autopilot will not perform any motion.

During normal operation, the algorithm presented in Section 7.3 is implemented, starting from capturing the current rover coordinates and converting them to a Cartesian frame, the UTM in this case (lines 5-7).

Afterwards, the attractor vector is constructed. The target point is passed to the function as an argument. After this is also converted in the UTM frame, the vector components from the rover in the x- and y- axis are calculated, such that the vector will have unit length.

The construction of the repulsor vector require the processing of the laser scanner data. Given that the corresponding angle of each measurement is known from the sensor configuration, each measurement sample is used to construct a repulsor vector component towards the opposite direction, in body frame coordinates. All these components are summed up to form the overall repulsor vector x- and y- coordinates in the `repulsor_body_x` and `repulsor_body_y` variables (lines 15-22).

Naturally, in order to add the repulsor vector to the attractor vector, a conversion from the body frame to the UTM frame is required, which takes place in lines 25-27 and requires the current rover heading.

The overall desired direction vector is calculated in lines 30-31 and normalized in lines 33-34, so that is serves as a vector field element, not a potential field element.

Its commanded displacement is added to the current rover location in lines 37-38, converted to WGS84 coordinates (line 41) and returned to the calling function (line 43).

Chapter 10

Results

In this chapter, all the above methods and technologies, as presented in this work, are used in practice. A typical mission is presented, the deployment of the system is described and its performance is discussed upon.

10.1 Typical Mission Description

The objective of this example is to use a UAV, equipped with a multispectral camera and fly an inspection mission over a field. Vegetation data is to be captured, processed and filtered, in order to locate the points on the field which need intervention the most. Based on those points, a mission will be created for the rover and uploaded onto it. The rover will visit all the provided waypoints, while avoiding obstacles on its path.

The field chosen for the presentation was a small one with young grapevines, located in eastern Attika, Greece. The mission was run during the month of April. An aerial photograph of the location under survey can be seen in Figure 10.1.

Figure 10.1: The vineyard under survey

10.2 UAV Mission Setup

As discussed in sections 4.5 and 5.3, the UAV employed was the 3DR IRIS+ quadrotor, equipped with a Multispek camera.

The mission was defined in Mission Planner and uploaded through it to the Pixhawk inside the IRIS+ through telemetry link. The survey grid was designed to be adequately tight, to ensure good coverage of the field. The mission waypoint list is presented below.

1. Set home position at (latitude:37.804459, longitude: 24.035542, altitude: 108.18)
2. Set desired airspeed at $3\,\mathrm{m\,s^{-1}}$
3. Set camera triggering every $5\,\mathrm{m}$
4. Go to point at (latitude: 37.804352, longitude: 24.035362, vertical offset: 40)
5. Go to point at (latitude: 37.804981, longitude: 24.035387, vertical offset: 40)
6. Go to point at (latitude: 37.804977, longitude: 24.035484, vertical offset: 40)

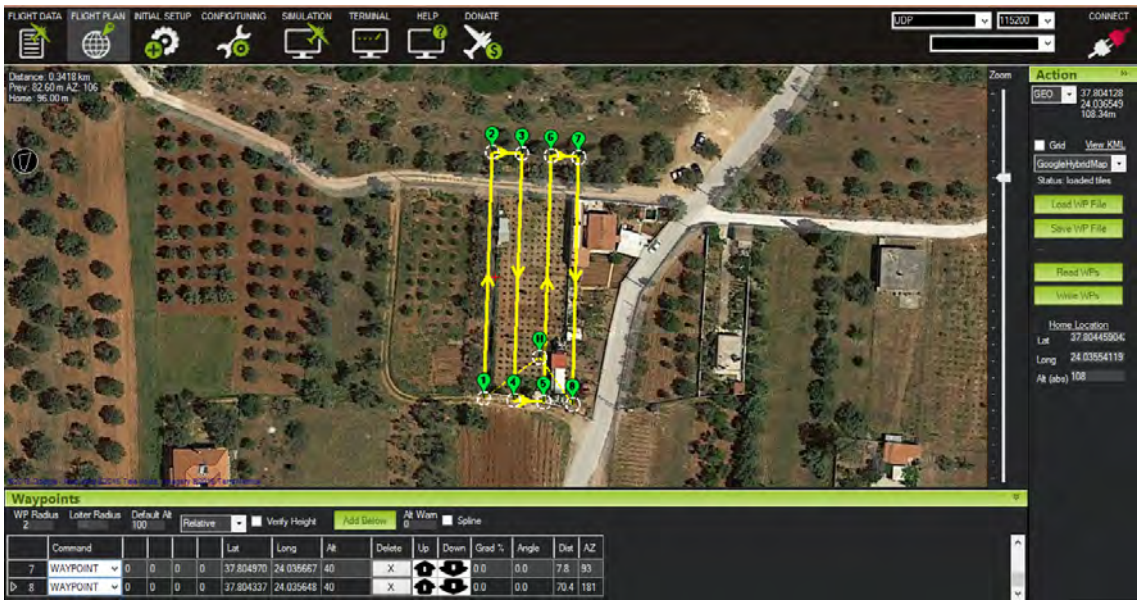

Figure 10.2: The mission planning screen of Mission Planner

7. Go to point at (latitude: 37.804348, longitude: 24.035458, vertical offset: 40)
8. Go to point at (latitude: 37.804344, longitude: 24.035553, vertical offset: 40)
9. Go to point at (latitude: 37.804974, longitude: 24.035578, vertical offset: 40)
10. Go to point at (latitude: 37.804970, longitude: 24.035667, vertical offset: 40)
11. Go to point at (latitude: 37.804337, longitude: 24.035648, vertical offset: 40)
12. Disable camera triggering
13. Set desired airspeed at $6\,\mathrm{m\,s^{-1}}$
14. Return to home position

A snapshot of the Graphic User Interface (GUI) of the Mission Planner software during the definition of the mission can be found in Figure 10.2.

The mission was completed by the UAV in about 100 s.

10.3 Image rectification

As each image was arriving from the UAV to the GCS laptop, it was processed by the corresponding ROS node. Let us examine the output of each processing step, on an example image, randomly selected from the dataset of the mission.

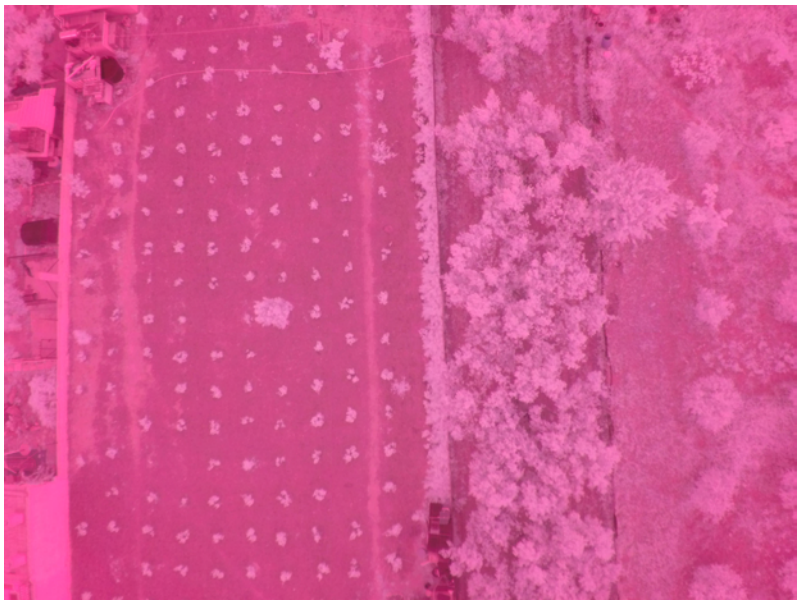

Figure 10.3: A multispectral image, captured during the UAV mission

The image under discussion, in its original form as was captured and created from the camera can be seen in Figure 10.3.

The orientation of the image is inverted, with the southern direction corresponding to the top of the image.

As has been discussed before the camera, as a combination of a lens and a sensor, has imperfections which distort the image geometrically. The camera calibration task has been performed offline once and the calibration coefficients have been extracted, but they need to be applied for every new image.

Camera calibration allows for the extraction of the transformation parameters from the image frame to the camera frame. Prior to calibration, it would be impossible to deduce the parameters of a ray stemming for the focal point and crossing a pixel in the image. The calibration procedure is also expected to correct for the so-called fish-eye effect that most lenses (especially the wide angle ones) suffer from, which essentially "bends" the lines outwards, the further to the edge of the image.

The rectified image is displayed in Figure 10.4, to help visualize the effect of image rectification.

In the original image, the fish-eye distortion is apparent in the concrete wall at the left side of the image, which appears obviously bent. After the image is calibrated, the concrete wall is significantly more straight. Note that black edges have appeared to the bottom and right of the image, binding towards the image center. This is a result of the image calibration procedure. They represent missing information which, geometrically should be present at that spot, but due to distortion, the lens could

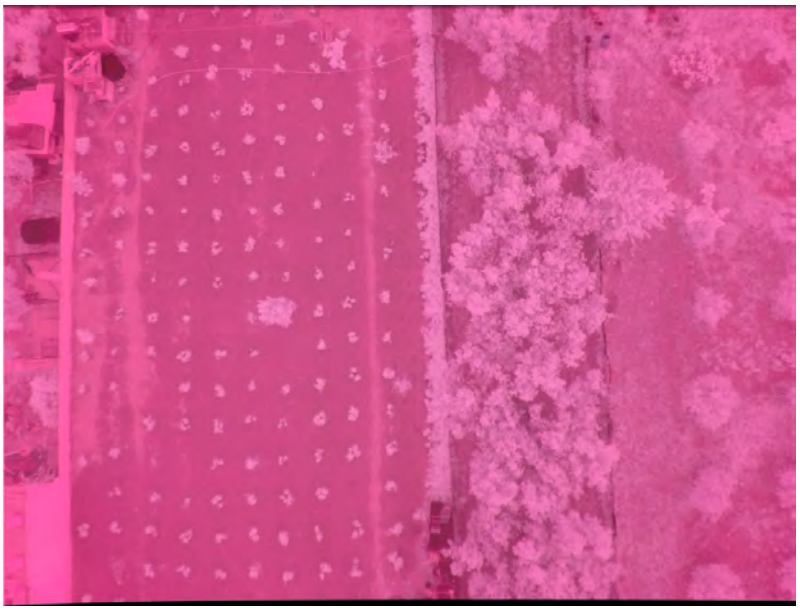

Figure 10.4: The rectified image

not capture it.

If the rectification procedure is not applied, then, depending on the camera imperfections and the height of flight, the introduced error might reach meter-level.

10.4 NDVI Extraction

We cannot make any remarks on the vegetation health from Figure 10.3. The NDVI operation must take place next. We remind that, as stated in Section 9.6, the correspondence between the actual colour bands and the camera channels is:

1. Image red corresponds to red band
2. Image green corresponds to green band
3. Image blue corresponds to NIR band, plus some red band leakage

The colour channels, as are extracted and singled-out from the raw image can be seen in Figure 10.5.

The RED band is corrected for NDVI leakage by subtracting the NDVI band from the red band

$$RED = RED_{corrputed} - NIR \tag{10.1}$$

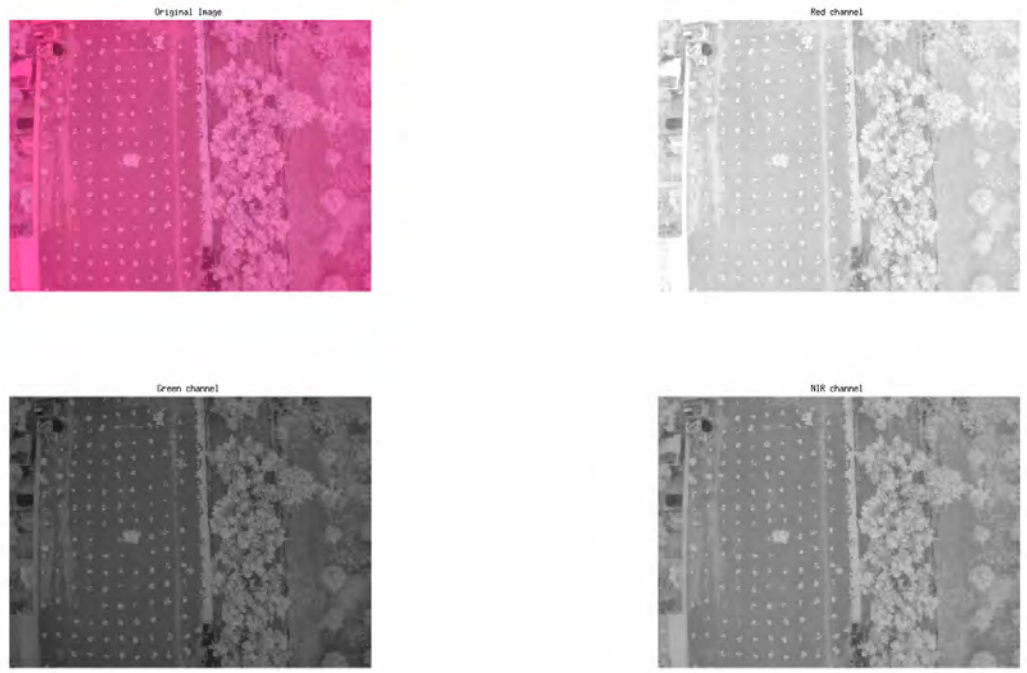

Figure 10.5: The original image and its three colour components

and the NDVI operation (see Section 5.2) is performed:

$$NDVI = \frac{NIR - RED}{NIR + RED} \tag{10.2}$$

$$= \frac{NIR - RED_{corrupted} + NIR}{NIR + RED_{corrupted} - NIR} \tag{10.3}$$

$$= \frac{2 \cdot NIR - RED_{corrupted}}{RED_{corrupted}} \tag{10.4}$$

The result of this operation on the sample image can be seen in Figure 10.6. A blue-to-red colormap has been applied for better visibility.

It is verified that the highest-scoring areas, shown in deep red, are those where lush vegetation is found. Each grapevine is clearly distinguishable from the ground. The buildings have a low score and the water surface inside the tank is evaluated at the lowest bottom of the scale, as expected. There is high distinguish-ability between vegetation and man-made structures.

It is of interest to extract from this scalar image a pixel mask, which will isolate the plats which is at the lower limit of the vegetation interval and witness low health scores. In theory, this is interval is (0.25, 0.35), but in practice, this has to

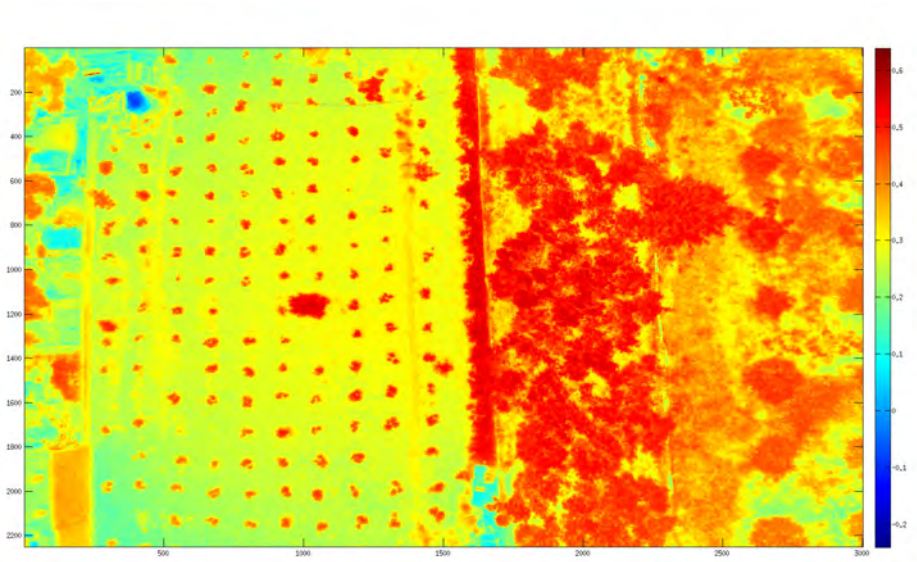

Figure 10.6: The NDVI operation applied on the sample image

be empirically determined, because the camera is not calibrated for the reflectivity of the surfaces. In our case, the unhealthy plant interval was situated around 0.5.

The result of the masking procedure for this interval can be seen in Figure 10.7.

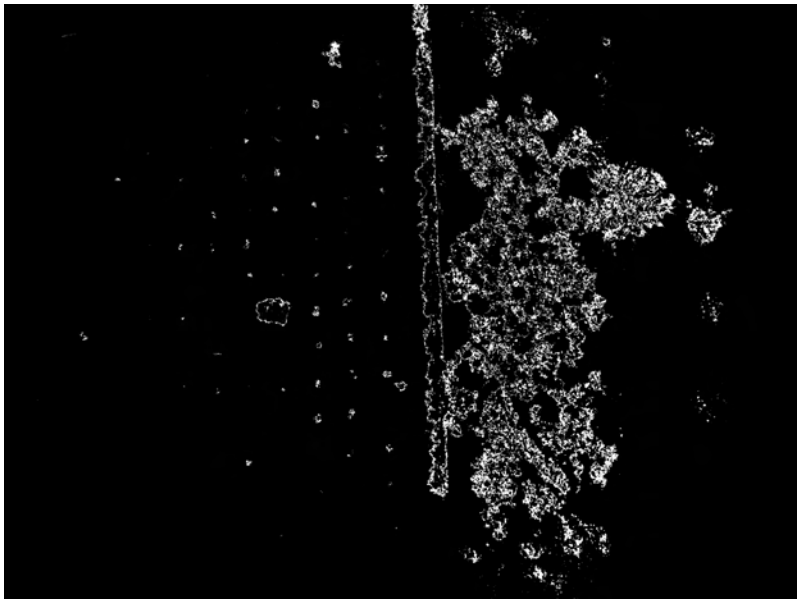

Figure 10.7: The corresponding thresholded NDVI index image

10.5 Points of Interest Geolocation and Aggregation

All of the white points (in the form of pixels) in Figure 10.7 constitute locations which need intervention and thus, are of interest. In this state, their coordinates are only known in the image frame, which isn't a very useful representation. In order for this information to drive the rover on the field, it needs to be converted into an inertial frame, i.e. NED, UTM or WGS84.

This is possible by combining the known transformation parameters between consecutive frames. The conversion from the image frame to the camera frame is possible because the camera calibration procedure provides a transformation from the image pixel coordinates to the image frame, given the distance of the object from the screen. In turn, this is known, because the flight height is user-selectable.

Afterwards, the conversion from the camera frame to the body frame is a simple -90 degree yaw rotation.

The conversion from the body frame to the NED frame is possible thanks to the telemetry information that Pixhawk creates and sends back to the GCS in the form of a MAVLink stream. `mavros` is able to publish the transformation information under a ROS topic as well, which facilitates the procedure greatly.

Lastly, the conversion from NED to UTM or WGS84 is trivial, and depends on the initialization coordinates.

The data extracted from the flight at hand was numerous and trying to process and visualize all of them would be impossible, given the technical limitation of this project. Only 1 every 100th unhealthy point was converted from the pictures to NED frame. The result can be seen in Figure 10.8.

For reference, the flight path that the UAV traversed during its mission is depicted as a blue trace. All the unhealthy points that were extracted from the images are depicted as green dots. Note that they cover a much wider area that the projection of the flight path. This is to be expected, since the angle of the camera lens can reach far to the side of the UAV trace.

The sheer number of data points and the irregular shape of their distribution makes them unwieldy, when it comes to sending the rover for intervention. The rover must be ordered to visit and intervene to only a few locations on the field, to save time and resources, but at the same time affect the largest possible amount of vegetation in need. It is reasonable to take the density of the data points into account during the process of extraction of the waypoints which the rover will visit.

There are various methods of point grouping, mostly found under the informatics scientific field, where they are referred to as clustering methods [6]. In this work, the

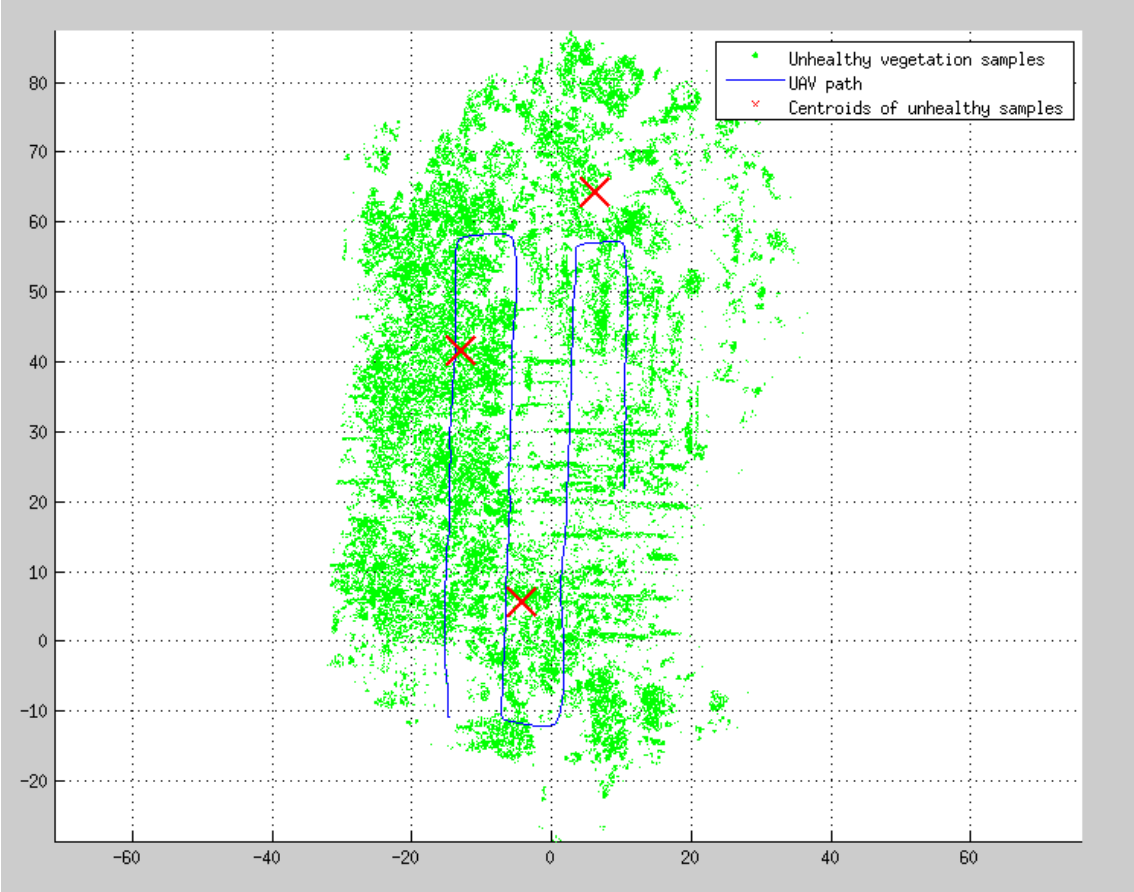

Figure 10.8: Points in need of intervention and the corresponding centroids

k-means method is used, as implemented in the *kmeans2* module of the *scipy.cluster* Python library.

The data points are given to the `kmeans2` algorithm as 2-dimensional points and a pre-defined number of centroids are returned. If all data points are assigned into disjoint groups, then the centroid of each group is the average of the points of that group. The computational difficulty stands in deciding which points should belong to which group and the *k-means* algorithm approaches it by starting with unit groups and merging the closest ones iteratively.

The centroids are ideal locations to intervene with the rover, since they are the locations which are exactly in the middle of a group of unhealthy measurements and are thus the most suitable for spreading the intervention agent.

In this work, 3 centroids were requested from the algorithm, which are then used to generate a path for the rover. Other centroid generation methods would be that of minimum distance with each other (to avoid overlapping intervention areas) or of predefined centroid radius (to reflect on the intervention area radius).

The centroids corresponding to the example data set, can be seen in Figure 10.8 as red X-marks.

10.6 Rover Trajectory Generation

The problem of navigating the rover through the field given a set of waypoints is a separate problem for the waypoint generation. The waypoints are created by the clustering algorithm and are passed onto the rover pathing problem. The solution of the pathing problem does not affect the waypoint generation. As a result, the rover pathing problem can be tackled completely separately.

In order to ensure repeat-ability and to avoid technical problems out of scope of this work, a secluded parking lot was selected as the test location for the rover pathing algorithm. The top view of the location can be seen in Figure 10.9. Light poles are visible which serve as permanent obstacles and so do the perimeter walls, but other artificial obstacles were also added.

The three points A, B, C defining the geometric area of the rover activity (as discussed in Section 7.2), as well as the direction of the virtual crop rows are also depicted in the image, corresponding to named points 1, 2 and 3. Point H is the starting location of the rover.

The coordinates of the delimiting points are:

Point #	Latitude	Longitude
A	37.9810554	23.7813543
B	37.9811167	23.7811598
C	37.9808239	23.7809989

The coordinates of the target waypoints are, in order:

Point #	Latitude	Longitude
WP1	37.9810322	23.7812550
WP2	37.9809613	23.7811357
WP3	37.9808292	23.7811209

3 waypoints are passed to the Odroid companion computer on the rover, as `NavSatFix` location messages, which correspond to the 3 centroid locations that the geolocation algorithm would produce. These are also seen in Figure 10.9.

Based on the rover initial location and depending on its orientation, for each successive waypoint the intermediate path which navigates through and around the crop rows is generated. In Figure 10.10 the automatically generated path from the initialization point to waypoint 1 can be seen. The intermediate corners are uploaded

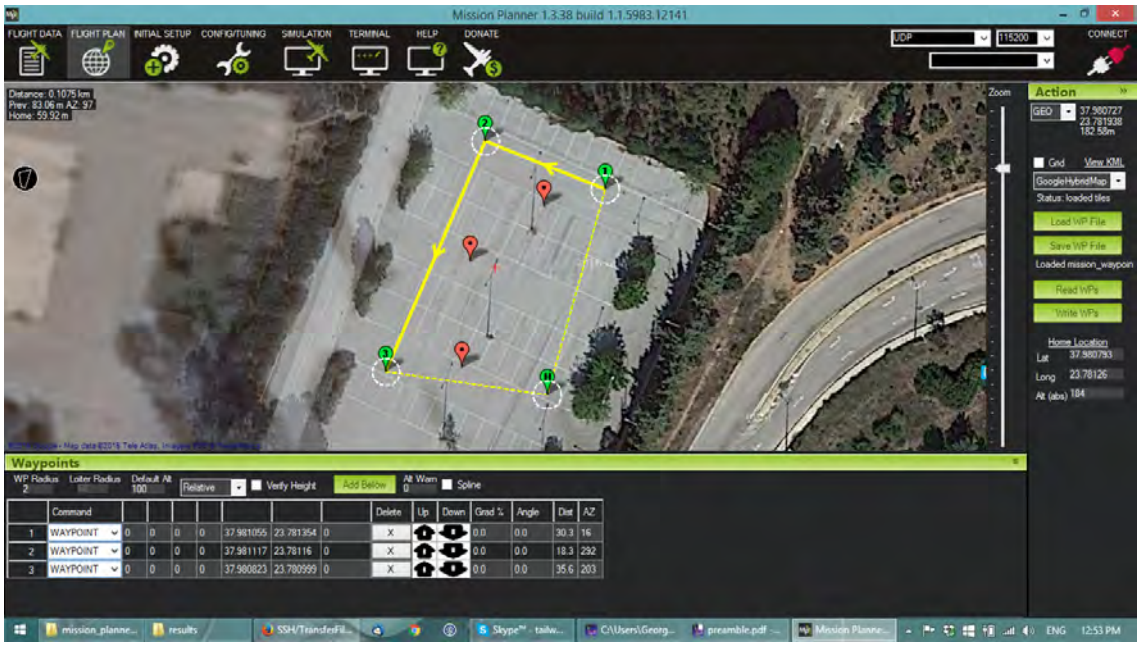

Figure 10.9: The location of the rover pathing test and its outline points

on the autopilot as mission waypoints. Points 1, 2 and 3 correspond to intermediate corners E, F and G of Figure 7.4.

After all the intermediate corners have been crossed and the first waypoint/centroid is visited, a new route, containing another two intermediate corners is generated. The new route can be seen in Figure 10.11 and the same holds for the final waypoint

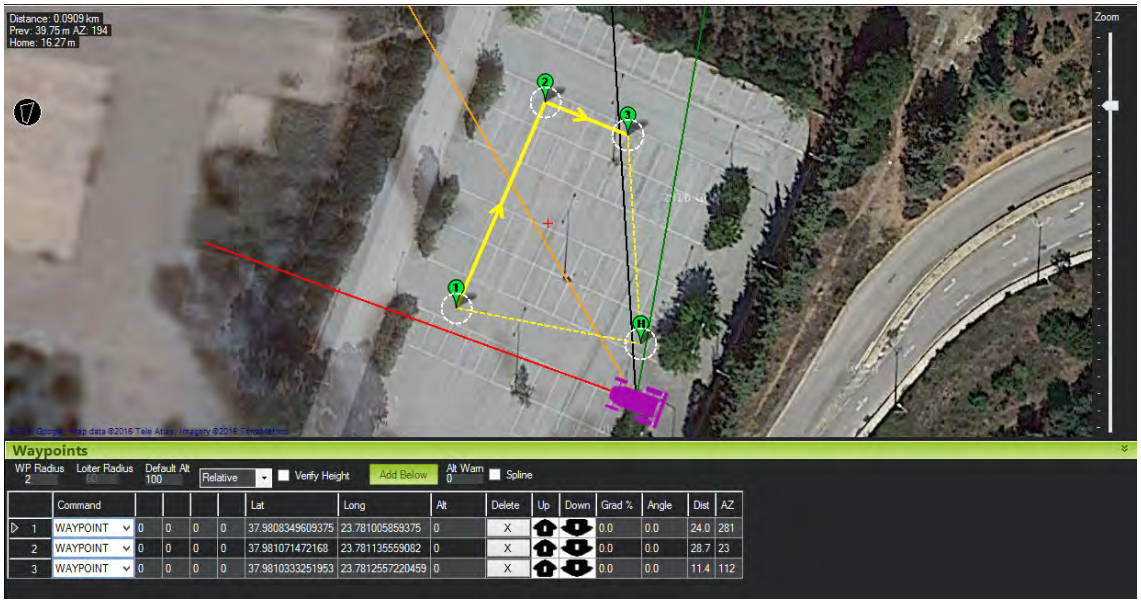

Figure 10.10: Intermediate corners to waypoint 1

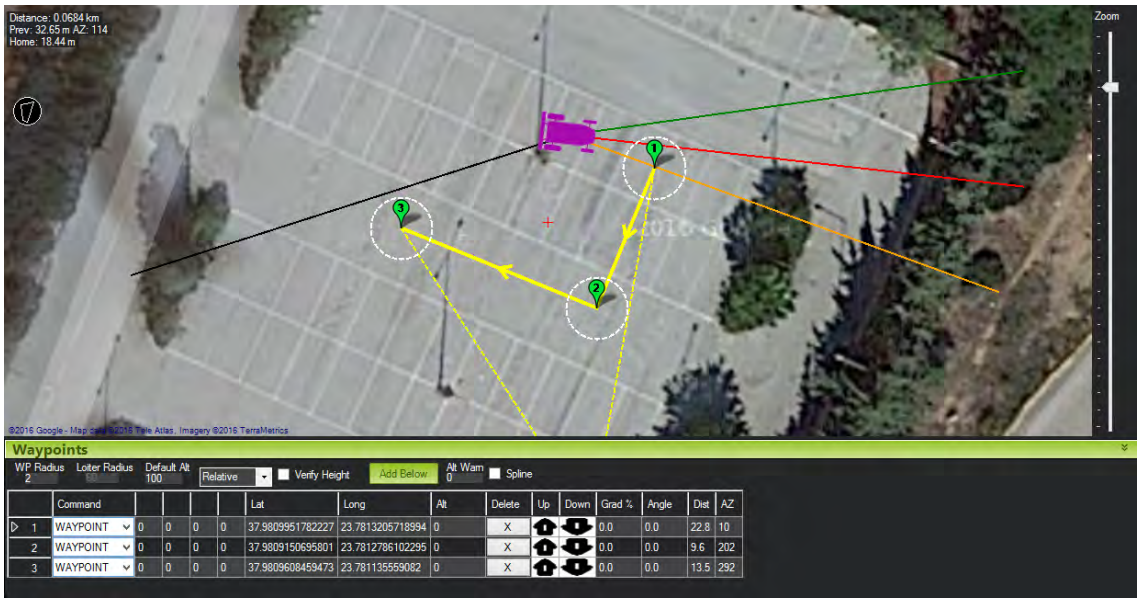

Figure 10.11: Intermediate corners to waypoint 2

route, visible in Figure 10.12.

For this section, the autopilot of the rover was allowed to take over and guide the rover through the generated missions. The result for each intermediate path can be seen in Figures 10.13, 10.14 and 10.15

Figure 10.12: Intermediate corners to waypoint 3

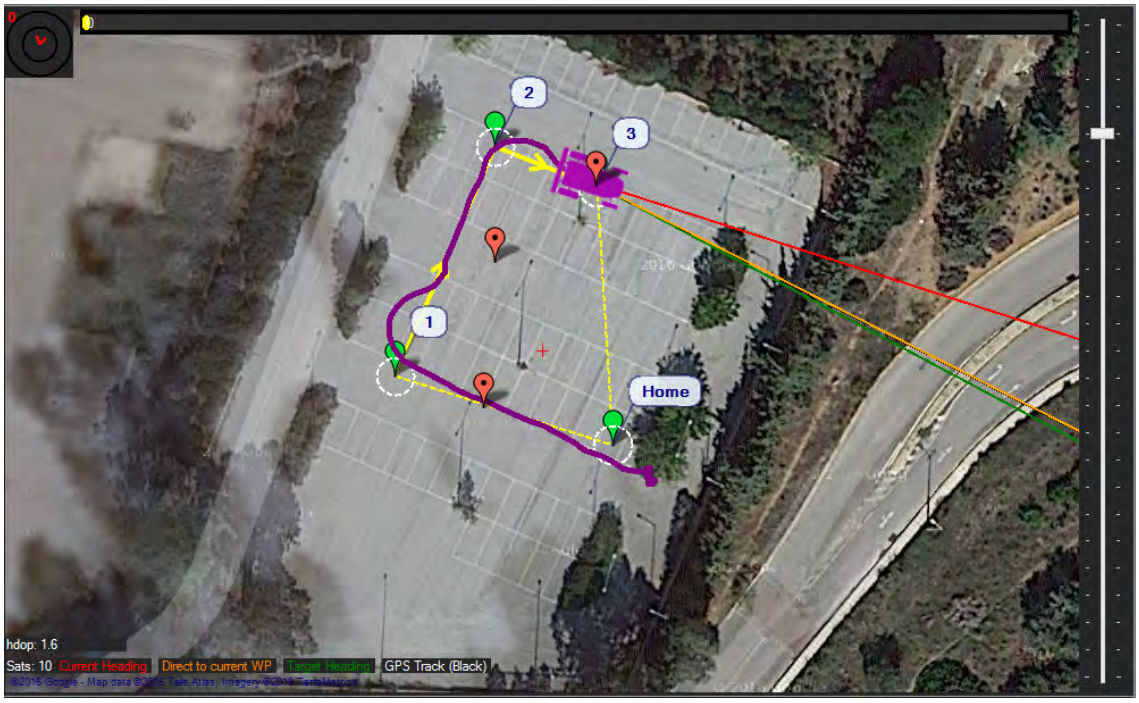

Figure 10.13: Rover trajectory for intermediate path 1

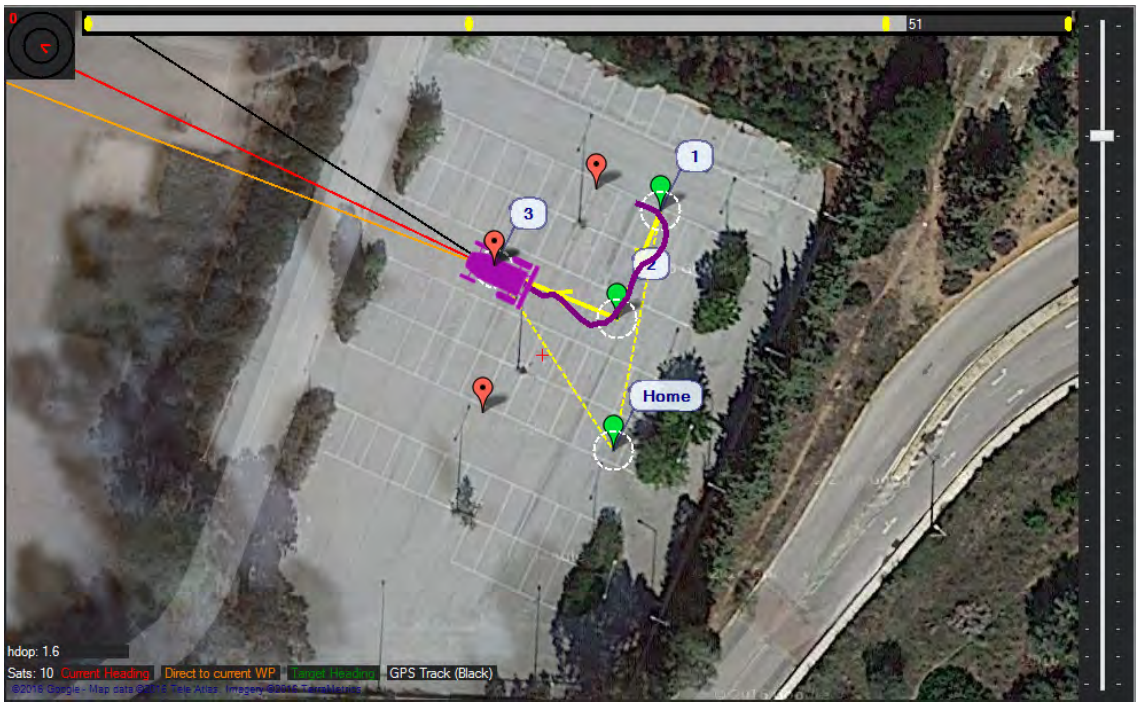

Figure 10.14: Rover trajectory for intermediate path 2

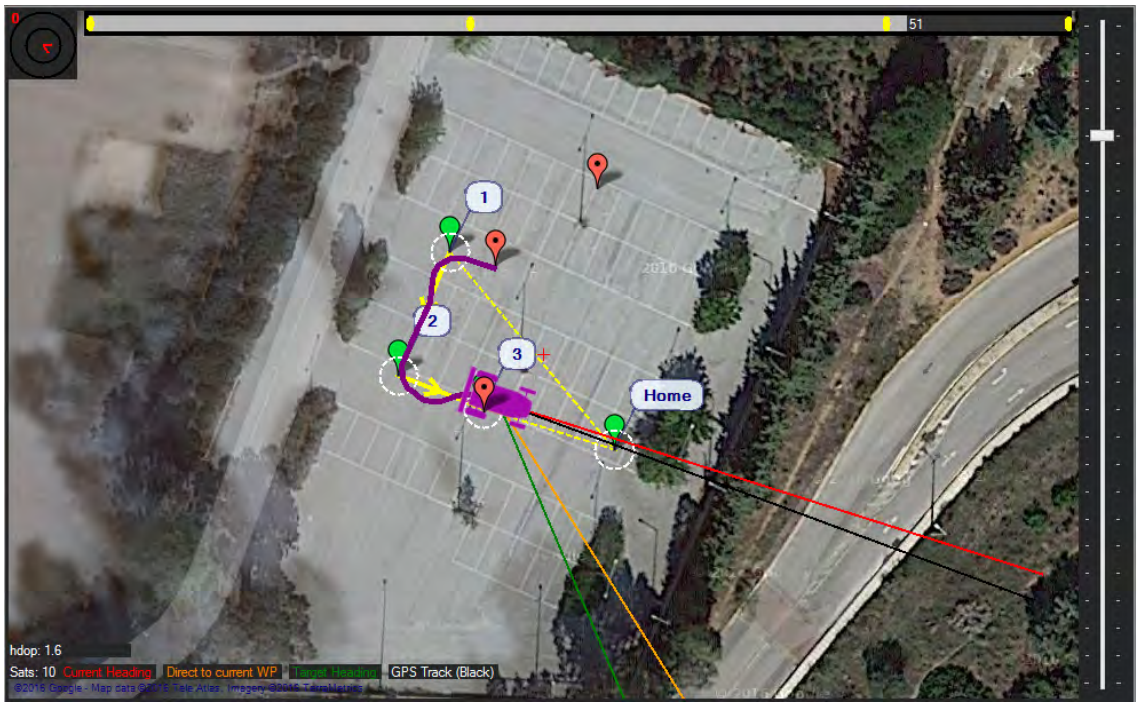

Figure 10.15: Rover trajectory for intermediate path 3

10.7 Obstacle Avoidance

In this section, the performance of the obstacle avoidance algorithm, based on vector fields will be presented. The corresponding tests were run on the same parking spot as the path generation tests, in order for the results to be comparable.

As seen on Figure 10.9, the most notable obstacles along and around the rover trajectory are the wall to the north of the parking lot, the flower beds to the east and west and the light poles in the middle of the lot.

Figures 10.16, 10.17 and 10.18 visualize the obstacles, as the rover perceives them through the laser scanner as well as the vector field components. The laser scanner data, subsampled for better visualization is displayed as red points. Notice how only obstacles in front of the rover are detected, since there is a 90 degree blind spot to the rear of the sensor. In accordance to Figure 7.7, the blue arrow points towards the next waypoint (displayed as a big blue circle), the red arrow is the repulsor field component and the green arrow is the vector sum of the blue and red arrows, the resulting desired direction that the rover should turn to, in order to align itself with the vector field.

Three interesting cases are isolated and discussed upon. The first one, visible in Figure 10.16, demonstrates a case where very little, far away obstacles are detected. This results in a very small repulsor vector. The attractor vector, since it has a constant magnitude, dominates over the vector sum and the green arrow generally

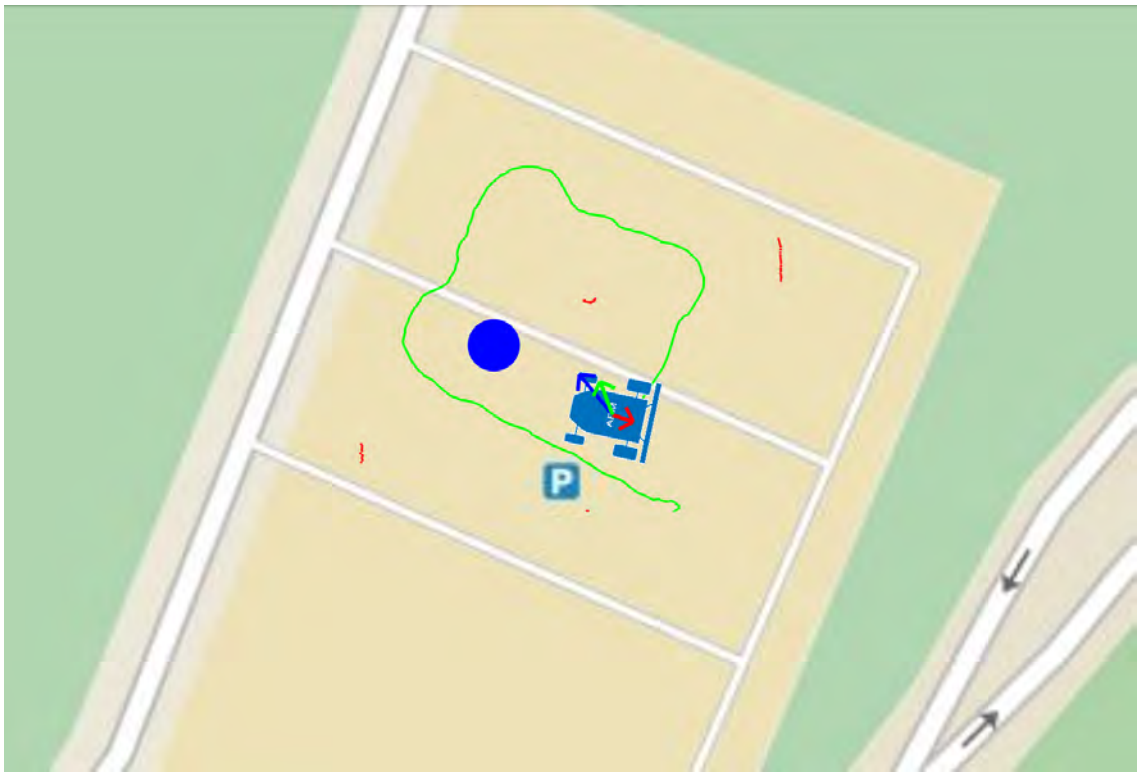

Figure 10.16: Attractor vector dominating the desired heading

points in the same direction as the blue arrow. The rover moves directly towards the next waypoint.

The second case is seen on Figure 10.17. The rover is very close to the obstacle, the flower bed to its right side. Thus, the magnitude of the repulsor vector grows very large, trying to draw the rover away from the obstacle. The overall vector thus points significantly away from the next waypoint.

The third case is presented in Figure 10.18. We see a configuration where, despite a large obstacle area is presented in front and to the left of the rover, the repulsor vector points to the left as well. With careful inspection, the trace of the light pole can be seen very close and to the right of the rover. Due to the inverse square distance law which constructs the repulsor vector, the contribution of the pole is much greater than that of the other obstacles, due to its proximity. As a result, the rover will prioritize navigating away from the pole. This is the intended behavior, since the pole is very close and poses a more significant threat to the rover. When the rover has distanced itself adequately from the pole, the repulsor vector will change, to point away from the walls and flower beds.

Overall, the vector field construction performs as intended, successfully combining the attractor and repulsor field components, to produce a desired heading for the rover to follow. Even though the laser scan data is quite dense and with high trans-

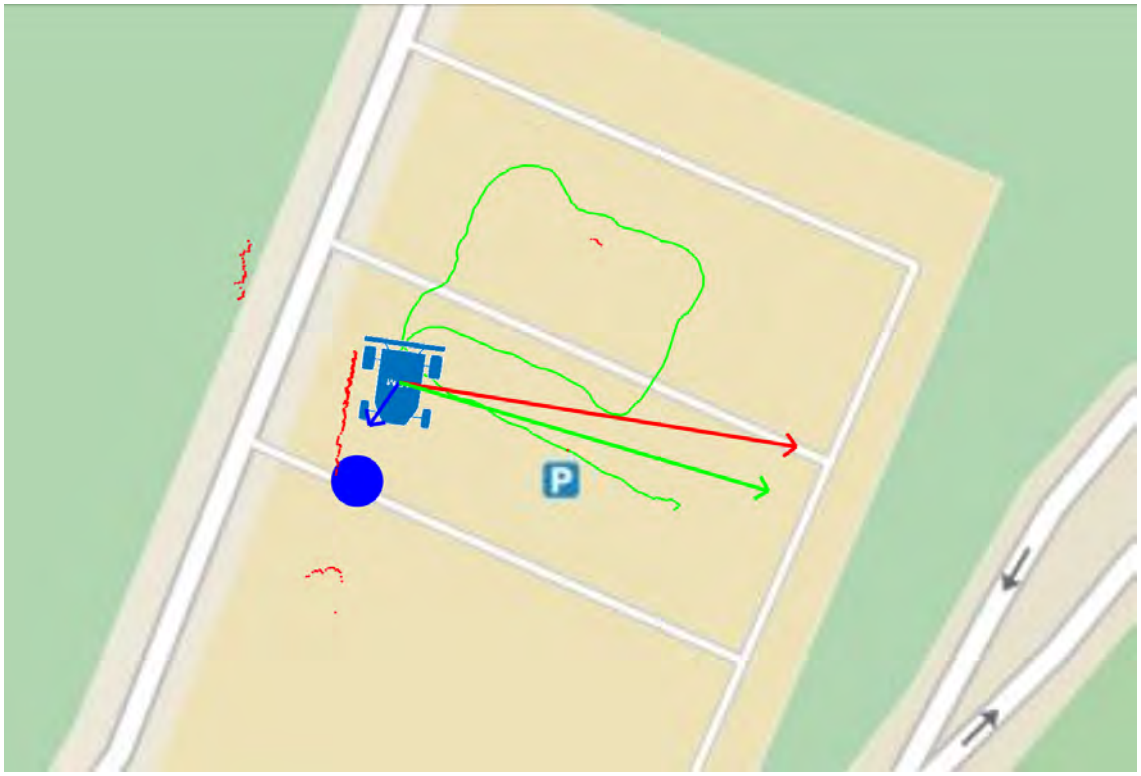

Figure 10.17: Repulsor vector dominating the desired heading

Figure 10.18: Nearest obstacle dominating the repuslor vector

fer rate, at about 220kB/s, the on-board Odroid computer manages to process it successfully.

10.8 Discussion and Future Work

The focus of this work was to integrate image processing and navigation methodologies with existing technical solutions to create a proof-of-concept precision agriculture system. In this context, the final result accomplished this goal.

A Unmanned Aerial System (UAS) which was able to capture imagery of the field was put together. The health assessment of vegetation was successfully performed with the use of NDVI. The resulting information was georeferenced in combination with the UAV odometry data and the points of interest in the field were localized. These points were transferred to the rover through a LAN in real time and a methodology for path generation was implemented. The rover was able to detect obstacles on its path and generate an obstacle avoidance strategy.

However, there naturally is room for improvement. On the topic of data collection, with the current implementation, the spatial overlapping of captured images is allowed and not taken into account. This can result in the same area appearing multiple times in the collected data, emphasizing any low health index and assigning it with unproportional weight. This problem can be mitigated by inserting a GIS between the image capture and processing. This will result in a single, non overlapping image, covering the whole field. This comes at a cost, though, because GIS software require a lot of time to process the input imagery and hence break the real-time nature of the application.

Regarding the performance of the rover used, it must be stated that its maneuverability was severely limited, in that it could not perform on-the-spot rotations, around its z-axis. This resulted in a serious pathing problem, and triggered the design the row go-around logic that was implemented. In general, it is a good rule of thumb that the hardware of any prototype system should be as capable as possible, so that the software will not hit implementation obstacles, such as model non-linearities and under-actuation. On the other hand, this has a significant impact on the cost of the hardware and a balance needs to be stroked.

On the way to system deployment, the intervention system (sprayer and container) need to be integrated onto the rover. This poses considerable technical difficulties. On the one hand, the system must be waterproof and ruggedized, but on the other antennae and the laser scanner need to protrude above the vehicle top side. Although similar vehicles exist, they utilize high-end and costly components and manufacturing procedures, which at this point don't seem to be avoidable.

Chapter 11

Conclusions

In this thesis we studied the use of an autonomous crop control system using robotic vehicles. Important conclusions were drawn regarding the feasibility of such a project, but also the benefits that the rural community will make with such an implementation. Significant results have also emerged on the interconnection of disparate systems so that a commercial or research package can be made ready for use outside the box. The main conclusions are the following:

1. A robotic system which be able to measure plant growth and health in agricultural crops as well as soil problems for small or large areas is feasible.

2. The time saved for checking a site is a fraction of what would be needed if a man or even a group of people were undertaking a similar task.

3. The overall cost reduction would also be significant for two reasons:
   a. the costs for seeds, fertilizers and pesticides distribution as well as paid labor, machinery expenses and fuels will be reduced.
   b. the robotic system requires little maintenance and can offer its services for many years.

4. The interconnection of dissimilar systems is feasible, and most importantly, this is done by using open-source software.

In addition, some constraints have also been observed that should be taken into account when developing such a system. The main points are as follow:

1. The final system should be proportional to cultivation coverage. The presented system is satisfactory for small crops due to the restriction of communications and the duration of the drone flight. In large crops, another type of drone should be studied. It should be able to cover larger areas and carrying proper communication systems.

2. The type of ground will determine the capabilities and characteristics of the rover and the type of communication devices.

3. More accurate results could be derived if there was a correlation of the data obtained from the multispec camera in combination with a sunlight gauge so as to minimize the errors of the daylight fluctuation.

A further improvement could be the implementation of a centralized database storing data from individual flights in different areas all over the country. Each record will include data such as: crop types, date, weather conditions, temperature, soil type, etc. With data mining systems, conclusions can be drawn for improving agriculture quality. This will be not only great advantage to the farmers, making each field as productive as possible but as well as for the research community.

Bibliography

[1] Px4 project. http://px4.io/.

[2] *Small Unmanned Aircraft: Theory and Practice*. 2012.

[3] Naze32 flight controller. http://www.abusemark.com/downloads/naze32_rev3.pdf, 2016. [Online; accessed 12-May-2016].

[4] 3DRobotics. Dronekit. http://dronekit.io/.

[5] 3DRobotics. Diydrones.com. http://diydrones.com/, 2016. [Online; accessed 12-May-2016].

[6] Rajaraman A. and Ullman Jeffrey D. *Mining of Massive Datasets*. 2010.

[7] J. S. Aber, S. W. Aber, L. Buster, W. E. Jensen, and R. O. Sleezer. "Challenge of infrared kite aerial photography: A digital update". *Kansas Academy of Science Transactions*, 112:31–39, 2009.

[8] BirdsEyeView Aerobotics. Firefly y6. http://www.birdseyeview.aero/products/firefly6.

[9] Airware. Flight core. https://www.airware.com/products/flight-core.

[10] Steven Dade Andrew Tridgell. Mavproxy. http://dronecode.github.io/MAVProxy/html/index.html.

[11] Tobias Bieniek Bart van Andel and Torstein I. Bo. utm python library. https://pypi.python.org/pypi/utm.

[12] Multi-Robot Systems Lab Boston University. Obstacle avoidance. http://sites.bu.edu/msl/research/coordinated-aggressive-control-of-aerial-vehicles/, 2016. [Online; accessed 23-May-2016].

[13] X. Burgos-Artizzu, A. Ribeiro, M. Guijarro, and G. Pajares. "Real-time image processing for crop/weed discrimination in maize fields". *Computers and Electronics in Agriculture*, 75:337–346, 2011.

[14] X. Burgos-Artizzu, A. Ribeiro, A. Tellaeche, G. Pajares, and C. Fernandez-Ouintanilla. “Analysis of natural images processing for the extraction of agricultural elements". *Image and Vision Computing*, 28:138–149, 2010.

[15] W. W. Casady, M. R. Paulsen, and J. F. Reid. “A trainable algorithm for inspection of soybean quality". *ASAE*, pages 90–7522, 1990.

[16] A. Chaimowicz, D. Cowley, Gomez-Ibanez B., M. Grocholsky, H. Hsieh, J. Hsu, V. Keller, R. Kumar, and C. T. Swaminathan. “Deploying air-ground multi-robot teams in urban environments". *International Workshop on Multi-Robot Systems*, Washington, DC (USA), 2005.

[17] L. Chaimowicz, B. Grocholsky, J. F. Keller, R. V. Kumar, and C. J. Taylor. “Experiments in multirobot air-ground coordination". *IEEE International COnference on Robotics and Automation*, 4:4053–4058, 2004.

[18] D. Daniels. *Ground Penetrating Radar*. IET, Sonar, Navigation and Avionics Series, 2007.

[19] SciPy developers. Scipy. http://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.kmeans2.html#scipy.cluster.vq.kmeans2.

[20] DJI. A2. http://www.dji.com/product/a2.

[21] DJI. Phantom 4. https://www.dji.com/product/phantom-4.

[22] M. C. Dobson, F. T. Ulaby, M. T. Hallikainen, and M. A. El-rayes. Microwave dielectric behavior of wet soil-part ii: Dielectric mixing models. *IEEE Transactions on Geoscience and Remote Sensing*, GE-23(1):35–46, Jan 1985.

[23] H. Duan and S. Liu. “Unmanned air/ground vehicles heterogeneous cooperative techniques: Current status and prospects". *Science China Technological Sciences*, 53:1349–1355, 2010.

[24] A. G. Eobin, G. Tiwari, R. N. Yadav, E. Peters, and S. Sadana. “UAV systems for parameter identification in agriculture". *Global Humanitarian Technology Conference: South Asia Satellite (GHTC-SAS), 2013 IEEE*, pages 270–273, 2013.

[25] Vladimir Ermakov. mavros ros package. http://wiki.ros.org/mavros.

[26] M. E. Everett. *Near-Surfaced Applied Geophysics*. Cambridge University Press, 2013.

[27] W .L. Felton, A. F. Doss, P. G. Nash, and K. R. McCloy. “A microprocessor controlled technology to selectively spot spray weeds". *In Proc. Automated Agriculture for the 21st Century Symp., Chicago,*, pages 427–432, 1991.

[28] J. Gago, C. Douthe, R. E. Coopman, P. P. Gallego, M. Ribas-Carbo, J. Flexas, J. Escalona, and H. Medrano. “UAVs challenge to assess water stress for sustainable agriculture". *Agricultural Water Management*, 153:9–19, 2015.

[29] M. Garzn, J. Valente, D. Zapata, and A. Barrientos. “An aerial-ground robotic system for navigation and obstacle mapping in large outdoor areas". *Sensors*, 13:1247–1267, 2013.

[30] G. J. Gaskin and J. D. Miller. “Measurement of soil water content using a simplified impedance measuring technique". *Journal of Agricultural Engineering Research*, 63:153–160, 1996.

[31] R. Gerhards and S. Christensen. “Real-time weed detection, decision making and patch spraying in maize sugarbeet winter wheat and winter barley". *Weed Research*, 43:385–392, 2003.

[32] R. Gerhards and H. Oebel. “Practical experiences with a system for site-specific weed control in arable crops using real-time image analysis and GPS-controlled patch spraying". *Weed Research*, 46:185–193, 2006.

[33] I. Giannakis, A. Giannopoulos, and C. Warren. A realistic fdtd numerical modeling framework of ground penetrating radar for landmine detection. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(1):37–51, Jan 2016.

[34] B. Grocholsky, S. Bayraktar, V. Kumar, and G. Pappas. “Uav and ugv collaboration for active ground feature search and localization". *Proc. of the AIAA 3rd “Unmanned Unlimited" Technical Conference*, 2004.

[35] D. E Guyer, G. E. Miles, M. M. Schrieiber, O. R. Mitchell, and V. C. Vanderbitt. *Transactions of the ASAE*, 29.

[36] Hardkernel. Odroid embedded computers. http://www.hardkernel.com/main/main.php.

[37] A. Harrigton. “Who controls the drones". *Engineering and Technology*, pages 80–83, 2015.

[38] Y. Inoue, S. Morinaga, and A. Tomita. “A blimp-based remote sensing system for low-altitude monitoring of plant variables: A preliminary experiment for agricultural and ecological applications". *Remote Sensing of Environment*, 21:379–385, 2000.

[39] Itseez. Opencv software library. http://opencv.org/.

[40] S. K. Khijwania, K. L. Srinivasanb, and J. P. Singha. “An evanescent wave optical fiber relative humidity sensor with enhanced sensitivity". *Sensors and Actuators B: Chemical*, 104:217–222, 2005.

[41] W. Kunzler, S. G. Calvert, and Laylor M. “Measuring humidity and moisture with fiber optic sensors". *Proceedings of SPIE*, 5278:86–pp, 2003.

[42] W. S. Lee, C. Slaughter, and D. K. Giles. “Robotic weed control system for tomatoes". *Precision Agriculture*, 1:95–113, 1999.

[43] Lockheed Martin. Kestrel autopilot. http://www.lockheedmartin.gr/us/products/procerus/kestrel-autopilot.html.

[44] Tom Igoe Gianluca Martino Massimo Banzi, David Cuartielles and David Mellis. Arduino microcontroller. https://www.arduino.cc/.

[45] Lorentz Meier. Mavlink - micro air vehicle message marshaling library. https://github.com/mavlink/mavlink, 2009–2014.

[46] Patrick Mihelich. image_transport ros package. http://wiki.ros.org/image_transport.

[47] Mikrocopter. Thermo 6s. http://www.mikrokopter.de/en/products/thermo6eng.

[48] M. S. Moran, Y. Inoue, and E. M. Barnes. “Opportunities and limitation for image-based remote sensing in precision crop Management". *Remote Sensing of Environment*, 61:319–346, 1997.

[49] M. Mozib and N. Zhang. “Weed detection using color machine vision". *Transactions of the ASAE*, 43:1969–1978, 2000.

[50] WM Mulaire. Department of Defense: World Geodetic System 1984. Technical report, National Imagery and Mapping Agency, 2000.

[51] S. R. Nandurkar, V. R. Thool, and R. C.. Thool. “Design and development of precision agriculture system using wireless sensor network". *First International Conference on Automation, Control, Energy and Systems (ACES), Hooghy*, pages 1–6, 2014.

[52] D.R. Nelson, D.B. Barber, T.W. McLain, and R.W. Beard. Vector field path following for small unmanned air vehicles. *American Control Conference, 2006*, pages 7–pp, 2006.

[53] Hrvoje Niksic. Gnu wget. https://www.gnu.org/software/wget/.

[54] Michael Oborne. Mission planner. http://ardupilot.org/planner/docs/mission-planner-overview.html.

[55] Multiple parties. Dronecode foundation. https://www.dronecode.org/.

[56] James Bowman Patrick Mihelich. image_pipeline ros package. http://wiki.ros.org/image_pipeline.

[57] A. J. Perez, F. Lopez, J. V. Benlloch, and S. Christensen. "Colour and shape analysis techniques for weed detection in cereal fields". *Computers and Electronics in Agriculture*, 25:197–212, 2000.

[58] Ardupilot project. Ardupilot autopilot. http://ardupilot.org/ardupilot/index.html.

[59] OpenPilot project. Cc3d flight controller. http://opwiki.readthedocs.io/en/latest/user_manual/cc3d/cc3d.html.

[60] Paparazzi project. Paparazzi-compatible autopilots. https://wiki.paparazziuav.org/wiki/Main_Page.

[61] Ardupilot projects. Supported boards by ardupilot. http://ardupilot.org/dev/docs/supported-autopilot-controller-boards.html, 2016. [Online; accessed 12-May-2016].

[62] Philip Rowse Laurens Mackay Dominik Honegger Julian Oes Sam Kelly Jeff Wurzbach Craig Elder px4dev, Lorenz Meier. Pixhawk flight controller. http://dev.px4.io/hardware-pixhawk.html.

[63] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[64] J. M. Reynolds. *An Introduction to Applied and Environmental Geophysics*. John Wiley and Sons Ltd, 2013.

[65] RM Rogers. *Applied mathematics in integrated navigation systems*. AIAA, 2 edition, 2003.

[66] C. A. Rokhmana. "The potential of UAV-based remote sensing for supporting precision agriculture in Indonesia". *Procedia Environmental Sciences*, 24:245–253, 2015.

[67] T. P. Seang and J. Mund. "Balloon based geo-referenced digital photo technique: a low cost high resolution option for developing countries". *In Proceedings of XXIII FIG Congress. Munich, Germany*, 2000.

[68] T. Seiyama, N. Yamazoe, and H. Arai. "Ceramic humidity sensors". *IEEE Transactions on Components Hybrids, and Manufacturing Technology*, 3:85–96, 1980.

[69] D. J. Selkowitz, G. Green, B. Peterson, and B. Wylie. “A multisensor LIDAR, multi-spectral and multi-angular approach for mapping canopy heigh in boreal forest regions". *Remote Sensing of Environments*, 121:458–471, 2012.

[70] SICK. Lms111 laser scanner. https://www.sick.com/media/pdf/2/42/842/dataSheet_LMS111-10100_1041114_en.pdf.

[71] J. V. Stafford. “Implementing precision agriculture in the 21st century: a review". *Journal of Agricultural Engineering Research*, 76:267–275, 2000.

[72] Cloud Cap Systems. Piccolo autopilots. http://www.cloudcaptech.com/products/auto-pilots.

[73] M. Takahashi, M. Shimada, T. Tadono, and M. Watanabe. “Calculation of tree heigh using PRISM-DSM". *International Journal of Remote Sensing*, pages 6495–6498, 2012.

[74] L. Tang, L. Tian, and J. Steward, Reid. “Texture-based weed classification using Gabor wavelets and neural networks for real-time selective herbicide applications". *Transactions of the American Society of Agricultural Engineers*, 99:pp–pp, 1999.

[75] H. G. Tanner and D. K. Christodoulakis. “Decentralized cooperative control of heterogeneous vehicle groups". *Robotics and Autonomous Systems*, 55:811–823, 2007.

[76] Alexander Tidenko. multimaster_fkie ros package. http://wiki.ros.org/multimaster_fkie.

[77] A. Tyagi, A. A. Reddy, J. Singh, and S. R. Chowdhury. “A low cost portable temperature-moisture sensing unit with artificial neural network based signal conditioning for smart irrigation applications". *International Journal on Smart Sensing and Inteligent Systems*, 4:94–104, 2011.

[78] J-B. Vioix, J-P Douzals, F. Truchetet, L. Assemat, and J-P Guillemin. “Spatial and spectral methods for weed detection and localization". *Journal on Applied Signal Processing*, 7:679–685, 2002.

[79] L. T. Waser, E. Baltsavias, K. Ecker, H. Eisenbeiss, C. Ginzler, M. Kuchler, P. Thee, and L. Zhang. “High-resolution digital surface models (DSMs) for modeling fractional shrub/tree cover in a mire environment". *International Journal of Remote Sensing*, 29:1261–1276, 2008.

[80] J. Weaver. *Collaborative coordination and control for an implemented heterogeneous swarm of UAVS nad UGVS*. PHd thesis submitted at the Univeristy of Florida, 2 edition, 2014.

[81] Wikipedia. Bell Boeing V-22 Osprey — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Bell%20Boeing%20V-22%20Osprey&oldid=718000997, 2016. [Online; accessed 12-May-2016].

[82] Wikipedia. I2C — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=I%C2%B2C&oldid=716229181, 2016. [Online; accessed 12-May-2016].

[83] Wikipedia. Universal Transverse Mercator coordinate system — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Universal%20Transverse%20Mercator%20coordinate%20system&oldid=717190708, 2016. [Online; accessed 13-May-2016].

[84] D. Woebbecke, G. Meyer, K. VonBargen, and D. Mortensen. "Color indices for weed identification under various soil, residue, and lighting conditions". *Transactions of ASAE*, 38:271–281, 1995.

[85] X. Wu, W. Xu, Y. Song, and M. Cai. "A detection method of weed in wheat field on machine vision". *Procedia Engineering*, 15:1998–2003, 2011.

[86] C. Yang, J. H. Everitt, Q. Du, B. Luo, and J. Chanussot. "Using high-resolution airborne and satellite imagery to assess crop growth and yield variability for precision agriculture". *Proceedings of the IEEE*, 101:582–592, 2013.

[87] C-C Yang, S. O. Prasher, J. A. Landry, H. S. Ramaswamy, and A Ditommaso. "Application of artificial neural networks in image recognition and classification of crop and weeds". *Canadian Agricultural Engineering*, 42:147–152, 2000.

[88] P. J. Zarco-Tejada, R. Diaz-Varela, V. Angileri, and P. Loudjani. "Tree height quantification using very high resolution imagery acquired from an unmanned aerial vehicle (UAV) and an automatic 3D photo-reconstruction method". *International Journal of Remote Sensing*, 55:89–99, 2014.

[89] C. Zhang and J. Kovacs. "The application of small aerial systems for precision agriculture: a review". *Precision Agric*, 13:693–712, 2012.

[90] C. Zhang, D. Walters, and J. Kovacs. "Applications of low altitude remote sensing in agriculture upon farmers requests- A case study in northeastern Ontario, Canada". *Plos One*, 9, 2013.

[91] N. Zhang and C. Chaisattapagon. "Effective criteria for weed identification in wheat fields using machine vision". *Transactions of the ASAE*, 38:965–974, 1995.