Detecting and Approaching Points of Interest with Drones using Visual Markers

Εντοπισμός και προσέγγιση σημείων ενδιαφέροντος με Drones χρησιμοποιώντας οπτικά σημάδια

Ioannis Badakis

Supervisor: Assoc. Prof. Spyros Lalis

2nd committe member: Assist. Prof. Antonios Argyriou

A Thesis submitted in fulfillment of the requirements
for the degree of Diploma Thesis
in the
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

September 2018

Dedicated to

my family

Εντοπισμός και προσέγγιση σημείων ενδιαφέροντος με Drones χρησιμοποιώντας οπτικά σημάδια

Περίληψη

Το αντικείμενο της διπλωματικής είναι ο εντοπισμός και η προσέγγιση οπτικών σημαδιών χρησιμοποιώντας μία κάμερα σε μη επανδρομένα οχήματα, όπως ένα Drone. Πιο συγκεκριμένα, για να εντοπιστούν τετράγωνα οπτικά σημάδια, χρησιμοποιούνται τεχνικές της βιβλιοθήκης OpenCV. Επιπλέον, για να επιτευχθεί η εκτιμήση της θέσης της κάμερας σε σχέση με τα οπτικά σημάδια, η κάμερα πρέπει να είναι βαθμονομημένη. Η διπλωματική προτείνει δύο μεθόδους για να καθοδηγήσει το Drone έτσι ώστε να προσεγγίσει τα σημάδια. Η πρώτη μέθοδος, χρησιμοποιεί διανύσματα μετατόπισης και περιστροφής που εξάγονται από την εκτίμηση της θέσης της κάμερας σε σχέση με τα οπτικά σημάδια. Η δεύτερη μέθοδος, βρίσκει τις GPS συντεταγμένες του οπτικού σημαδιού, κάνοντας χρήση την γνωστή GPS θέση του Drone, την απόσταση μεταξύ αυτού και του οπτικού σημαδιού αλλά και την γωνία που ορίζεται μεταξύ τους. Η κυριότερη διαφορά ανάμεσα στις δύο μεθόδους είναι η ακρίβεια της προσέγγισης, με την πρώτη να υπερτερεί έναντι της δεύτερης. Τέλος, οι δύο μέθοδοι μπορούν να συνδυαστούν, ώστε το όχημα, πρώτα, να προσεγγίσει τον στόχο μέσα στα πλαίσια της ακρίβειας του GPS, και στην συνέχεια να προσεγγίσει τον στόχο με μεγαλύτερη ακρίβεια χρησιμοποιώντας τα διανύσματα μετατόπισης και περιστροφής.

Detecting and Approaching Points of Interest with Drones using Visual Markers

Abstract

The aim of this thesis is to detect visual markers using a monocular camera in unmanned vehicles, such as quadcopters, in order to approach them. More specifically, to detect squared fiducial markers, OpenCV (Open Source Computer Vision library) techniques are used. Moreover, to achieve a pose estimation of the camera with respect to the visual marker, a camera has to be calibrated. This thesis proposes two procedures to guide the quadcopter so as to approach the marker. The first procedure uses the translation and rotation vectors that are extracted from the pose estimation of the camera. The second procedure finds the GPS (Global Position System) position of the visual marker using the known GPS position of the quadcopter, the ground distance and the relative bearing between. The main difference between these methods is the precision of the approach, with the former having better accuracy. Finally, these two methods can be combined, so that the vehicle, firstly, approaches the target within the error range of the GPS, and secondly within the more accurate method of direct navigation commands.

Acknowledgements

First of all, I would like to thank to my supervisor Professor Spyros Lalis for his guidance, great support and kind advice throughout my undergraduate years, especially in the development of this thesis. It was an honour for me and privilege to absorb a small chunk of his exceptional knowledge.

I also would like to thank to my co-supervisor Professor Antonios Argyriou for his valuable advice and helpful suggestions for the accomplishment of this thesis.

I am grateful to my friend Tryfon Tsakiris for our excellent collaboration while working on several projects throughout these years, and my fellow students for all their support, unconditional friendship and patience.

Last, but not least, I would like to express my deepest gratitude to my parents and my brother who have always supported and helped me with their own unique way to achieve my goals, and will always be the most important thing of my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, a preface is given in order to understand the aim of this thesis. In particular, it explains why the camera pose estimation is an importance problem as well as what strategy does this thesis follows, in order to guide the drone so as to approach the visual marker.

1.1 Importance of the problem

Camera pose estimation is a popular problem in numerous computer vision applications such as robot navigation or AR (Augmented Reality), which is usually based on obtaining correspondences between known points on the environment and their camera projections-image points. Using natural features to estimate the pose, such as key points on images, is a very common strategy which does not require modifying the environment. However, the use of visual markers is still of great importance and an attractive approach for camera pose estimation since it provides easy detection and point correspondences more precisely, robustly and efficiently. Moreover, they, easily, allow the camera to extract pose from their four corners, given that the camera is properly calibrated. One only needs to create visual markers with a regular printer and place them in the target environment.

1.2 Approach

This thesis propose two methods in order the Drone to approach the visual marker. In the first method, the quadcopter navigates in the environment with the help of the camera pose relative to the visual marker. On the other hand, the second procedure makes use of the GPS coordinates. More specifically, it extracts the GPS coordinates of a the visual marker with the help of the known GPS position of the quadcopter, the distance and the relative bearing between them.

Moreover, the Gazebo simulator is used in order to implement and visualize a quadcopter model. In particular, the SITL (Software in the Loop) simulator of the ArduPilot, which allows to run a quadcopter without any hardware, interact with the Gazebo simulator. Also, a visual marker box is created within the Gazebo so as the camera, that is mounted on the gimbal of the quadcopter, can detect it.

The image processing of the frames that are produced by the quadcopter's camera, is done on ROS (Robotics Operating System) in order to detect the visual marker. After the extraction of the marker's GPS coordinates as well as the translation and rotation vectors between the camera and the visual marker, the ArduPilot flight controller receives commands via the MAVLink protocol so as the quadcopter can be guided successfully which will allow it to localize itself and approach the target.

1.3 Thesis structure

The rest of this thesis is structured as follows. Chapter 2 provides a background of the visual markers, the Gazebo Simulator as well as the ROS (Robot Operating System). Chapter 3 provides information, necessary for one, to understand the basic concepts of a perspective projection and a rigid transformation. In other words, how does a 3-dimensional world point is projected onto an image plane. Then, it explains why a camera calibration is an obligatory step in order to estimate the pose of a camera. Moreover, it shows the procedure of how the quadcopter extracts the visual marker's GPS coordinates, as well as the meaning of rotations in the 3-dimensional space. Finally, Chapter 3 illustrates the two methods that the quadcopter uses in order to approach the visual marker. Chapter 4 begins with a brief overview of each different component that have been implemented, so as the UAV (unmanned aerial vehicle) is able to reach the target - visual marker. Moreover, a bigger picture from a software engineering perspective is shown, so that the reader understands how the different components are combined. Finally, it demonstrates the procedures that are followed so as the quadcopter is able to localize itself and approach the target. In Chapter 5 an indicative test is presented in order to verify that what this thesis builds is actually works. Chapter 6 mentions some related works and the main differences between them and this thesis. Finally, Chapter 7 concludes the thesis and describes future work.

Chapter 2

Background

2.1 Visual Marker

Recognition of visual markers (Fig. 2.1a) is an intersection topic, which has applications in different areas. One application is AR (augmented reality), where with the help of computer vision one can find them in a picture or a video stream and substitute them (Fig. 2.1b) for artificially generated objects creating a view which is half real and half virtual - virtual objects in a real world. Another application is robotics, where markers can be used either to give commands to a robot or as directions so as a robot can navigate within some environment.

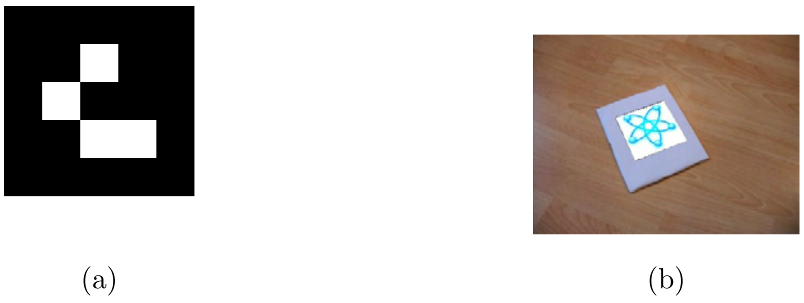


Figure 2.1: (a) Visual Marker and (b) Visual Marker with Augmented Reality.

2.1.1 Marker's composition

Markers are comprised by an external black border and an inner region that encodes a binary pattern (Fig. 2.2b). The binary pattern is unique and identifies each marker. Depending on the dictionary (which is a set of visual markers with equal bits), there are markers with more or less bits (Fig. 2.3). The more bits, the more words in the dictionary, and the smaller chance of a confusion. However, more bits means that more resolution is required for correct detection.

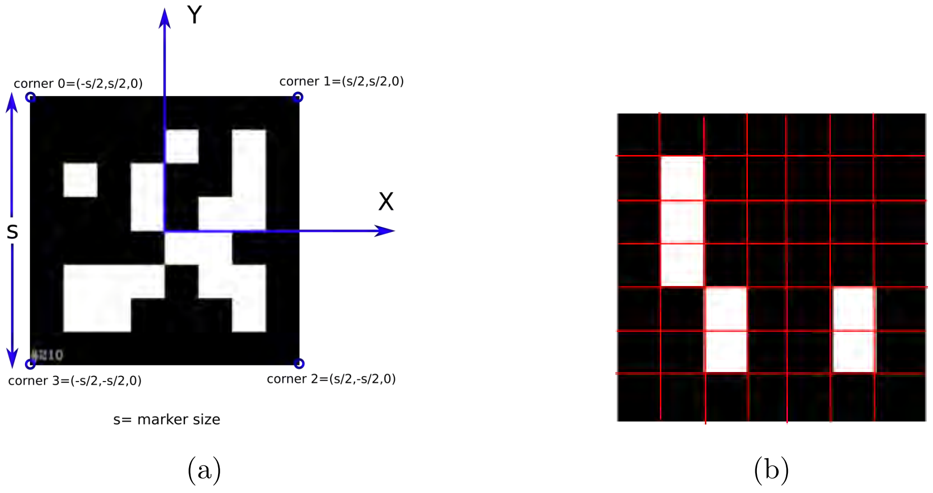


Figure 2.2: (a) Coordinate system of a marker (X is to the right, Y is up and Z is forward-out of the page). (b) The first and the last row - column of the mark is the black border and the rest is the inner part.

In other words, all the markers, are represented with a square grid divided equally to the same number of rows and columns. Each cell of the grid is filled with either black or white color. The first and the last row - column of each marker contain only black cells, creating a black border around each marker. All such markers are printed on white paper in such a way, that there is white area around black borders of a marker and we denote s the size of the marker once it is measured (Fig. 2.2a).

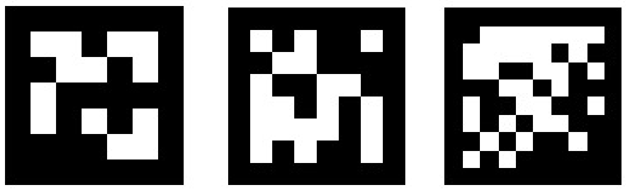


Figure 2.3: Markers of different sizes. Black cells denotes to 0 and white cells denotes to 1. From the left to right: n = 5, n = 6 and n = 8.

2.1.2 Marker Detection

In order to detect the visual marker, this thesis uses the ArUco library [1], [2], [3]. More specifically, the process (Fig. 2.4) of ArUco, which is comprised by several steps aimed to detecting rectangles and extracting the binary code from them, is as follows: *a)* Image segmentation, *b)* Contour extraction and filtering, *c)* Marker Code extraction and *d)* Corner refinement.

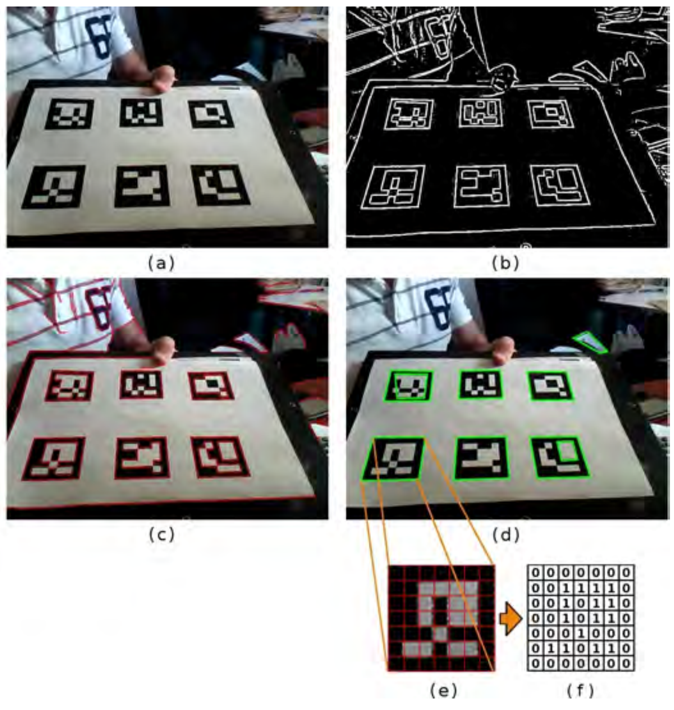


Figure 2.4: Steps of the process of marker detection.

Image segmentation

The goal of image segmentation is to partition an image into multiple segments and to simplify or change the representation of an image into something more meaningful and easier to analyze. In other words, image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

Since a marker is designed to have an external black border surrounded by a white space, the borders can be found by segmentation. In their approach, an adaptive thresholding is employed. More specifically, the mean intensity value m of each pixel is computed using a window size w. If the intensity value of its pixel is below than $m - c$ then the pixel is set to one, otherwise is set to zero, where the c is a constant value (Fig. 2.4b).

Contour extraction and filtering

On the thresholded image that was produced by the previous step, a contour (outline of a shape) extraction and filtering must be applied in order to take the four-vertex rectangles.

Firstly, using the Suzuki and Abe algorithm, we obtain a set of contours from the thesholded image. However, it is possible to have irrelevant background elements in the output image (Fig. 2.4c). That is to say, a filtering step is required in order to wipe out them. For this purpose, the Douglas-Peucker algorithm is used to perform a polygonal approximation. More specifically, we care about contours with four-corner polygons, since markers are squares. Thus, the elements, that do not approximate well to a four-corner polygon, are discarded. Finally, too small contours are discarded, leaving only the external ones. The resulting polygons from this process can be seen by (Fig. 2.4d).

Marker Code extraction

The ArUco library, in order to determine which of the remaining contours are valid markers, firstly, removes the perspective projection, so as to obtain a frontal view of the rectangle using a homography and then is thresholding the image using the Otsu's method, which provides the optimal image threshold value (Fig. 2.4e).

Then, the binarized image (Fig. 2.4f) is divided into a regular grid and each element is assigned the value 0 or 1 depending on the values of the majority of the pixels into it. For each marker candidate, it is necessary to determine whether it belongs to the set of valid markers or if it is a background element.

For example, if we divide the binarized image into a 6x6 grid, the 5x5 cells contains the id information, while the rest coorespond to the external black border. First, it must check that the external black border is present. After that, it reads the internal cells and see if they provide a valid code. Four possible identifiers are obtained for each candidate, corresponding to the four possible rotations of the canonical image. If any of the identifiers belong to the set of valid markers, then it is accepted.

Corner refinement

The last step, of the ArUco process, consists of estimating the location of the corners with subpixel accuracy.

Working with images on a digital system, the smallest part of an image is a pixel. There is no way to access information "between" pixels. Fig. 2.5 shows a corner that does not lie on a single pixel. Since several application, like tracking and camera calibration, require higher accuracy than a camera can provide, it is fundamental to find the corner more precisely.

To do so, the method estimates the lines of the marker's sides employing all the contour pixels and computes the intersections in order to get higher accuracy at the corner.

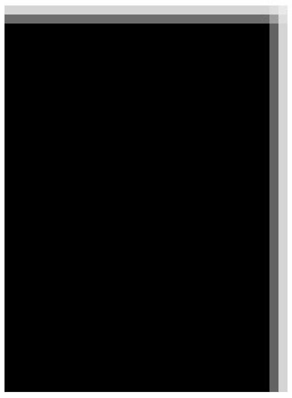


Figure 2.5: A picture of a square with a lot of zoom. The corner does not lie on a single pixel.

2.2 Gazebo

Robot simulation is a fundamental tool for every roboticist. A well-designed simulator offers the ability to immediately test algorithms, design robots and perform regression testing using realistic scenarios.

In order to simulate the quadcopter to detect and approach a visual marker without a hardware, this thesis uses a robot simulator.

Gazebo [4] offers this ability since it is a 3D dynamic simulator with the capability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.

2.2.1 Gazebo Components

In order to understand the components that comprise the Gazebo simulator, the thesis makes a brief overview about its terminology. More specifically, *a)* the term *Model* is used to describe all the objects, static or dynamic. To be more specific, models can range from simple shapes to complex robots. *b)* The term *Static* is for those objects that have only a collision geometry. In other words, all the objects which are not meant to move and they only declare their geometry in order to determine their collisions with the other elements of the environment. *c)* The term *Dynamic* is for those objects that have both inertia and collision geometry. This kind of objects, can be managed by a programmer in order to move them within the simulation environment or acquire their sensor data (if they have one). *d)* The term *Word* is used to describe a collection of models, i.e robots and static objects (such as buildings, tables, and lights), and global parameters including the sky, ambient light, and physics properties.

All of the above elements of the simulation are represented by the SDF (Simulation Description Format). More specifically, SDF is an XML format that describes robots, and environments for robot simulators, visualization, and robot control. Within the robot, SDF can describe it's kinematic and dynamic attributes, sensors, joint frictions and many more properties. Finally, within the environment, SDF describes the existence of the various models (such as robots, objects) and their interaction.

The items involved in describing the elements and running the Gazebo Simulation are:

World Files

The world description file contains all the elements in a simulation, such as robots, lights, sensors, and static objects. This file is formatted by SDF (Simulation Description Format) using XML, and typically has a .world extension.

A simple example of a world file is:

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <world name="default">

    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>

    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
  </world>
</sdf>
```

Model Files

A model file uses the same SDF format as world files, but should only contain a single tag <model>...</model>. Moreover, a model can have multiple sensors that acquire data. The purpose of these files is to facilitate model reuse, and simplify world files. Once a model file is created, it can be included in a world file using the following SDF syntax:

```
<include>
  <uri>model://model_file_name</uri>
</include>
```

A simple example of a model file that represents a box is:

```
<?xml version='1.0'?>
<sdf version="1.4">
  <model name="visual_marker">
    <pose>0 0 0.1 0 0 0</pose>
    <static>true</static>
      <link name="box">
        <collision name="collision">
          <geometry>
            <box>
              <size>0.2 0.2 0.2</size>
            </box>
          </geometry>
        </collision>
        <visual name="visual">
```

```
          <geometry>
            <box>
              <size>0.2 0.2 0.2</size>
              <!-- here one can use their own mesh with textures that improve
                  visual appearance -->
            </box>
          </geometry>
        </visual>
      </link>
    </model>
</sdf>
```

The tag *link* contains the physical properties of one body of the model. For the case that the model is a simple wheel, only one link must be listed in order to represent a model. However, a model may be more complex and consists of different parts, e.g, a table or a robot. In this case, a model is described by multiple links, each one of them corresponds to a different part of the model. These links are connected together with the joint tag. The *joint* tag connects two links, a parent and child relationship. For example, a table model could consist of 5 links (4 for the legs and 1 for the top) connected via joints.

Since each link contains the physical properties of one body of the model, these physical properties consist of collision and visual elements. A *collision* element encapsulates a geometry that is used to collision checking, while the *visual* element is used to visualize parts of a link in the simulator. Finally, a link may have inertial, sensor and light elements. An *inertial* element describes the dynamic properties of the link, like the mass of the body. The *sensor* element describes a sensor that a body of the model may have which collects data from the world. Finally, a *light* element describes a light source attached to a link.

Gazebo Server

The server parses a world description file given on the command line, and then simulates the world using a physics and sensor engine.

Graphical Client

The graphical client connects to a running server and visualizes the elements. This application provides a nice visualization of simulation. Moreover, the user within the client may also modify the scene by adding, modifying, or removing models. Additionally, there are some tools for visualizing and logging simulated sensor data.

Plugins

A plugin, or alternatively a controller, is a chunk of code that is compiled as a shared library and is inserted either into the SDF files (world or model) or as an argument in the command line (when the simulation starts). The plugin lets developers control almost any aspect of the functionality of Gazebo through the standard C++ classes. Moreover, one should use a plugin when *a)* want to programmatically alter a simulation, *b)* move models, respond to events, insert new models given a set of preconditions.

Plugins should be chosen based on the desired functionality and are managed by a different component of Gazebo. More specifically, a world plugin is attached to a world file and is able to control physics engine, lighting. A model plugin is attached to a model file and controls joints and state of a model. Moreover, one should use a sensor plugin to acquire sensor information or control sensor properties. Finally, a system plugin is specified on the command line and gives the user control the startup process. For example, a user can use a system plugin in order to save images from user camera of the simulator into a directory.

2.3 Robot Operating System

2.3.1 What is ROS?

The Robot Operating System (ROS) [5], [6], [7] is an open-source, meta-operating system for robots. It provides the services one would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

Moreover, the ROS runtime ”graph” is a peer-to-peer network of processes. In other words, a system built using ROS consists of a number of processes, possibly on a number of different hosts, connected at runtime in a peer-to-peer topology. The peer-to-peer topology requires a lookup-mechanism to allow processes to find each other at runtime. ROS call this, the name master. If processes are running on different hosts, only one host is required to start the master process.

Finally ROS implements several different styles of communication between processes, including synchronous RPC (Remote Procedure Calls)-style communication over *services*, asynchronous streaming of data over *topics* and storage of data on a *Parameter Server*.

2.3.2 ROS architecture

The fundamental concepts of the ROS implementation are *nodes*, *messages*, *topics*, and *services*.

The computation in ROS is done by using a network of process called ROS nodes. This computation network is called the computation graph (Fig. 2.6). The main concepts in the computation graph are Nodes, Master, Parameter Server, Messages, Topics, Services, and Bags. Each one of these, has its own role in the computation graph.

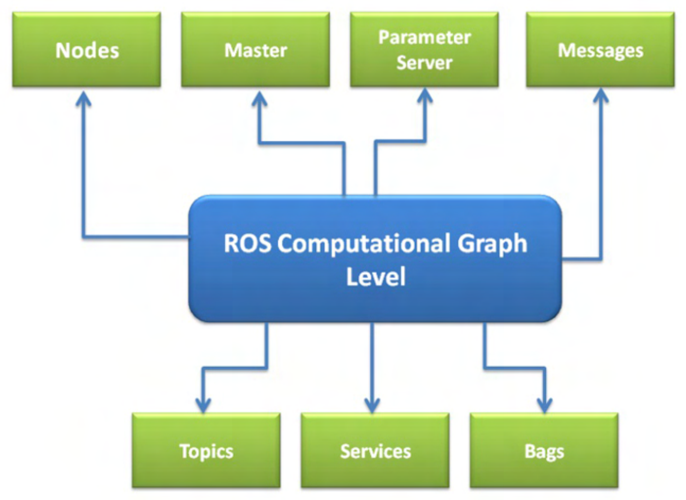


Figure 2.6: Structure of the ROS Graph layer.

Nodes

Nodes are processes that perform computation. There is a peer-to-peer communication between nodes. The name node arises from the fact that when multiple nodes are running, it is easy to translate the peer-to-peer communication as a graph, the processes as graph nodes and the peer-to-peer links as arrows between them.

A basic example of a robot is that a robot control system comprises many processes (nodes). In other words, one node may control a laser, one node may control the wheel motors, one node may control the localization e.t.c.

Finally, a ROS node is written with the use of a ROS client library (roscpp, rospy).

Master

The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, trade messages, or apply services.

Parameter Server

The Parameter Server runs inside the ROS Master and is a shared, multi-variate dictionary. Nodes can use this server in order to retrieve or store parameters at runtime.

Messages and Topics

A message is a a strictly typed data structure. Also, messages can be composed of other messages, and arrays of other messages. A node sends a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic (Fig. 2.7), and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence.

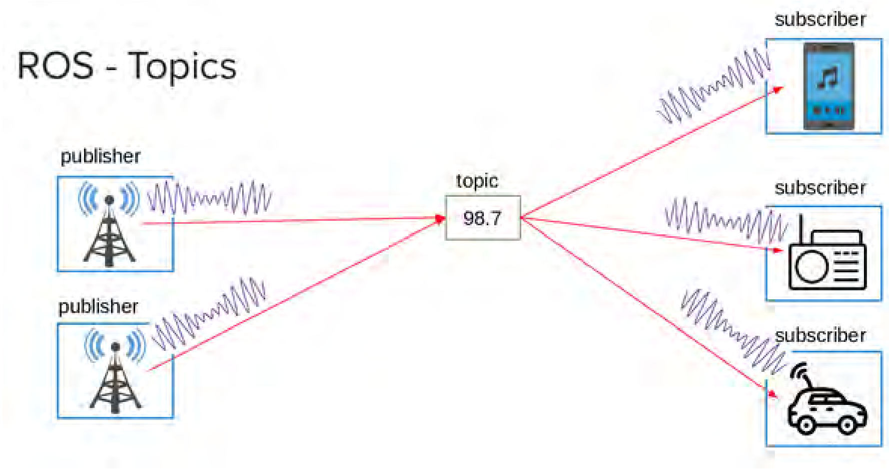


Figure 2.7: An example of ROS Topic.

Services

Although messages are the primary method in order to communicate in ROS, there are a few limitations. For this purpose, ROS introduce another way of communication, services. The differences between messages and services are:

- Service calls are bi-directional. One node sends information to another node and waits for a response. That is to say, both directions share the same information. On the other hand, when a message is published, there is no concept of a response, and no one guarantee that any node is subscribing to those messages.

- Service calls implement one-to-one communication. Each service call is initiated by one node, and the response goes back to the same node. On the other hand, each message (as mentioned before) is related to a topic that might have plenty of publishers and subscribers.

Services are defined by a pair of message structures: one for the request and one for the reply. A provider node offers a service under a specific name and a client uses this service by sending the request message and waiting the reply message. See Fig. 2.8.

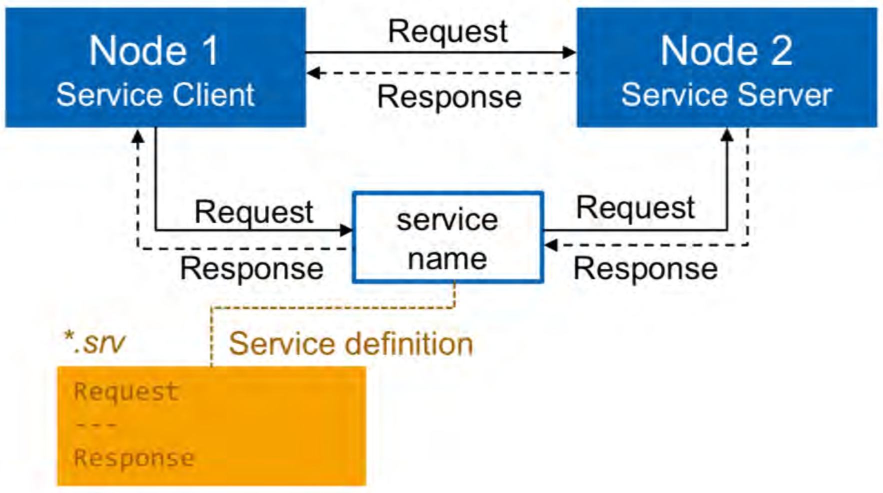


Figure 2.8: An example of ROS Service.

Chapter 3

Conceptual Approach

This chapter, firstly, gives an overview about the pinhole camera model (i.e how does a 3-dimensional world point is projected onto an image plane) and the fundamental parameters of a camera, in order to understand the importance of the calibration method. Then, it illustrates the two proposed methods that the quadcopter follows in order to approach the visual marker, by explaining the meaning of rotations and translations in the 3-dimensional space, as well as, the meaning of the navigation angles.

3.1 Perspective Projection - Transformation

Before we continue talking about camera calibration, we must first, make an overview of perspective projection. In other words, how we are mapping three-dimensional points that a camera sees in the scene to a two-dimensional plane (screen - image plane).

3.1.1 Pinhole Camera Model

Pinhole camera is an ideal camera with a small aperture (Fig. 3.1) and without lens, therefore geometric distortions or blurring of unfocused objects are not included. The model of a pinhole camera describes the relationship between the $3D$ coordinates of a point in space and its projection onto the image plane. Also, notice that the pinhole camera model does not take into account the coordinate transformations from one reference system to another, for example a point transformation from a world coordinate system to a camera coordinate system. Thus, this model can only be used as a first order approximation of the mapping from a $3D$ scene to a $2D$ image.

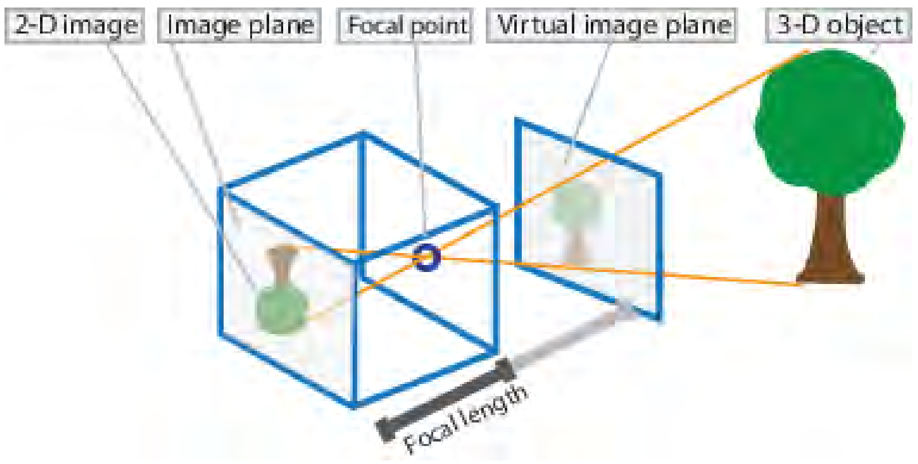


Figure 3.1: Pinhole Camera Model. One can think of the virtual image plane as being in front of the camera and hold the upright image of the scene.

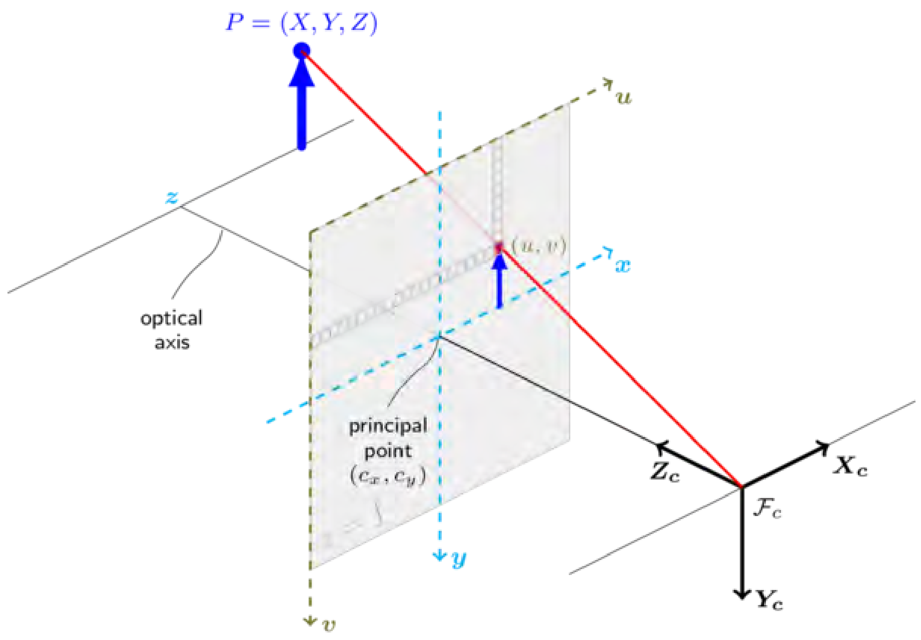


Figure 3.2: The mathematical model and the geometry of a pinhole camera.

Figure 3.2 shows the $3D$ coordinate system (Xc, Yc, Zc) of a camera with its Fc as origin (where the small aperture of the pinhole camera is located). The Zc axis is referred as the *optical axis*, *principal ray* or *principal axis*. Also the origin of the camera Fc is a point in the $3D$ space which is referred as the $opticalcenter$, lens center or camera center.

It is important to mention that in reality the image plane is located behind the $3D$ orthogonal system of the camera. However, it is easier for someone to see the connection between the coordinate systems if the image plane is reflected, so that it is located in front.

Moreover, the Fig. 3.2 contains the following basic objects:

- An image plane where the $3D$ point is projected and is located at a distance f from the origin Fc, called $focallength$.
- The *principal point* - image center, which is the intersection of the *optical axis* and the image plane.
- A point P with coordinates (X, Y, Z), which is located in the world.
- The *projection line* (red line in the Fig. 3.2) of the point P and the origin of the camera Fc.
- The projection of the point P onto the image plane, which is the intersection of the *projection line* (red line) and the image plane.

Lastly, the image plane has its own coordinate system, with its origin at the *principal point* and the axes x and y parallel to X_c and Y_c, respectively.

To associate the coordinates (x, y) in the image plane with those (X, Y, Z) of a point P in the scene is done via similar triangles and we ended up with the following equations.

$$x = f\frac{X}{Z} \tag{3.1.1}$$

$$y = f\frac{Y}{Z} \tag{3.1.2}$$

3.1.2 Camera Movement

Most of the time, cameras have to capture images from different viewpoints, such as in this thesis; remember that the camera is mounted on the drone. Therefore, we need to have a way of modeling camera movements. A camera can apply rotations and translations. A translation is represented by the vector $t \in \mathbb{R}^3$ and it means that a camera can move from its current $3D$ location (X, Y, Z) to a new $3D$ location (X', Y', Z'). The rotation is represented by a 3 x 3 matrix R and it means that a camera can rotate about the X, Y and Z axes. Also, in order for a matrix R to represent a rotation has to fulfill two requirements:

$$\det(R) = 1 \tag{3.1.3}$$

$$R^{-1}R = I, \text{ where } I \text{ corresponds to the identical matrix.} \tag{3.1.4}$$

Lastly, since R represents the rotation of a three orthogonal axes from a specific coordinate system, is an orthogonal. Thus, it follows that:

$$R^T = R^{-1} \tag{3.1.5}$$

$$R^{-1}R = R^T R = I \tag{3.1.6}$$

The Fig. 3.3 shows the two coordinate systems, world and camera. A world coordinate system can be anything, since the camera movements will be related to a specific coordinate system that we will define.

Now, let's suppose a scene point P with coordinates (X, Y, Z) in the world coordinate system and (X', Y', Z') in the camera coordinate system. These two different coordinates of the same point P can be related through:

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \tag{3.1.7}$$

Thus, a rotation matrix R and a translation vector t relates the two different coordinate frames.

The points of the equation 3.1.7 can be written in the form of homogeneous coordinates (see subsection 3.1.3). Thus, in a matrix form we end up:

$$Z^{'}\begin{bmatrix} X^{'}/Z^{'} \\ Y^{'}/Z^{'} \\ 1 \end{bmatrix} = \begin{bmatrix} X^{'} \\ Y^{'} \\ Z^{'} \end{bmatrix} = \begin{bmatrix} R & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.1.8}$$

The $\begin{bmatrix} R & t \end{bmatrix}$ is a 3 x 4 matrix, where the first three rows and columns are constitute the rotation matrix R and the last column is the translation vector t. In the subsection 3.1.1, we ended up into two equations 3.1.1, 3.1.2, which represent the projection of the point P onto the image plane (assume that the focal length is 1). Thus, the equation 3.1.8 computes the projection of the scene point P, but with the coordinates given in the world reference system rather than in the camera's one. Moreover, this equation shows that, in order to obtain the projected point onto the image plane, a vector $a = \begin{bmatrix} X^{'} & Y^{'} & Z^{'} \end{bmatrix}^T = \begin{bmatrix} R & t \end{bmatrix} \begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^T$ must be computed followed by a division of its elements by the third coordinate.

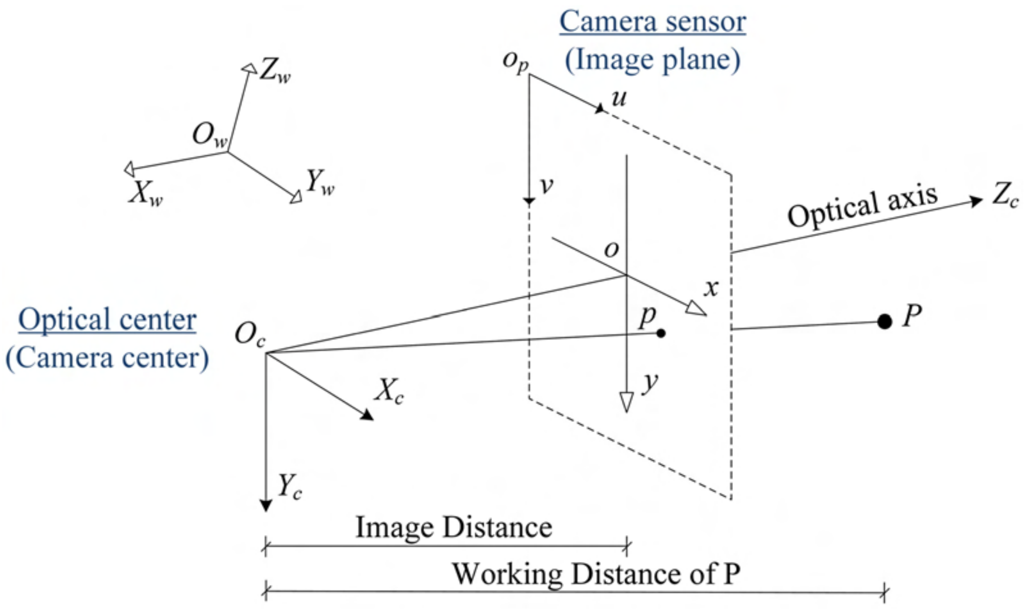


Figure 3.3: The correlation between the Camera Coordinate System (Bottom - Left) and the World Coordinate System (Top - Left).

3.1.3 Homogeneous Coordinates

Most of the time, in $3D$ computer vision or computer graphics, is much simpler to represent coordinates in the form of *homogeneous coordinates*. Homogeneous coordinates (or projective coordinates) are the system used in projective geometry, as the cartesian coordinates used in euclidian geometry. Moreover, homogeneous coordinates take an advantage over its opponent, since they include points at infinity. To summarize, they are always applied in computer graphics, since they allow vector operations such as translation, rotation and scaling to be combined in a form of a matrix.

To represent cartesian coordinates in the form of homogeneous coordinates, an extra dimension must be added. Thus, for a given point $2D$ (x, y) on the euclidean plane, the triple (wx, wy, w) is called a set of homogeneous coordinates of that point for any non-zero real number w. By this definition, multiplying this triple of homogeneous coordinates by a common, non-zero factor, gives a new set of homogeneous coordinates for the same point. Particularly, $(x, y, 1)$ is such a system of homogeneous coordinates for the point (x, y). For example, the point $(16, 2)$ of the cartesian coordinates can be represented by the homogeneous coordinates as $(16, 2, 1)$ or $(32, 4, 2)$. In this example, the point $(16, 2)$ was multiplied by the non-zero factor 2. Lastly, to recover the original point in the cartesian geometry, the coordinates of the homogeneous point must be divided by its third - last element. Therefore, in the previous example, the homogeneous point $(32, 4, 2)$ is divided by its third element, i.e 2, in order to recover the point $(16, 2)$ in the cartesian geometry.

All of the above definition can be easily expanded to projective spaces greater than the $2D$ image projective plane. Thus, for example, a $3D$ point (X, Y, Z) can be represented in the cartesian coordinates as a (WX, WY, WZ, W) in homogeneous coordinates, where in the simplest situation is $(X, Y, Z, 1)$.

3.1.4 Camera Matrix

The Camera matrix is given by $C = K \begin{bmatrix} R & t \end{bmatrix}$, where matrix K includes the intrinsic parameters and matrix $\begin{bmatrix} R & t \end{bmatrix}$ includes the extrinsic parameters.

Extrinsic Camera Parameters

So far, we have seen how the pinhole camera model (subsection 3.1.1) represents the projection of a $3D$ point in the scene into a $2D$ point in the image plane. Also, in the subsection 3.1.2, we have seen how to relate the world coordinates to camera coordinates and then, finally, to the image plane coordinates, when a camera movement take place. Moreover, in order to apply the rotation and translation in a matrix (3 x 4) form, rather than separately as the equation 3.1.7 does, we changed the euclidian world coordinates of a $3D$ point P into the homogeneous coordinates 4D point. This is why, rotation only require three columns to be applied, but in order to do a translation, the matrix requires one more column to be added. So, we end up with a four-column matrix. Thus, to multiple a four-column matrix with a point P in the scene, is obligated to have a four-element vector (homogeneous coordinates) rather than a three-element vector (cartesian coordinates).

The above 3 x 4 matrix $\begin{bmatrix} R & t \end{bmatrix}$ represents the *extrinsic parameters* of the camera matrix. The more analytically representation of this matrix is:

$$\begin{bmatrix} R & t \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \tag{3.1.9}$$

To summarize extrinsic parameters are the parameters that define the location and orientation of the camera reference frame with respect to a known target reference frame. So, once we multiple a point P in the target reference frame, assume a word reference frame, with the matrix $\begin{bmatrix} R & t \end{bmatrix}$, we end up with the same point P but now expressed in the camera reference frame. Fig. 3.4 shows an illustration of those coordinate systems.

Intrinsic Camera Parameters

In the subsection 3.1.1 as we discussed the pinhole camera model, this model assumes that image plane coordinates $\epsilon\ \mathbb{R}^3$. That is to say, image projections are given in the length unit (e.g meters) of $\mathbb{R}^3$. Furthermore, the center of the image is located, in homogeneous coordinates at $(0, 0, 1)$, so the image coordinates are $(0, 0)$ (the point

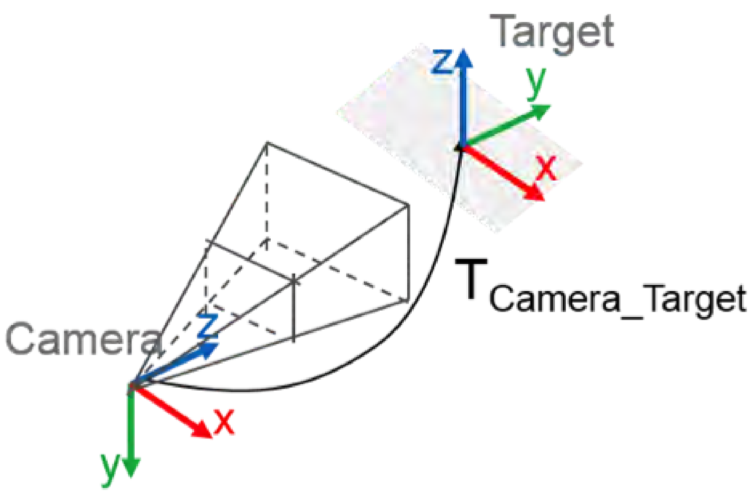


Figure 3.4: Definition of the T_{Camera_Target} transformation that defines the location and orientation of the camera coordinate system with respect to the known target coordinate system.

O in Fig. 3.3).

On the other hand, real cameras capture images which are measured in pixels with the $(0, 0)$ in the upper left corner (point O_p in Fig. 3.3). Consequently, to be able to do geometrically meaningful computations we need to transform from image plane coordinates to pixel coordinates.

As extrinsic parameters are formed by a matrix $\begin{bmatrix} R & t \end{bmatrix}$, intrinsic parameters can be formed by a 3 x 3 triangular matrix K. This matrix 3.1.10 contains the *focal length* in pixels ($f_x = f/p_x$, $f_y = f/p_y$, where f is the focal length in word units, and p_x, p_y is the size of the pixel in word units) and the *principal point* (c_x, c_y) and as we mentioned before, transforms the image plane to image coordinates.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1.10}$$

So, in this case, K matrix transforms the point in the image plane coordinates to pixel coordinates according to:

$$\begin{bmatrix} f_x x + c_x \\ f_y y + c_y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3.1.11}$$

In other words, the coordinates are scaled by the focal length and translated by the principal point.

Combining extrinsic and intrinsic camera parameters we end up with the camera matrix C that projects $3D$ points in space onto the camera sensor.

$$C = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \tag{3.1.12}$$

3.2 Camera Calibration

Camera calibration is the process that estimates the parameters of a camera lens and image sensor. These parameters, as mentioned before, are called fundamental parameters of a camera and are categorized into intrinsics and extrinsics. These two categories are included into the camera matrix C (see subsection 3.1.4). One can use these parameters in order to correct lens distortion, determine the location of the camera in the scene, e.t.c. Fig. 3.5 shows an example of how the image is presented before and after the correction of the lens distortion.

Moreover, the matrix of intrinsic parameters K, does not depend on the scene viewed. So, once estimated, it can be re-used as long as the focal length is fixed. By the phrase "focal length is fixed", we mean that there is not any zoom lens, so the parameters f_x, f_x are always the same. In this thesis, we assume that there is not any zoom lens, so once the camera is calibrated the K matrix can be used for the rest of the process.

3.2.1 Geometrical Distortion

Another important thing that a camera calibration take into account, is the lens distortion [8]. That is because the camera matrix does not account for lens distor-

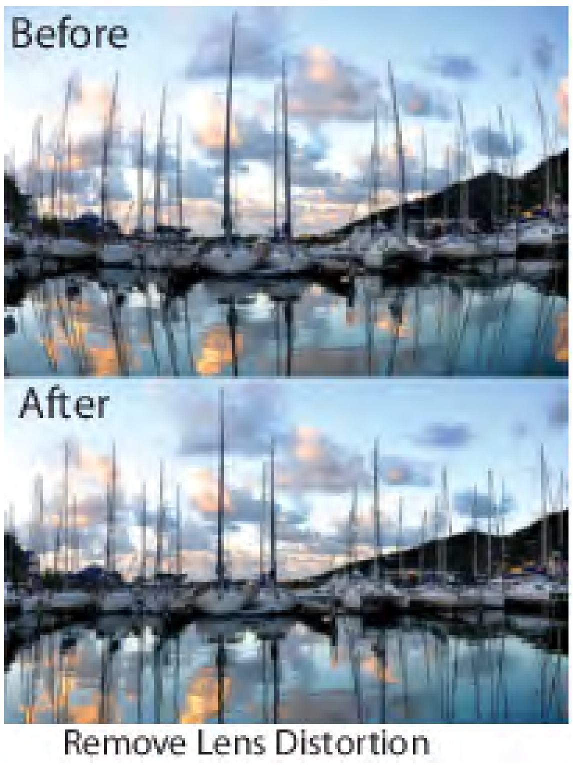


Figure 3.5: Upper image before the camera calibration procedure. Here, straight lines are seen curved because of distortion of the lens. Down image after the camera calibration procedure and correction of the lens distortion.

tion, since an ideal pinhole camera does not have any lens. To accurately represent a real camera, the camera model includes the *radial* and *tangential* lens distortion.

Radial Distortion

Radial distortion causes an inward or outward displacement of a given image point from its ideal location. There are two subcategories of a radial distortion, negative and positive. A negative radial displacement of the image points is referred to as *barrel distortion* and it causes points to crowd increasingly together and the scale to decrease. A positive radial displacement is referred to as *pincushion distortion* and it causes points to spread and the scale to increase. Fig. 3.6 illustrates the effect of radial distortion.

The distorted points $(x_distorted, y_distorted)$ are denoted as:

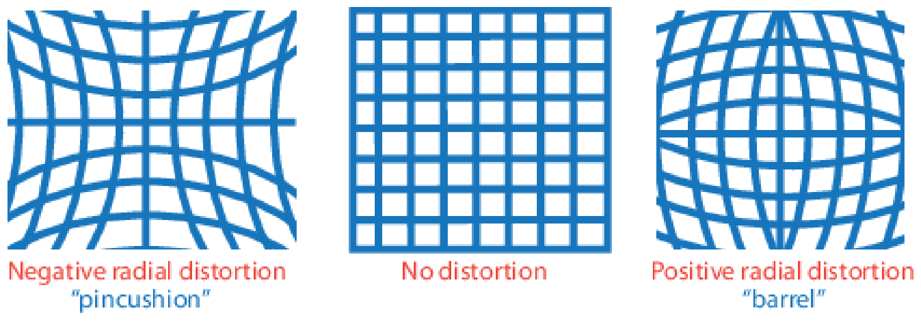


Figure 3.6: Left image with positive distortion. Right image with negative distortion. In the middle there is not any distortion.

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{3.2.13}$$

$$y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{3.2.14}$$

where: *a)* x, y are the coordinates of the projection of a $3D$ point in camera coordinates, as found in eq. 3.1.8, *b)* k_1, k_2, k_3 are the radial distortion coefficients of the lens, and *c)* r^2 is equal to $x^2 + y^2$.

Tangential distortion

Tangential distortion occurs when the lens and the image plane are not parallel. Fig. 3.7 illustrates the effect of radial distortion.

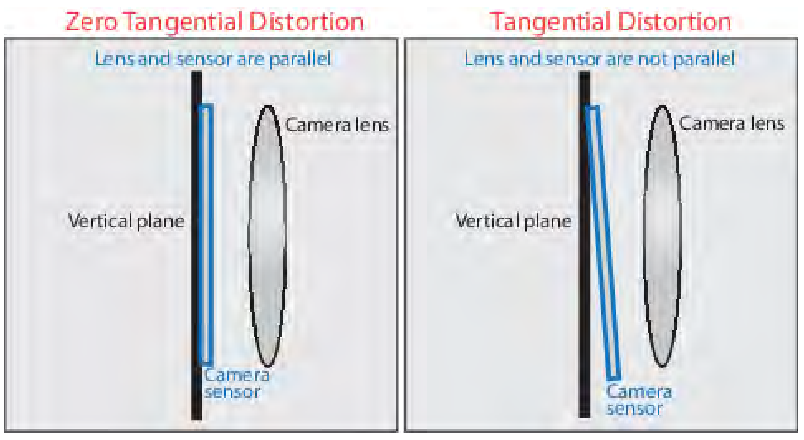


Figure 3.7: Left image with no distortion. Right image with tangential distortion as the image plane and lens are not parallel.

The distorted points $(x_distorted, y_distorted)$ are denoted as:

$$x_{distorted} = x + [2p_1xy + p_2(r^2 + 2x^2)] \tag{3.2.15}$$

$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2xy] \tag{3.2.16}$$

where: *a)* x, y as mentioned before, *b)* p_1, p_2 are the tangential distortion coefficients of the lens, and *c)* r^2 as mentioned before.

If we add radial and tangential distortion together for the x, from the equations 3.2.13 and 3.2.15 we end up with $x_{distorted}$:

$$x_{distorted} = x(1 + k_1r^2 + k_2r^4 + k_3r^6) + [2p_1xy + p_2(r^2 + 2x^2)] \tag{3.2.17}$$

Similarly for $y_{distorted}$:

$$y_{distorted} = y(1 + k_1r^2 + k_2r^4 + k_3r^6) + [p_1(r^2 + 2y^2) + 2p_2xy] \tag{3.2.18}$$

Lastly, in contrast to an ideal camera model where projected image coordinates x, y extract the u, v pixel coordinates (as subsection 3.1.4 mentions), now we use the $x_{distorted}, y_{distorted}$ in order to extract them. More specifically:

$$u = f_x x_{distorted} + c_x \tag{3.2.19}$$

$$v = f_y y_{distorted} + c_y \tag{3.2.20}$$

The distortion coefficients do not depend, like the intrinsic parameters, on the scene viewed. Therefore, they are also called intrinsic parameters and remain the same regardless of the captured image resolution. For example, if the calibration was done on images with resolution of 640 x 480, the distortion coefficients remain the same on resolution of 1024 x 768. On the contrary, focal length f_x, f_y and principal point c_x, c_y need to be scaled appropriately.

3.2.2 Calibration Process

The calibration process can be done via different ways and is trying to extract the fundamental parameter of the camera. This can be done by a set of $3D$ to $2D$

correspondences. In this thesis, the calibration process of the camera is done with the help of the OpenCV library [9], [10], [11]. More analytically, this can be done within an image sequence of a square chessboard (Fig. 3.8) that is captured from different poses.

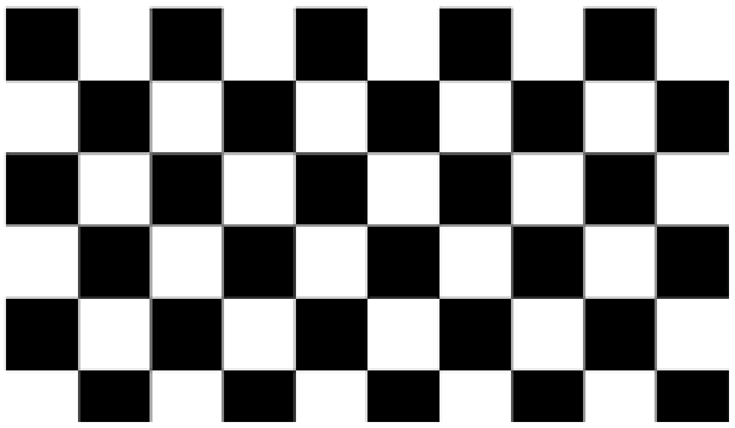


Figure 3.8: A square chessboard used to do the calibration process.

Therefore, the important input data needed for a camera calibration is a set of $3D$ real world points and its correspondences in $2D$ image plane. The points of interest are the locations where two black squares touch each other in the chessboard.

The $2D$ image points can be found every time from the current captured image, with the help of the openCV function - *findChessboardCorners()*.

On the other hand, in order to find the $3D$ (X, Y, Z) points we need to make one simplicity. A chessboard is a plane, so the Z-dimension is always zero $(Z = 0)$. Thus, only the (X, Y) elements are required to specify. Moreover, we set a world coordinate system with the origin at the top left corner of the chessboard. In this way, the first object point of interest is a corner with coordinates $(0, 0)$, the second $(0, square\_size)$ and so on. This is a collection of the points where these important points are present. The *square_size* one can measure it from the time is in printed form (in a paper).

Once we find the $3D$ object points and $2D$ image points we can use the openCV function - *calibrateCamera()*, which takes input the $3D$ and $2D$ correspondences and returns the intrinsic, extrinsic parameters of the camera matrix and the distortion coefficients.

Note that we are not going into much details about the calibration process. For more information see [12], [13], [14].

3.3 Rotations

At this point, since we have already spoken about the importance of the rotation matrix in the subsection 3.1.2, the next step is to find a way to determine a method, which it represents the rotation of a body as expressed in a given coordinate system.

There are different ways to specify and perform a rotation in $3D$ of a body frame [15], like. *a)* Matrices, *b)* Euler angles, *c)* Quaternions, *d)* Axis-angle. There are positives and negatives in each one of the different rotations.

To be more specific, axis-angle and Euler angles representations can be comprehend more easily but they suffer from significant problems. For example, Euler angles, have the *Gimbal Lock* problem. Axis-angle (as euler angles) representation suffers when they are applying two rotations in a row, since it is not valid to say that the total rotation is the sum of the individual rotations.

Matrix representation have the advantage of defining both angular and linear motion (i.e both rotation and translation), so they used most in kinematics, where they need to be done both and do not keep transferring between mathematical notations. Moreover, a matrix multiplication lets the addition of n rotations (i.e to be combined) in order to extract the total rotation of a body.

Lastly, quaternions can represent operations in $3D$ space such as a complex number represents operations in $2D$ space. One of these operations in $3D$ space, is a rotation. Moreover, it takes an advantage over the euler angles, since it is not suffering from the instabilities that associated with the euler angles or axis-angle representations.

The table 3.1 gives an illustration about different orientation presentations.

3.3.1 Axis angle

Axis angle, as defined before, is one of the many ways we can use to represent the rotation of an object in 3 dimensional world (Fig. 3.9). More specifically, this

	No. Parameters	Concatenate Rotations by Multiplication	Def. both Rotation and Translation
Axis-Angle	4	no	no
Euler angles	3	no	no
Quaternions	4	yes	yes, dual quaternions
3x3 Matrix	9	yes	yes, 4x4 Matrix with 16 param

Table 3.1: An illustration of different orientation presentations.

rotation is comprised of a unit vector, that indicates the direction of an axis of rotation, and an angle, that describes the magnitude of the rotation about that axis. The openCV library always works with the axis-angle notation and returns a $3D$ *rvec* rotation vector from which we can extract the *unit vector* and the *angle* as follows:

$$angle = \sqrt{rvec_1^2 + rvec_2^2 + rvec_3^2} \tag{3.3.21}$$

$$axis = rvec/theta \tag{3.3.22}$$

The basic flaw of axis angle representation is that we can not combine two rotations to give an equivalent total rotation. Fortunately, it is easy for someone to convert between axis angle and matrix rotation or quaternions. Lastly, is given an example of this representation.

Assume we want to rotate an object 90 deg about the Z axis in $3D$ world with X, Y and Z axes coordinate system then the axis angle would be:

$$\begin{bmatrix} axis \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, angle = \tfrac{\pi}{2}$$

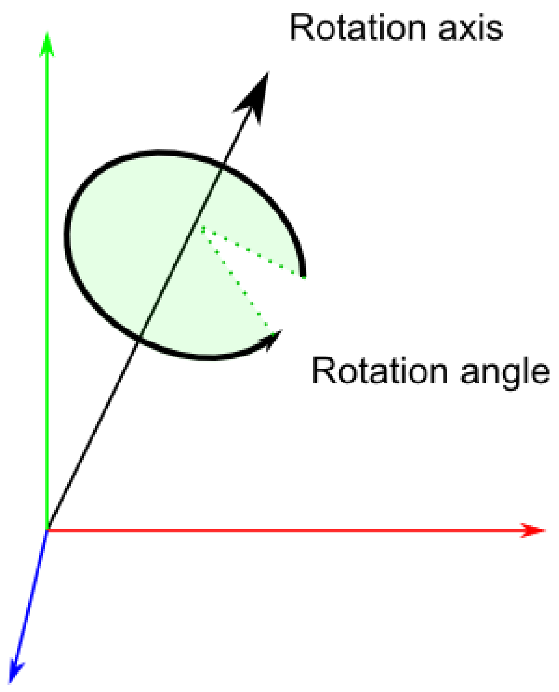


Figure 3.9: Axis angle rotation about a unit vector and an angle.

3.3.2 Matrices

In order to represent $3D$ rotations with 3x3 matrices, a matrix must be an *orthogonal.* This means that: *a)* the determinant is +1, *b)* the transpose is equal to the inverse matrix, $M^{-1} = M^T$ and *c)* the construction of the matrix M is done by a set of mutually perpendicular basis vectors (e.g a $3D$ coordinate system). Since basis vector are perpendicular to each other, the dot product of any two basis vectors is zero. Also, the basis vectors must have unit length, therefore are unit vectors. Finally, if we know two basis vector, we can derive the other one by using the cross product.

An illustration of an orthogonal matrix that represents a rotation is:

$$R_{total} = \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} \tag{3.3.23}$$

The column vectors $\begin{bmatrix} r_{00} & r_{10} & r_{20} \end{bmatrix}^T$, $\begin{bmatrix} r_{01} & r_{11} & r_{21} \end{bmatrix}^T$, $\begin{bmatrix} r_{02} & r_{12} & r_{22} \end{bmatrix}^T$ are the *basis* vectors of a coordinate system, which each one of them rotates a specific axis about an angle θ.

Finally, the direction of each rotation is given by the right hand rule, where the thumb is in the direction of the axis, while the fingers show the positive direction.

An example of how to rotate about an angle θ on each axis (Fig. 3.10) is shown below:

Rotation about X axis

This rotation is given by a matrix $R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$

Rotation about Y axis

This rotation is given by a matrix $R_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$

Rotation about Z axis

This rotation is given by a matrix $R_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$

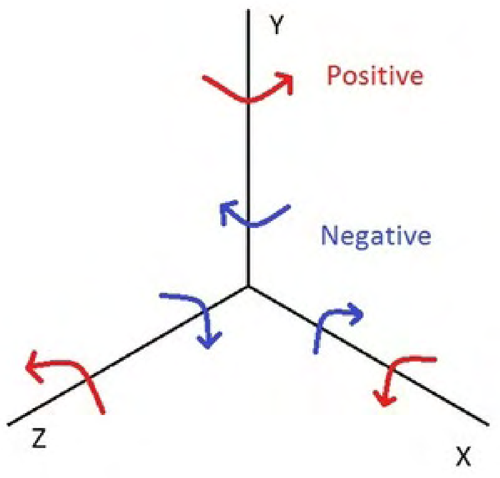


Figure 3.10: Rotation along each axis.

If we want to do successive rotation then we must multiple the matrices R_x, R_y and R_z representing the individual rotations. The order of rotations that we multiple is a great importance. This can be seen by the following example:

Firstly, assume that the order of successive rotations are: *a)* 90 degrees about the x axis, *b)* 90 degrees about the y axis, and *c)* -90 degrees about the x axis. Then, the final rotation is $R_{total} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.
In other words, this outputs a 90 degree rotation about the Z axis.

But, if we assume that the order of successive rotations are: *a)* 90 degrees about the x axis, *b)* -90 degrees about the x axis, and *c)* 90 degrees about the y axis. This outputs a 90 rotation about the Y axis (calculations are done similarly).

3.3.3 Euler Angles

Euler Angles represents the three dimensional rotation of a body as expressed in a given coordinate system. Moreover, they are three successive rotations relative to the three axes of the coordinate system. So, there are three angles of rotation, assume the angles θ, ϕ and ψ. These three rotations can be categorized into *extrinsics* and *intrinsics*.

Extrinsics rotations

Are rotations about the axes xyz of the original coordinate system, which assumed to remain motionless. For example, assume we want to apply three rotations, first to the x axis, second to the y axis and third to the z axis. With the first rotation, we end up with a new coordinate system with the axes as $xy'z'$. In other words, the two axes y, z are transformed to a new y' and new z' axes but the x remains the same. The second rotation, however, is applied in the first coordinate system xyz and not in the new rotated $xy'z'$ system. Similarly is done with the third rotation.

Intrinsics rotations

Are rotations about the axes of the rotating coordinate system xyz, solidary with the moving body, which changes its orientation after each rotation. In this thesis, we make use of the intrinsic rotations.

There are twelve possible sequences of rotation axes (i.e the order in which multiplication will take place in order to build the rotation matrix, see subsection 3.3.2). This is very important, since we need to define which rotations applied 1_{st}, 2_{nd} and 3_{rd}, so we can extract the euler angles from the rotation matrix. The sequences are: *a) Proper Euler angles* (z-x-z, x-y-x, y-z-y, z-y-z, x-z-x, y-x-y) and *b) Tait–Bryan angles* (x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z).

Also, notice that, the three angles θ, ϕ and ψ, are also called *yaw*, *pitch* and *roll* angles. The illustration of what exactly these angles mean is given in the figure 3.11. In this particular figure, an aircraft can rotate in all of the three dimensions. More specifically, it can rotate its nose to the left or to the right, about an axis that is running from up and down of the aircraft. This is called yaw. Moreover, it can rotate its nose up or down, about an axis that is running from wing to wing. This is called pitch. Finally, it can move up or down its wings about an axis that is running from the nose to tail, this is call roll. This thesis makes the assumption that the yaw angle is applied first, the pitch angle is applied second, and the roll angle is applied third. This assumption plays a major role in order to find the solution of the problem. To be more specific, since this thesis, already, defines in which order these angles are applied, it remains the matter which one of these angles rotates each one of the axes x, y and z.

3.4 GPS Coordinates

The aim of this section is to show how to find the GPS coordinates of the visual marker. More specifically, in order to find the latitude and longitude of the marker we make use of the haversine formula. This formula is:

$$latitude = \phi_2 = \arcsin\left(\sin\phi_1 \cos\delta + \cos\phi_1 \sin\delta \cos\theta\right) \tag{3.4.24}$$

$$longitude = \lambda_2 = \lambda_1 + \text{atan2}(\sin\theta \sin\delta \cos\phi_1, \cos\delta - \sin\phi_1 \sin\phi_2) \tag{3.4.25}$$

, where ϕ_1, λ_1 is the latitude, longitude of the quadcopter respectively, θ is the bearing clockwise from North, δ is the angular distance $\frac{d}{R}$; d being the distance between quadcopter and visual marker, R the earth's radius.

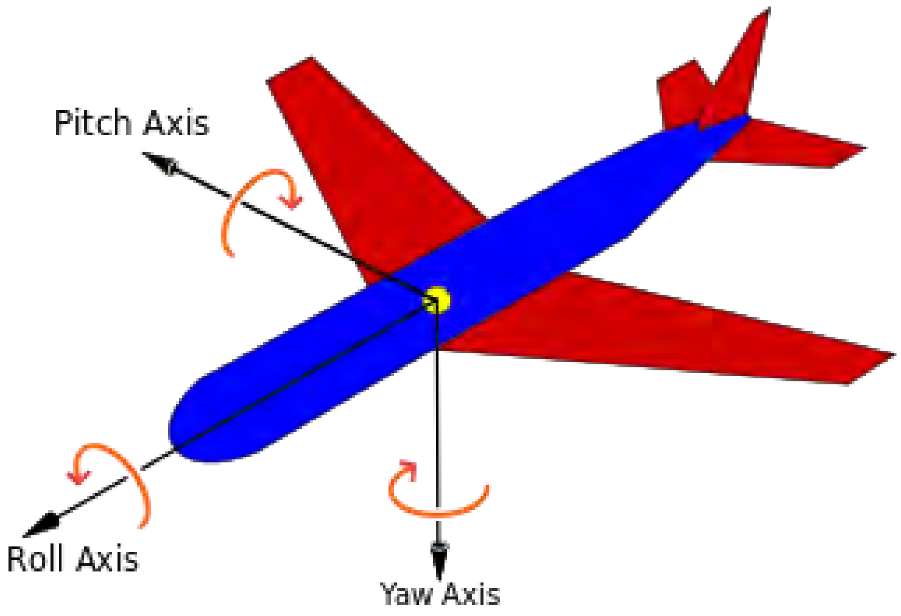


Figure 3.11: Shows the yaw, pitch and roll angles in a vehicle and how they rotate the aircraft. Notice that is uses the right-hand rule to determine the positive and negative rotation.

3.4.1 Navigation Angles

In order to understand what is the *bearing* and *heading* angles, so as to use them in guiding the quadcopter with GPS coordinates, we give the following definitions and an example.

Heading

Angle of where the vehicle's nose points. In other words, this is where the head of the vehicle is pointing relative to North.

Course

Angle that represents the intended path of travel that have calculated taking into account winds and variation.

Track

Angle that represents the actual path traveled over ground relative to North.

Bearing

This is the angle between the location of an object (destination) and either:

- Current heading. This is called *Relative Bearing*.
- Magnetic north (direction toward the magnetic north pole). This is called *Magnetic Bearing*.

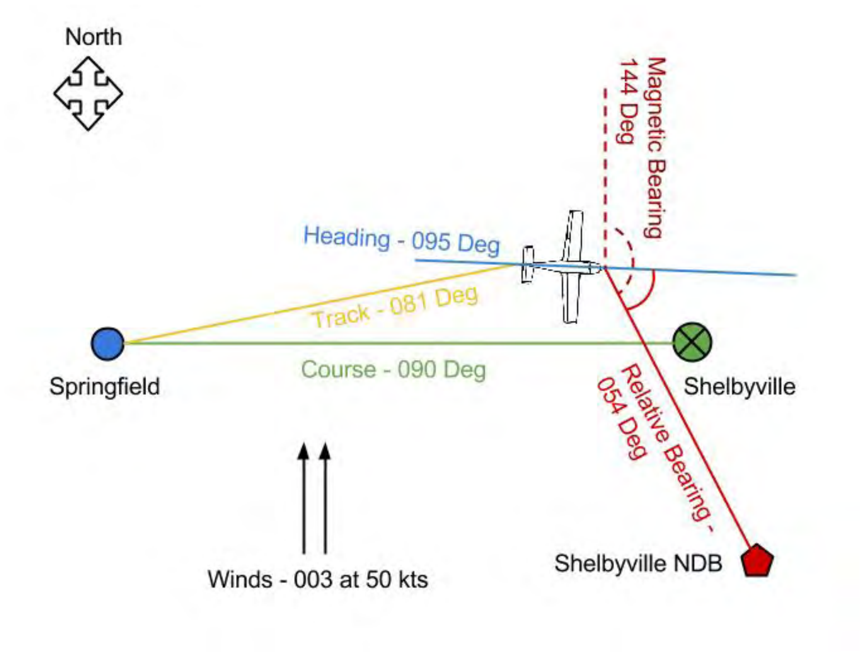


Figure 3.12: Representation of heading, course, track and bearing angles.

From the figure 3.12 we can easily see, that if we take off from Springfield on the way to Shelbyville, the *course* (the intended path) is 90 degrees. But due to the winds we make the vehicle's *heading* 95 degrees so as to compensate for wind drift. Normally, the course and track are the same. However, due to misjudging the winds, the *track* over the ground is 81 degrees and not 90 degrees. Therefore, we should correct the heading to get back to course. This can be done if we take the *bearing* to the Shelbyville NDB either relative to current heading or relative to magnetic North to find its position.

3.5 Proposed Approaching Methods

This section illustrates the two approaching methods, that are proposed by the thesis, in order to navigate the quadcopter and approach the visual marker. The details of these methods are described in the implementation chapter 4.

3.5.1 Pose Extraction Method

This method makes use of the camera pose estimation relative to the visual marker. More specifically, since this thesis uses the ArUco library in order to detect the visual marker, the ArUco returns the rotation and translation vectors that represent the pose of the camera with respect to the visual marker. In others words, ArUco outputs the transformation between the camera and the visual marker.

However, the goal is to find the pose of the quadcopter relative to the visual marker. Therefore, this method does an extra transformation between the camera's reference frame and the quadcopter's one. In this way, this method extracts the rotation and translation vectors that the quadcopter uses to localize itself and approach the marker.

Finally, the method manipulates these vectors as follows: *a)* It uses the translation vector (dx, dy, dz), which represents the displacement that the quadcopter has to do in its three body axes (X, Y, Z) respectively, in order to direct navigate the vehicle to approach the target. *b)* It extracts the euler angles from the rotation vector, that represents the rotation that the quadcopter has to do in its three body axes (X, Y, Z), in order to align the quadcopter's coordinate system with the visual marker's one. Notice that, since the visual marker lies on the ground (therefore it has not any inclines) only a yaw rotation has to be applied to the vehicle so that the axes of the quadcopter can be aligned with the marker's one.

3.5.2 GPS Navigation Method

This method makes use of the formulas 3.4.24, 3.4.25 described in the subsection 3.4. For this purpose, this method needs to find the GPS position of the quadcopter, the magnetic bearing of the marker relative to North and the ground distance between the quadcopter and the marker. Each of these elements are founded as follows. *a)* The GPS position of the quadcopter is founded via the autopilot and it is used as the initial point in the above formulas. *b)* The magnetic bearing of the visual marker is founded by adding the relative bearing (of the quadcopter and the marker) with the current heading of the quadcopter. The relative bearing is the yaw angle that is extracting from the rotation of the quadcopter with respect to the visual marker. The current heading, similarly to the vehicle's GPS position, is founded via the autopilot and it describes the heading of the quadcopter relative to the North. *c)* The ground distance is founded via the translation vector (this vector is extracted by the pose of the quadcopter relative to the marker, as explained in the previous method).

Chapter 4

Implementation

This chapter discusses the components that have been implemented, so as the UAV (unmanned aerial vehicle) is able to reach the target - visual marker. Moreover, a bigger picture from a software engineering perspective is shown, so that the reader understands how the different components are combined. Finally, it explains the steps that are followed in order to implement the detection of the visual marker and the guidance of the quadcopter so as to approach the target.

4.1 Combining existing parts

4.1.1 Illustration of the overall System Architecture

It is of high importance, before going into much details, to show the overall illustration of the whole system architecture. In figure 4.1, we see that we use Gazebo as the Physics Simulator on our system. Gazebo provides us the visualization of a virtual world that represents the model files of the quadcopter, the gimbal and the camera which is attached upon the gimbal. These models are controlled by the plugins that are located inside the Ardupilot_SITL_Gazebo_plugin repository. Model plugins, as explained in the subsection 2.2.1, control joints and state of the models (here, quadcopter and gimbal). Moreover, there are plugins, inside this repository, that enable the communication between the Physics Simulator Gazebo and the ArduPilot code. Notice that, since we use the SITL ArduPilot code, SITL

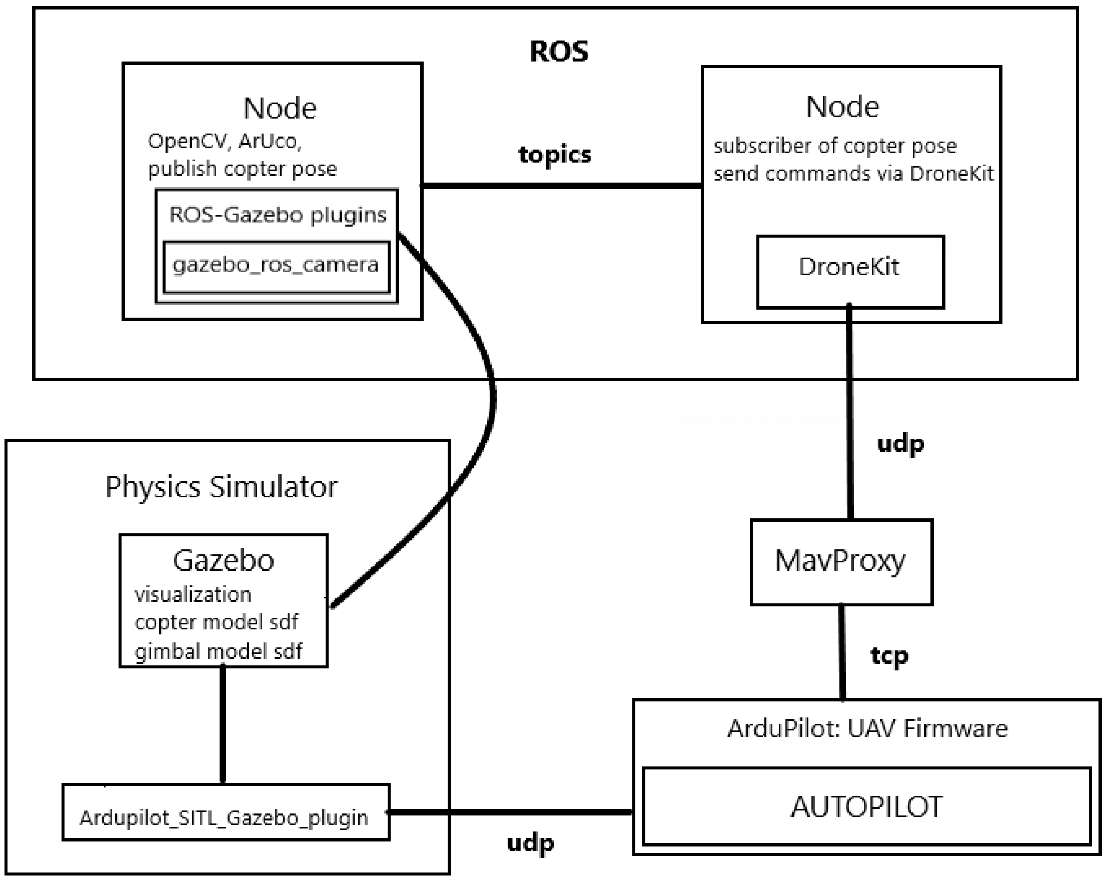


Figure 4.1: Illustration of the overall System Architecture

is communicating via TCP with a default command-line MAVProxy GCS (Ground Control Station). In order to send commands and guide the quadcopter, this system make use of the DroneKit platform that communicates with the default MAVProxy via UDP. However, DroneKit API is used inside of a ROS Node that is subscribing to specific topics in order to take pose data of the quadcopter after the image processing. On the other hand, the node on the upper right of the Fig. 4.1 make use of the gazebo_ros_camera plugin. This plugin, is attached to a sensor on a model SDF file that represents the camera inside the Gazebo environment. Then it publishes to a topic, image raw data from the Gazebo environemnt to ROS in order for further processing. In other words, the upper right node, subscribes to a specific topic and then takes the raw data from the physics simulator in order to convert them to OpenCV format and complete the detection of the marker with the help of the ArUco library. Figure 4.2 shows a more detailed illustration of how these two nodes communicate.

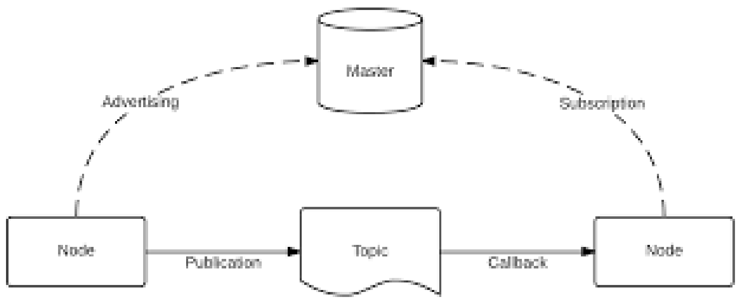


Figure 4.2: Communication between the two nodes.

The rest of this section represents each component of the Fig. 4.1.

4.1.2 ArduPilot-SITL

In this thesis, we use the *ArduPilot* [16], [17], also known as *APM* to represent the quadcopter. Ardupilot is an open source autopilot software suite for unmanned vehicles. Moreover, in order to simulate the quadcopter we use the *SITL Simulator* (Software in the Loop). SITL simulator is a build of the autopilot code using a C++ compiler, giving us the functionality to run the quadcopter and test the behaviour of the code without any hardware. Running in SITL, sensor data comes from a flight dynamics model in a flight simulator. Therefore, we use the Gazebo Simulator for the real time physics simulation that it provides. Gazebo, as said in 2.2, offers a variety of sensors to be simulated and includes models of cameras, range finders, etc. All of these sensors, models and an excellent GUI that provides for visualization, was chosen as a physics simulation for the ArduPilot code.

SITL Architecture

Fig 4.3 shows an illustration of the architecture of SITL Simulator. The port numbers here are indicative and can be vary. For example, ArduPilot and the Physics Simulation (i.e Gazebo) are connected via the ports 5501/5502 but they can easily vary to be ports 5504/5505 or other ports depending on the machine that one is running the SITL.

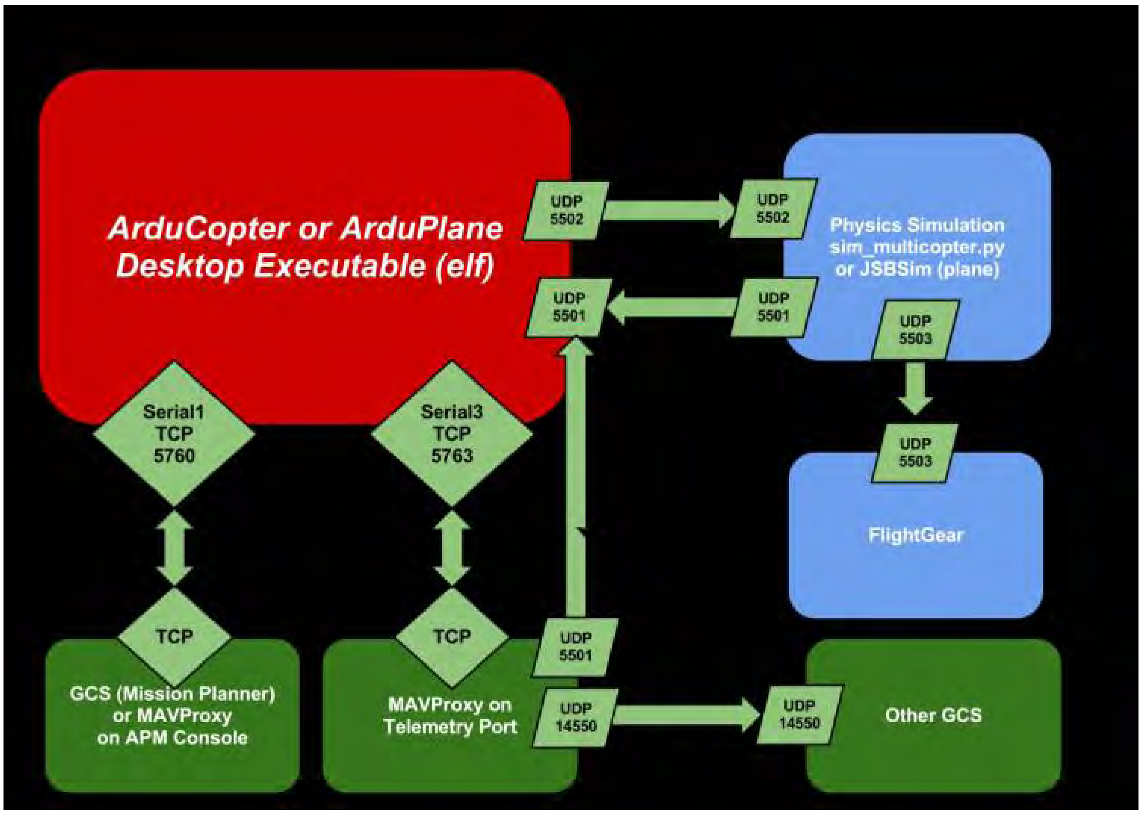

ArduCopter or ArduPlane Desktop Executable (elf)
UDP 5502
UDP 5501
UDP 5502
UDP 5501
Physics Simulation sim_multicopter.py or JSBSim (plane)
UDP 5503
UDP 5503
FlightGear
Serial1 TCP 5760
Serial3 TCP 5763
TCP
TCP
GCS (Mission Planner) or MAVProxy on APM Console
MAVProxy on Telemetry Port
UDP 5501
UDP 14550
UDP 14550
Other GCS

Figure 4.3: SITL Simulator architecture.

4.1.3 Ground Control Station

Ground Control Station, also know as *GCS* is a control centre that provices the facilities for a user to control the unmanned aerial vehicles. ArduPilot SITL includes *MAVProxy* GCS and starts by default.

MAVProxy

MAVProxy is a command-line, console base application that sends *MAVLink* messages on UAVs (such as APM) which support the protocol MAVLink.

DroneKit

Because MAVProxy is a command-line application, we use the *DroneKit* [18] aerial platform. DroneKit can be connected to SITL via UDP on a localhost ip and port 14550 or one can add additional $ip_s : port_s$ created by the MAVProxy's command*output add "ip:port"*. Moreover, through the DroneKit we send MAVLink commands to guiding and controlling the quadcopter.

4.1.4 MAVLink Protocol

MAVLink (Micro Air Vehicle Link) is the protocol to allow the communication between Unmmanned Vehicles and a Ground Control Station (GCS) so as to change the configuration of the system or to report the current state of the vehicle like the orientation, the GPS location and the speed. Moreover, MAVLink can be used for the inter-communication of the subsystem of the vehicle.

The MAVLink protocol defines a large number of MAVLink mission command messages (*MAV_CMDs*) in order to guide the quadcopter. ArduPilot, however, has adopted only a subset of the MAVLink protocol command set [19] and unsupported commands that are sent to the autopilot will simply be dropped. Table 4.1 shows the structure of ArduPilot's MAVLink mission command package format. Finally, there are *Movement commands* like:

- SET_POSITION_TARGET_LOCAL_NED
- SET_POSITION_TARGET_GLOBAL_INT
- SET_ATTITUDE_TARGET (for Guided_NoGPS mode)

Each one of these movement commands has a different format which will be explained in section 4.3.3 when we are setting the movement of the quadcopter in order to approach the visual marker.

4.1.5 Ardupilot Gazebo Plugin & Models

As explained in subsection 4.1.2 we make use of the Gazebo simulator for the Physics Simulator that it provides. In order to set up the Gazebo Simulator for SITL, we use the repository in [20]. This repository includes model SDF files of the quadcopter, gimbal, cameras e.t.c. Moreover, it includes the appropriate plugins that attached to the rotors, the IMU and the communication between the Gazebo (Physics Simulator) and the ArduPilot.

Field Name	Type	Description
target system	uint8_t	System which execute the command
target component	uint8_t	Component which execute the command, 0 for all
command	uint16_t	Command ID
confirmation	uint8_t	0: First transmission of this command. 1-255: Confirmation transmissions
parameter 1	float	Param 1, as defined for this command
parameter 2	float	Param 2, as defined for this command
parameter 3	float	Param 3, as defined for this command
parameter 4	float	Param 4, as defined for this command
parameter 5	float	Param 5, as defined for this command
parameter 6	float	Param 6, as defined for this command
parameter 7	float	Param 7, as defined for this command

Table 4.1: ArduPilot MAVLink Mission Command Package Format

4.1.6 Gazebo with ROS

Gazebo runs along ROS (Robot Operation System). To achieve ROS integration with Gazebo, a set of ROS packages provides wrappers around the Gazebo. These packages are called *gazebo_ros_pkgs* and provide the necessary interfaces so as to use ROS messages and services in order to simulate the robot (quadcopter) inside the

Gazebo environment. These gazebo_ros_pkgs packages, also, provide the necessary gazebo plugins so as to grab various sensor data to ROS.

In this thesis, in order to make the image processing, for the visual marker detection and extraction of the orientation of the quadcopter with respect to the visual marker, from the visual camera of the Gazebo to ROS, we need to add a camera plugin to the model SDF file that represents the gimbal model of the quadcopter and holds the camera sensor. This camera plugin is called *gazebo_ros_camera* and provides the raw image data from the Gazebo environment to ROS.

4.2 Gazebo Part

This section shows the necessary configuration that has been done in the Gazebo environment.

4.2.1 Creation of the visual marker

The first step was to create a SDF model file that represents and holds the information of the visual marker. To achieve this, first of all, we had to create a simple model box on the Gazebo. Then we had to improve the model appearance to include the visual marker so as simulated camera that feed information to vision processing can detect the marker. This can be done with textures on its surfaces (and 3D meshes). More specifically, on the top surface of the box we attach a texture of the visual marker and on the rest surfaces (i.e left, right, frontal, back and bottom) random textures.

A simple box

The creation of a simple box in Gazebo is describing in the next chunk of XML code. Notice that we only give the collision geometry, i.e how does it interacts with the other models in the environment of Gazebo, and the geometry visual, i.e how does it looks like in the environment of Gazebo. The size of both of them is 0.2 meters of all the directions (x, y and z). Also, notice that the coordinate system is the same as described in the figure 2.2a.

```xml
<?xml version='1.0'?>
<sdf version="1.4">
  <model name="aruco_marker">
    <pose>0 0 0.1 0 0 0</pose>
    <static>true</static>
      <link name="box">

        <collision name="collision">
          <geometry>
            <box>
              <size>0.2 0.2 0.2</size>
            </box>
          </geometry>
        </collision>

        <visual name="visual">
          <geometry>
            <size>0.2 0.2 0.2</size>
          </geometry>
        </visual>

      </link>
    </model>
</sdf>
```

Add textures

In order to add our textures in the box-model-SDF-file, Gazebo recognize STL, OBJ and Collada files. Here, we make use of Collada files. Moreover, textures were added with the help of the *Blender* (an open source 3D creation suite). Figure 4.4 shows the textures that has been added. After the insertion of the textures upon the surfaces of the box, we export the file from the Blender as Collada (.dae) file. Then it can be added in the SDF file that represents the box as follows:

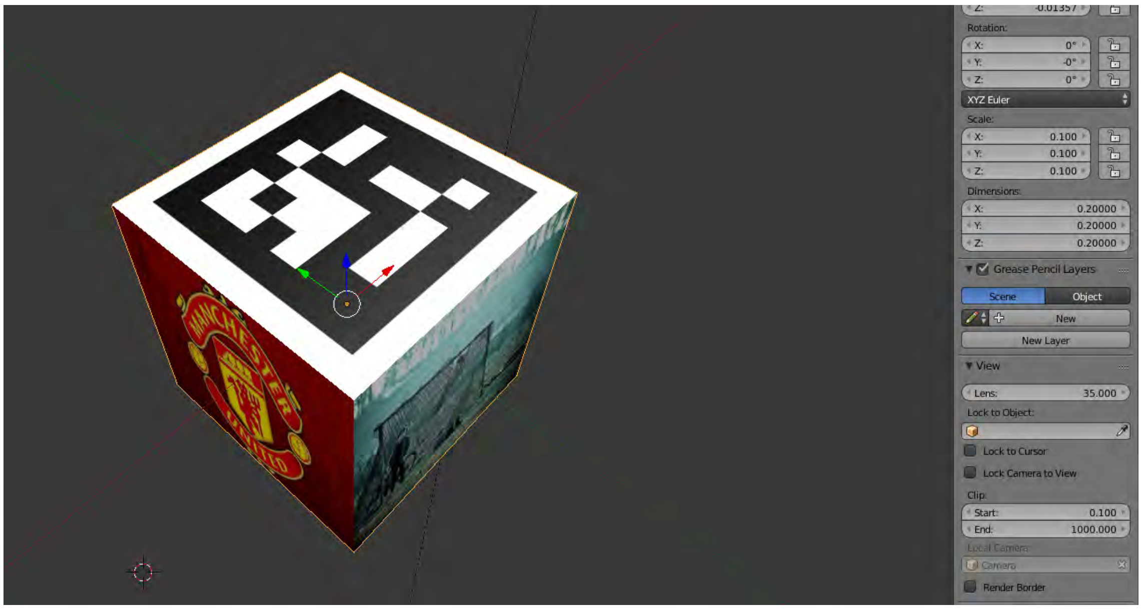
Replace:

```xml
        <visual name="visual">
```

```
      <geometry>
        <size>0.2 0.2 0.2</size>
      </geometry>
    </visual>
```

with:

```
    <visual name="visual">
      <geometry>
        <mesh>
          <uri>model://aruco_marker/meshes/aruco_marker.dae</uri>
        </mesh>
      </geometry>
    </visual>
```



```
Z: -0.01357
Rotation:
X: 0°
Y: -0°
Z: 0°
XYZ Euler
Scale:
X: 0.100
Y: 0.100
Z: 0.100
Dimensions:
X: 0.20000
Y: 0.20000
Z: 0.20000
Grease Pencil Layers
Scene  Object
New
New Layer
View
Lens: 35.000
Lock to Object:
Lock to Cursor
Lock Camera to View
Clip:
Start: 0.100
End: 1000.000
Local Camera:
Camera
Render Border
```

Figure 4.4: Visual box with textures upon its surfaces. On the upper surface of the box, the coordinate system of the visual marker is presented. Also, in the middle right, the x, y, z dimensions of the box are presented.

4.2.2 Camera sensor plugin

As explained in subsection 4.1.6, in order to make the image processing and capture the frames from the virtual camera of Gazebo to ROS we need to add a plugin to the camera sensor. The camera sensor is attached to a gimbal model-SDF-file. This functionality can be achieved by adding the following code in the gimbal's model-SDF-file. Note that all the SDF files (i.e gimbal, quadcopter e.t.c) are provided by the repository of Seunghwan Jo as said in 4.1.5).

```
<sensor name="camera1" type="camera">
    <pose>0 0 0 -1.57 -1.57 0</pose>
    <updateRate>10.0</updateRate>
    <camera>
        <horizontal_fov>1.0471975512</horizontal_fov> <!-- 60 degrees -->
        <image>
            <width>640</width>
            <height>480</height>
        </image>
        <clip>
            <near>0.05</near>
            <far>300</far>
        </clip>
    </camera>

    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>10.0</updateRate>
      <cameraName>iris/camera1</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>tilt_link</frameName>
      <hackBaseline>0.07</hackBaseline>
      <Cx>3.3765716562185179e+02</Cx>
      <Cy>2.3847924463232465e+02</Cy>
      <focalLength>8.1706663835848678e+02</focalLength>
      <distortionK1>-2.4333343952387267e-02</distortionK1>
      <distortionK2>1.2902389844068224e-01</distortionK2>
      <distortionK3>-3.3069024740403519e-01</distortionK3>
      <distortionT1>-4.3578152766553984e-03</distortionT1>
      <distortionT2>2.1160667357386460e-03</distortionT2>
    </plugin>
  </sensor>
```

These properties of this sensor plugin are for a *Logitech HD Webcam C270*. Let's discuss some of the properties:

The sensor name *camera1* must be unique from all the other sensor names.

```
<sensor name="camera1" type="camera">
```

A rotation of the camera sensor so as the coordinate axes of the camera are matched with those of OpenCV (Z: forward, Y: down, X: right), in contrast with the Gazebo default coordinate system (X: forward, Z: up, Y: right). More specifically, a negative $\frac{\pi}{2}$ rotation over the x axis followed by a negative $\frac{\pi}{2}$ rotation over the y axis. Therefore, after the rotation, Z axis becomes the optical axis, see Fig. 3.1.

```
<pose>0 0 0 -1.57 -1.57 0</pose>
```

Number of times per second a new camera image is taken within Gazebo simulator.

```
<updateRate>10.0</updateRate>
```

These values are matched with these of manufacturer's specs on our physical camera hardware. The near and far clip parameters are simulation-specific parameters that give an upper and lower bound to the distance in which the camera is able to see objects in the Gazebo environment.

```
<horizontal_fov>1.0471975512</horizontal_fov> <!-- 60 degrees -->
<image>
  <width>640</width>
  <height>480</height>
</image>
```

```
<clip>
    <near>0.05</near>
    <far>300</far>
</clip>
```

Here we define topics on ROS that the camera will be publishing to, for both the image topic and the camera info topic.

```
<cameraName>iris/camera1</cameraName>
<imageTopicName>image_raw</imageTopicName>
<cameraInfoTopicName>camera_info</cameraInfoTopicName>
```

Finally, the parameters Cx, Cy, $focalLength$ and distortion coefficients $K1$, $K2$, $K3$, $T1$, $T2$ are found via the calibration process. Notice that in Gazebo environment there is not any distortion on capture frames, so these values are optional and can be replaced by zeros or not add them at all.

4.2.3 Joint Frames

The final step, for the Gazebo setup, is to make the quadcopter model-sdf-file, called *Iris*, to integrate the camera gimbal. This can be done by the joint tag in the iris model-SDF-file with the following way:

```
<include>
    <uri>model://gimbal_small_2d</uri>
</include>

<joint name="iris_gimbal_mount" type="revolute">
    <parent>iris::base_link</parent>
    <child>gimbal_small_2d::base_link</child>
</joint>
```

First we must include the gimbal model-SDF-file to the iris model-SDF-file. After that we join the two models, with the Iris model as the parent and the gimbal model as the child.

4.3 ROS Part

This section shows the process that have done in the ROS part. In particular, in this thesis, firstly we have to create a catkin package so as to execute the nodes. To achieve are goal we construct two nodes. The first node does the image processing and detect the visual marker. After that, it extracts the yaw, pitch, roll angles and the translation vector and publishes them to topics. The second node is subscribing to those topics and is guiding the quadcopter with the help of those values that is receiving. The guidance of the quadcopter, as said in previous sections is done via the API of DroneKit which sends command messages to ArduPilot.

4.3.1 Creation of the package

First of all, we need to create a *catkin package*. To be more specific, catkin is a CMake-based build system that is used to build all packages in ROS. On the other hand, packages are the software organization unit of ROS code and contain libraries, executables, scripts e.t.c. Moreover, a catkin package must contain a package.xml file which is a description of the package and it servers to define dependencies between other packages of ROS system and meta information about it like version, maintainer, license e.t.c.

Our package has dependencies on geometry messages (to publish the pose of the quadcopter), sensor messages (to receive the capture images of the virtual camera of Gazebo), cv_bridge package (to interface OpenCV with the ROS raw data of the sensor, see Fig 4.5) and ArUco library (to detect the visual marker).

4.3.2 Pose Extraction Node

This pose extraction node is subscribing to the topic *iris/camera1/image_raw* so as to take the sensor data of the virtual camera (as explained in the subsection 4.2.2). The next step is to transform the raw data that is sending from the Gazebo environment to the OpenCV format. In other words, converting ROS images into OpenCV images with the CvBridge ROS library that provides that interface. More analytically, ROS passes around images in its own sensor_msgs/Image message format, but

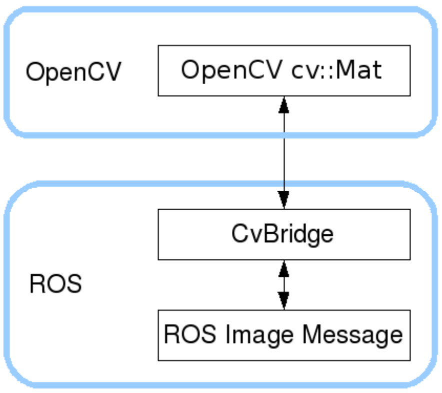


Figure 4.5: Cv_bridge for converting OpenCV images to ROS format to be published over ROS and vice versa.

we want to use images in conjunction with OpenCV. After he have a capture image in OpenCV format we can, then, do the appropriate image processing in order to detect the visual marker with the help of the ArUco library.

The ArUco library uses the *detect* function which takes the intrinsic camera parameters, the distortion coefficients and the current image and outputs the pose (*rotation* and *translation*) of the camera with respect to visual marker if it exists in the current frame. Since ArUco library rely on the OpenCV library, OpenCV works with the Axis-Angle notation. More specifically, it always returns a vector *rvec* that represents the rotation of a coordinate system with respect to another reference coordinate system (e.g the rotation of the camera coordinate system with respect to the visual marker coordinate system), *see subsection 3.3.1.* The first step, is to transform the Axis-angle rotation to a 3 x 3 matrix rotation. This can be done, easily, with the build-in *Rodrigues* function that the openCV library provides. After that we end up with a 3 x 3 matrix that represent the rotation of the camera with respect to the visual marker. The next step, is to take the orientation of the quadcopter with respect to the visual marker. Thus, we need to make an extra rotation by a post multiplication of the 3 x 3 rotation matrix from camera to visual marker with the 3 x 3 rotation matrix from the quadcopter to the camera. The second matrix, in other words, represents how the axes of the quadcopter are aligned with those of the camera. Equation 4.3.1, shows the rotation chain that has to be done.

$$R^{marker}_{quadcopter} = R^{marker}_{camera} R^{camera}_{quadcopter} \tag{4.3.1}$$

In order to find the second matrix we need to define the quadcopter and camera coordinate systems. These definitions are explained in the following subsection.

Different Coordinate Systems

As previously said, we assume that the coordinate system of the camera, as defined in the OpenCV, is Y axis down, Z axis forward and X axis to the right (see Fig. 3.1). Moreover, we assume that the coordinate system of the UAVs body frame is Z axis up, X axis forward and Y axis to the left. Figure 4.6 shows an illustration of the UAVs body frame.

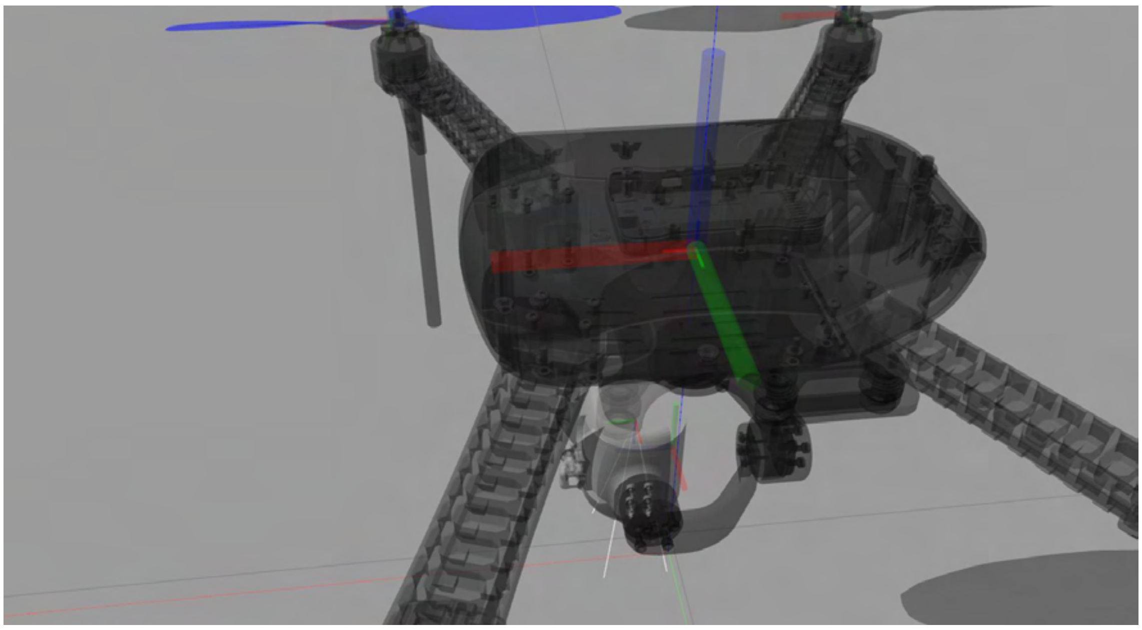


Figure 4.6: Quadcopter coordinate system with Z up, Y left and X forward.

Before we construct the 3 x 3 matrix $R^{camera}_{quadcopter}$, we need to remember that the camera is pointing straight down. Therefore, we have the two coordinate systems (quadcopter, camera) as the Fig. 4.7 shows.

Therefore, the 3 x 3 matrix $R^{camera}_{quadcopter}$ will be:

$$R^{camera}_{quadcopter} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{4.3.2}$$

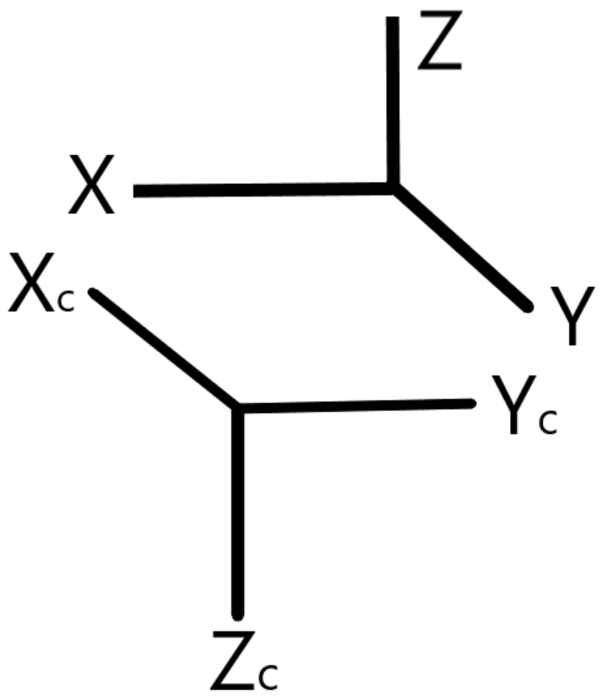


Figure 4.7: Quadcopter Coordinate System (Upper) and Camera Coordinate System (Bottom).

Equation 4.3.2 shows that the X axis of the quadcopter coordinate system is parallel but in the opposite direction of the Y_c axis of the camera's one. Similarly, the Z axis of the quadcopter with the Z_c axis of the camera and the Y axis of the quadcopter with the X_c axis of the camera.

After this step and the multiplication of the equation 4.3.1 we are ready to extract the euler angles that represent the rotation of the quadcopter with respect to the visual marker.

Euler Angles Extraction

In order to extract the euler angles from the rotation of the quadcopter coordinate system relative to the visual marker we make use of the 3 x 3 rotation matrix of the eq. 4.3.1. This matrix, can be transformed to euler angles if we determine which of these angles applied first, second and third (see subsection 3.3.3). Therefore, the *yaw* angle (heading) is applied on the Z axis, the *roll* angle on the X axis and the *pitch* angle on the Y axis (see Fig. 4.6.

In short, what we have, so far, is the order of the successive rotations for the quadcopter as *z*-*y*-*x* and,

- yaw angle (heading) $\boldsymbol{\theta}$ applied first on Z axis
- pitch angle $\boldsymbol{\phi}$ applied second on Y axis
- roll angle $\boldsymbol{\psi}$ applied third on X axis

In a more mathematical representation we can express the above operations as follows:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & -\sin(\psi) \\ 0 & \sin(\psi) & \cos(\psi) \end{bmatrix} =$$
$$\begin{bmatrix} c(\theta)c(\phi) & c(\theta)s(\phi)s(\psi) - c(\psi)s(\theta) & s(\theta)s(\psi) + c(\theta)c(\psi)s(\phi) \\ c(\phi)s(\theta) & c(\theta)c(\psi) + s(\theta)s(\phi)s(\psi) & c(\psi)s(\theta)s(\phi) - c(\theta)s(\psi) \\ -s(\phi) & c(\phi)s(\psi) & c(\phi)c(\psi) \end{bmatrix}$$

, where c denotes to cos and s denotes to sin.

After that we can get the euler angles as follows:

$$yaw = \theta = \text{atan2}(m_{10}, m_{00}) \tag{4.3.3}$$

$$pitch = \phi = \text{atan2}(-m_{20}, \sqrt{m_{00}^2 + m_{10}^2}) \tag{4.3.4}$$

$$roll = \psi = \text{atan2}(m_{21}, m_{22}) \tag{4.3.5}$$

, where $m_{i,j}$ denote to a cell of the matrix with i, j the row and column respectively.

Publish Transformation

After the extraction of the euler angles and the translation vector, the pose extraction node is publishing the transformation (rotation and translation) of the quadcopter with respect to the visual marker with the help of ROS messages. To be more specific, it advertise a geometry_msgs/Pose and a geometry_msgs/Vector3 messages.

The geometry_msgs/Vector3 is advertising to the topic *"aruco/linear/ypr"* and is on the form:

```
float64 x // rotation on the X axis
float64 y // rotation on the y axis
float64 z // rotation on the z axis
```

The geometry_msgs/Pose is advertising to the topic *aruco/linear/pose* and is on the form:

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

As we see, it is comprised of geometry_msgs/Point and geometry_msgs/Quaternion. More specifically, geometry_msgs/Point and geometry_msgs/Quaternion, respectively, are:

```
/* geometry_msgs/Point */
float64 x // translation on the x direction
float64 y // translation on the y direction
float64 z // translation on the z direction
/* geometry_msgs/Quaternion */
float64 x
float64 y
float64 z
float64 w
```

Notice that, in this thesis, we do not make use of the quaternions. However, to cover all the aspects of rotations (for future work) we simply transform to Axis-Angle rotation (as explained in the subsection 3.3.1) to quaternions with the help of a function that *tf* ROS package provides. This function is called *q_axes(axis, angle)* and takes as input the axis and the angle of rotation of the Axis Angle representation and outputs the quaternion. Alternatively, we could just fill the quaternion elements with arbitrary numbers.

4.3.3 Navigation Node

This navigation node subscribes to the topics *aruco/linear/ypr* and *aruco/linear/-pose* so as to take the rotation and translation of the quadcopter. When the quadcopter's camera captures the visual marker on its image frames, the pose extraction node is advertising the data to the navigation node. More analytically, in order the navigation node to retrieve these data, is having two callback function for each topic. The callback functions of the *aruco/linear/ypr* and *aruco/linear/pose* topics are retrieving, respectively, the data as follow:

```
def ypr_callback(data):
    yaw =data.z
    pitch =data.y
    roll =data.x

def pose_callback(data):
    dx =data.position.x
    dy =data.position.y
    dz =data.position.z
    # quaternion data not used
```

Then, through three procedures, with the help of DroneKit API, we are guiding and giving commands to the quadcopter. More analytically, these procedures are:

Position Target Global

This procedure, after the extraction of the new GPS coordinates (latitude, longitude) as explained in the section 3.4, is guiding the quadcopter to approach the visual marker using the new GPS coordinates and is given by the following command:

```
msg =vehicle.message_factory.set_position_target_global_int_encode(
       0, # time_boot_ms (not used)
       0, 0, # target system, target component
       mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT_INT, # frame
       0b0000111111111000, # type_mask (only positions enabled)
       location.lat*1e7, # lat_int - X Position in WGS84 frame in 1e7 * meters
       location.lon*1e7, # lon_int - Y Position in WGS84 frame in 1e7 * meters
```

```
        location.alt, # alt
        0, # X velocity in NED frame in m/s
        0, # Y velocity in NED frame in m/s
        0, # Z velocity in NED frame in m/s
        0, 0, 0, # afx, afy, afz acceleration (not supported)
        0, 0) # yaw, yaw_rate (not supported)
    # send command to vehicle
    vehicle.send_mavlink(msg)
```

Notice that, the *MAV_FRAME_GLOBAL_RELATIVE_ALT_INT* is the frame of reference which uses the WGS84 global coordinate system for latitude and longitude, but sets altitude as relative to the home position in metres rather than the mean sea level (MSL).

Position Target Local

This procedure, takes the translation vector from the *aruco/linear/pose* topic. More analytically, the quadcopter is guided with the following command:

```
msg =vehicle.message_factory.set_position_target_local_ned_encode(
        0, # time_boot_ms (not used)
        0, 0, # target system, target component
        mavutil.mavlink.MAV_FRAME_BODY_OFFSET_NED, # frame, x: forward, y:right, z
                                                    : down
        0b0000111111111000, # type_mask (only positions enabled)
        north, east, down, # x, y, z positions
        0, 0, 0, # x, y, z velocity in m/s (not used)
        0, 0, 0, # x, y, z acceleration (not supported yet)
        0, 0) # yaw, yaw_rate (not supported yet)
    # send command to vehicle
    vehicle.send_mavlink(msg)
```

Notice that, there are four frames of reference that can be used in the SET_POSITION _TARGET_LOCAL_NED command,

- MAV_FRAME_LOCAL_NED

- MAV_FRAME_LOCAL_OFFSET_NED
- MAV_FRAME_BODY_OFFSET_NED
- MAV_FRAME_BODY_NED

Here, we use the *MAV_FRAME_BODY_OFFSET_NED* frame, in order to move the vehicle relative to its current position. More specifically, within this frame, positions are relative to the current vehicle position in a frame based on the vehicle's current heading. We use this to specify a position x metres forward from the current vehicle position, y metres to the right, and z metres down.

Condition Yaw

This procedure, sends MAV_CMD_CONDITION_YAW message command to point the quadcopter at a specified heading in degrees. In other words, point (yaw) the nose of the vehicle towards a specified heading. This command has the following form:

```python
msg =vehicle.message_factory.command_long_encode(
       0, 0, # target system, target component
       mavutil.mavlink.MAV_CMD_CONDITION_YAW, #command
       0, #confirmation
       heading, # param 1, yaw in degrees
       0, # param 2, yaw speed deg/s
       1, # param 3, direction -1 ccw, 1 cw
       is_relative, # param 4, relative offset 1, absolute angle 0
       0, 0, 0) # param 5 ~ 7 not used
   # send command to vehicle
   vehicle.send_mavlink(msg)
```

Note that, the parameter 4 allows us to specify whether the target direction is absolute (i.e target heading in degrees [0-360] with 0 degrees as North) or relative to the current yaw direction (i.e the change in heading in degrees).

Moreover, if the direction is relative, as specified in parameter 4, we can also specify, with the parameter 3, whether the value is added or subtracted from the current heading.

For our purpose, we set the parameter 4 to *1*, so as to point the noise of the quadcopter to a new direction but relative to the previous heading.

Finally, in order to approach the visual marker, this thesis is applied the three procedures in a row (i.e in the order they were explained). In other words, firstly, the quadcopter approaches the target with the help of the GPS coordinates. Then, it increases the precision of the approachment while navigating with the help of the translation vector. Finally, it aligns its coordinate axes with the visual marker's one by setting its yaw angle that was published by the pose extraction ROS node. However, it is up to the application writer to determine how the quadcopter approaches the visual marker.

Chapter 5

Proof of Concept Scenario

This chapter shows an indicative test that has been done in order to verify that what this thesis proposes actually works. This test demonstrates a simple mission planning process that the Drone does, in order to find and approach the visual marker within the Gazebo environment with the methods that have been described in the implementation chapter 4.

5.1 Mission Planning

First of all, we define a new mission with five way points arranged in a square around the central position of the Drone. To define the five way points we make use of the MAV mission navigation command *MAV_CMD_NAV_WAYPOINT* (Table 5.1 shows the format).

Command Field	Mission Planner Field	Description
param 1	Delay	Hold time at mission waypoint in decimal seconds
param 2		(not supported)
param 3		(not supported)
param 4		(not supported)
param 5	Lat	Target latitude. If zero, the Copter will hold at the current latitude.
param 6	Lot	Target longitude. If zero, the Copter will hold at the current longitude.
param 7	Alt	Target altitude. If zero, the Copter will hold at the current altitude.

Table 5.1: ArduPilot MAV_CMD_NAV_WAYPOINT Command.

DroneKit's API sends this type of navigation command message with the following way.

```
Command(0, # target_system
        0, # target_component
        0, # seq number within the mission, API will automatically set this
        mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT, # The frame of reference
                                                  used for the location parameters
                                                   5, 6, 7. This frame uses the
                                                  WGS84 global coordinate system
                                                  for latitude and longitude, but
                                                  sets altitude as relative to the
                                                   home position in metres (home
                                                  altitude = 0)
        mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, # The specific mission command
        0, 0, # (not supported)
        0, # param 1
        0, # param 2
        0, # param 3
        0, # param 4
        point.lat, # param 5
        point.lon, # param 6
        altitude # param 7))
```

We set five commands with different points to define the mission. The waypoints are positioned to form a square of side length 8 meters around the current Drone's position.

5.2 Approachment of the target

While the Drone is doing the mission, is looking for the visual marker within the environment. To be more specific, the navigation node is listening on a callback function to hear the data that will be published from the pose extraction node which estimates the pose of the Drone with respect to visual marker. When the Drone, finally, detects the marker, the pose extraction node publishes the pose and the navigation node receives the data. The final step is the navigation node to send MAVLink commands so as the Drone can localize itself and approach the target.

After that, firstly, the Drone approaches the visual marker within the first procedure (as described in the chapter 4). In other words, we are giving to the Drone the new location to follow which is the GPS coordinates of the visual marker. Secondly, we increase the accuracy of the approach. This can be done, by giving the Drone a command to navigate itself with the second procedure (i.e the translation vector of the Drone with respect to the visual marker). Finally, we rotate the axes of the Drone to be aligned with those of the marker. If the drone does not detect the visual marker, it returns to the home position.

5.3 Planned Mission and Actual Path Followed

Bellow, in Fig. 5.1, is given a plot that shows the planned mission and the actual path travelled by the Drone so as to reach the visual marker, as well as, the position of the visual marker within the environment. To be more specific, the white building shows the home position of the Drone while the black line illustrates the planned mission that was set. Moreover, the red $WP1, WP2$ and so on, are the waypoints of the planned mission, while the yellow dots illustrated the actual path that the drone followed. Also, the purple triangle shows the location of the visual marker within the environment. Finally, the orange rectangles correspond to each blue letters A, B and so on, which show the points that are captured within the gazebo environment to illustrate the motion of the Drone.

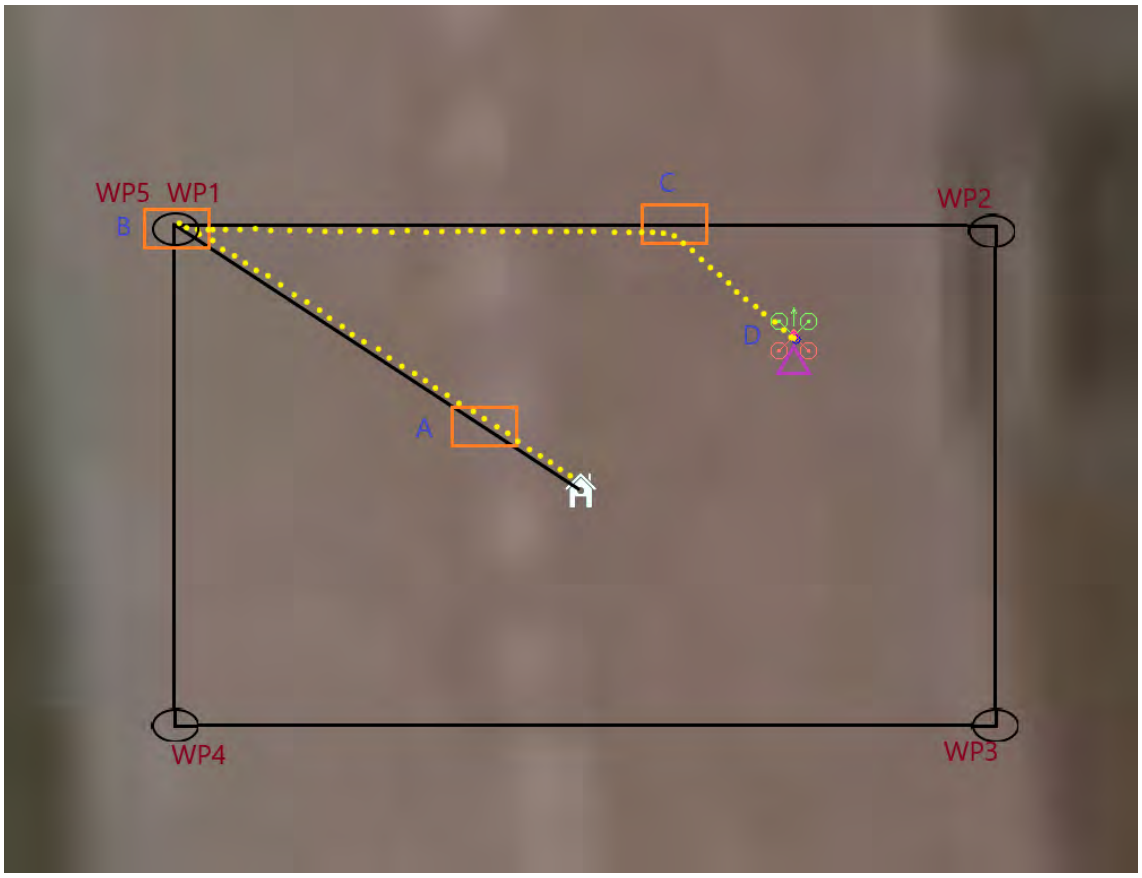


Figure 5.1: The path that the Drone follows in order to find and approach the Visual Marker.

As the Fig. 5.1 shows, the Drone is starting the mission from its home position.

While is looking for the visual marker within the environment without a success, it reaches the waypoint 1. Then, it continues its mission, and once it reaches the waypont 2 the visual marker is, finally, detected (purple triangle in the Fig. 5.1). Then, the Drone approaches the target, firstly, with the method that makes use of the GPS position of the visual marker, and, after that, with the method that makes use of the translation vector that is extracted from the pose of the quadcopter relative to the marker. Finally, it aligns its axis with the visual marker's axis by applying a yaw rotation.

The next figures illustrate the above mission, as well as the motion of the Drone until it reaches the destination (i.e detect the visual marker and approach it).

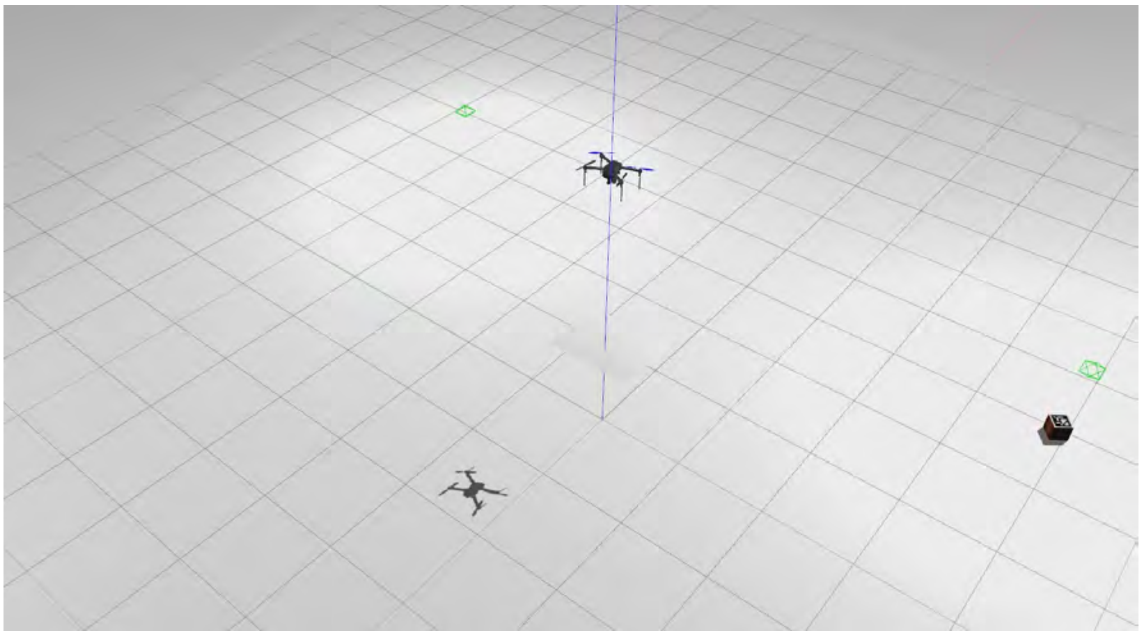


Figure 5.2: Illustrates the Gazebo world-environment. Here, the Drone is taking off and starting its mission, while the visual marker (purple triangle on the Fig. 5.1) lies randomly within the world. The Drone is at its home position (white building in Fig. 5.1).

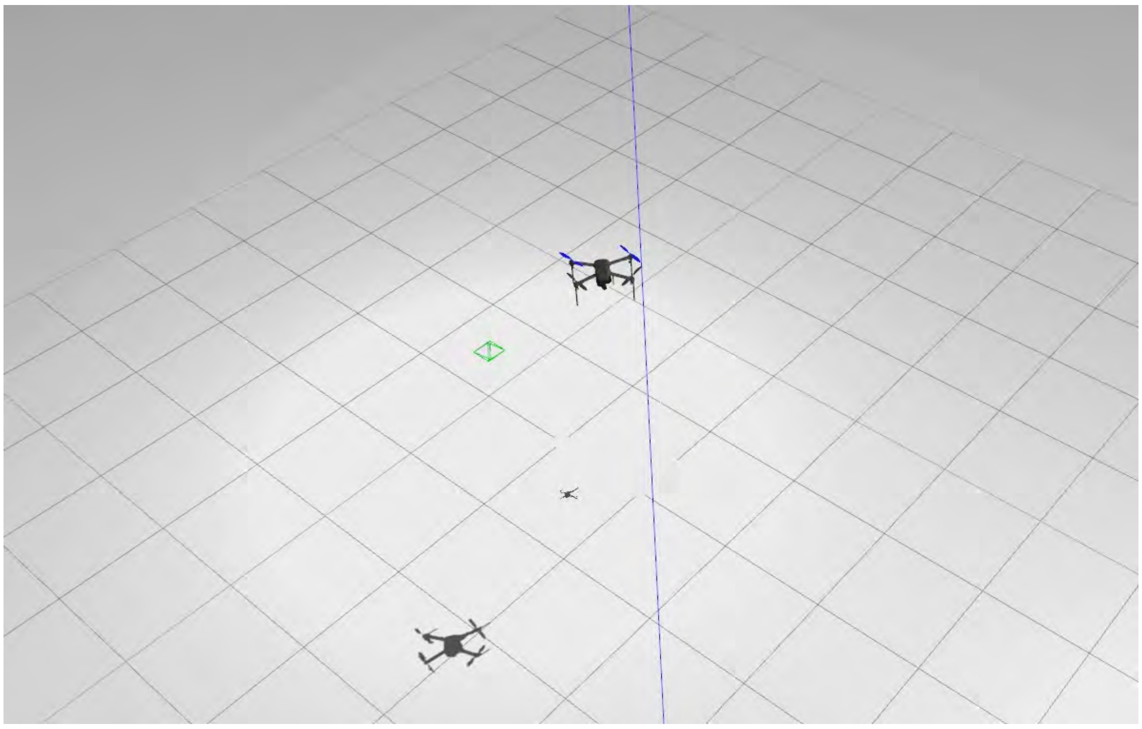
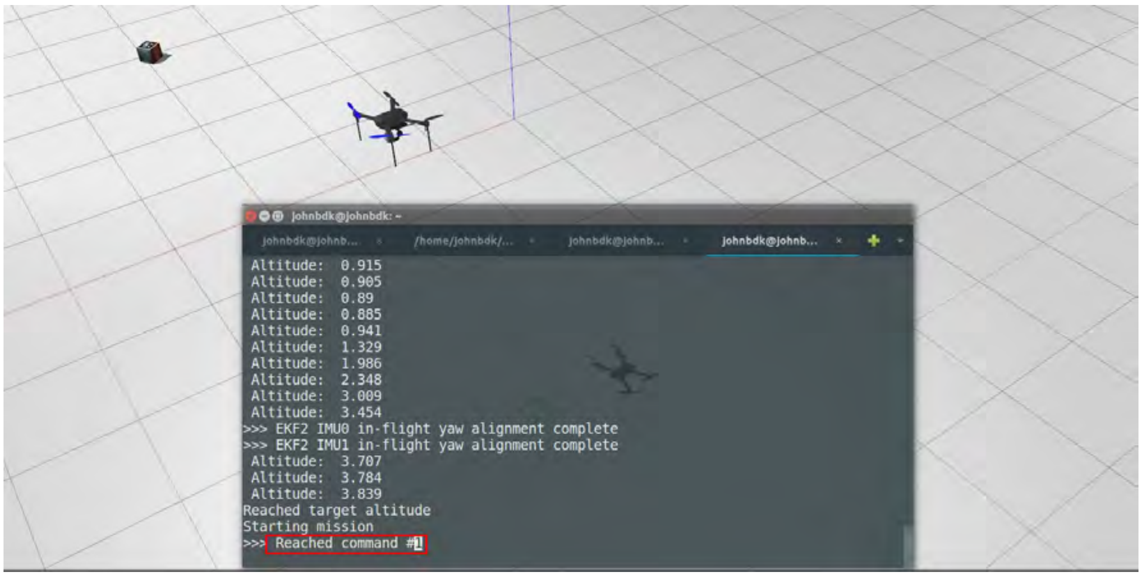


Figure 5.3: While the Drone is searching for the visual marker, it reaches the point A (blue) in Fig. 5.1 without finds the visual marker.



```
 Altitude:  0.915
 Altitude:  0.905
 Altitude:  0.89
 Altitude:  0.885
 Altitude:  0.941
 Altitude:  1.329
 Altitude:  1.986
 Altitude:  2.348
 Altitude:  3.009
 Altitude:  3.454
>>> EKF2 IMU0 in-flight yaw alignment complete
>>> EKF2 IMU1 in-flight yaw alignment complete
 Altitude:  3.707
 Altitude:  3.784
 Altitude:  3.839
Reached target altitude
Starting mission
>>> Reached command #1
```

Figure 5.4: The Drone reaches the waypoint #1 of the planned mission, point B (blue) in Fig. 5.1, without finds the visual marker. After that, it continues its navigation to the next waypoint #2.

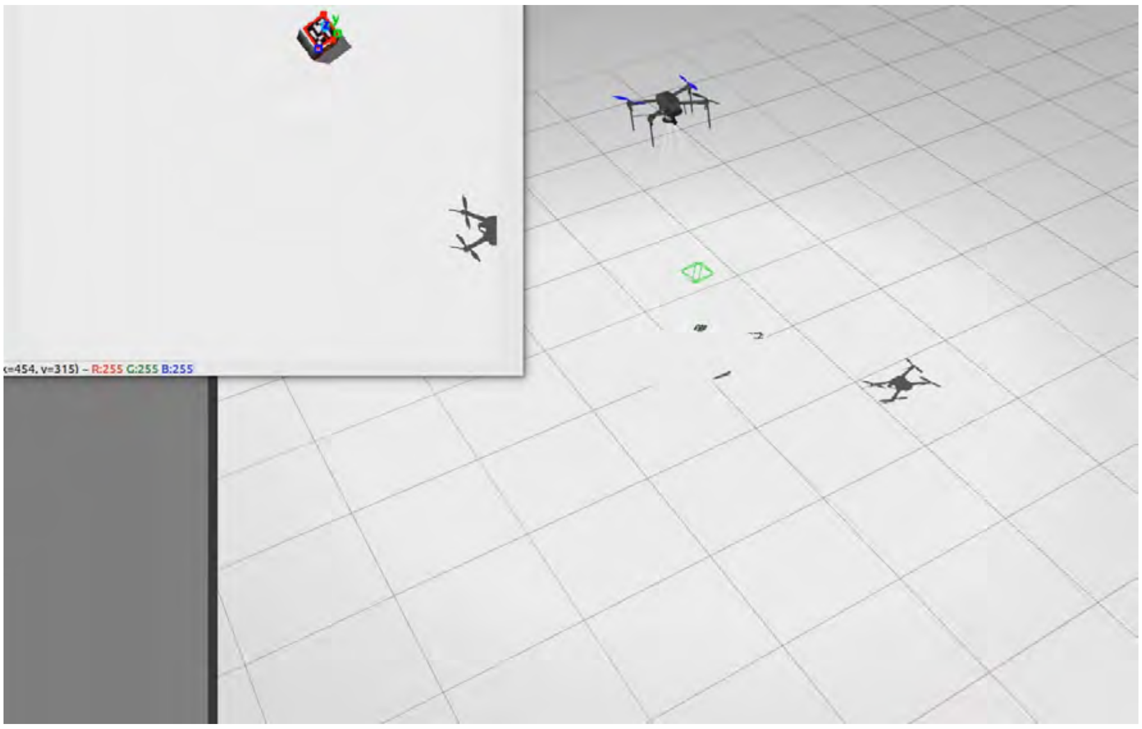


Figure 5.5: While the Drone is reaching the waypoint #2, it sees the visual marker in its camera frame at the blue point C in Fig. 5.1. After that, it is navigating with the method that uses the GPS coordinates of the visual marker in order to approach it.

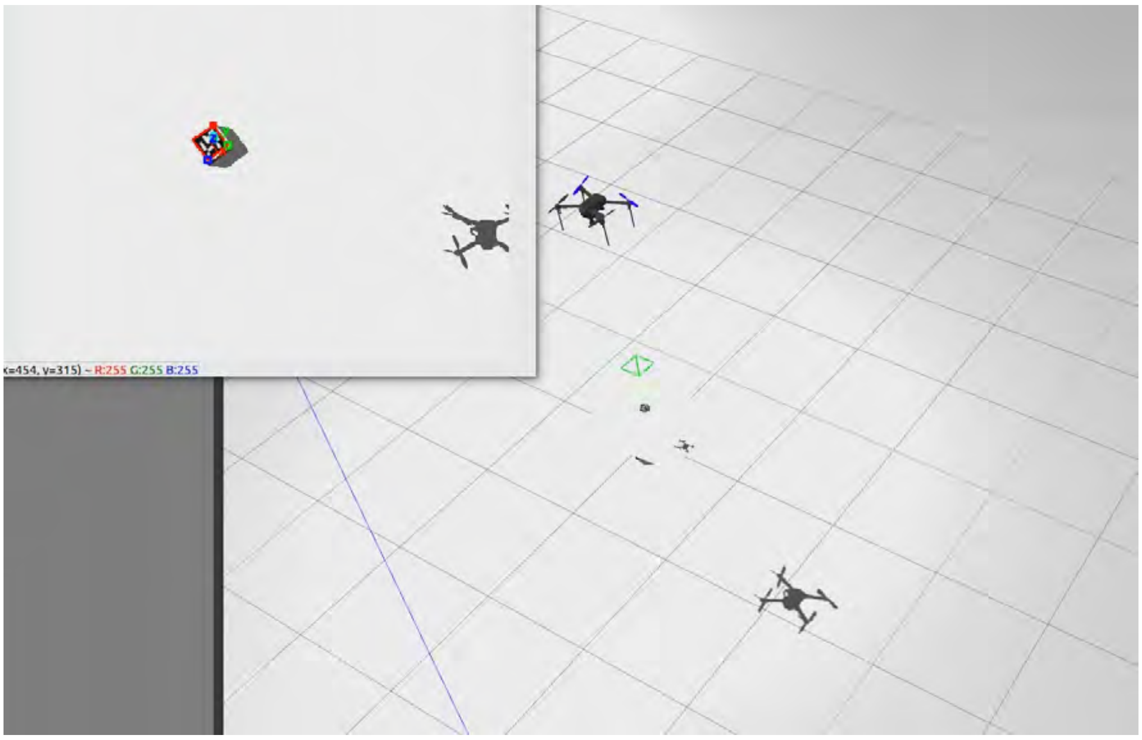


Figure 5.6: The Drone reaches its destination of the new GPS coordinates, point D (blue) in Fig. 5.1, and it is hovering upon the visual marker (purple triangle in Fig. 5.1) without decreases its altitude (i.e it remaines 4m above the ground).

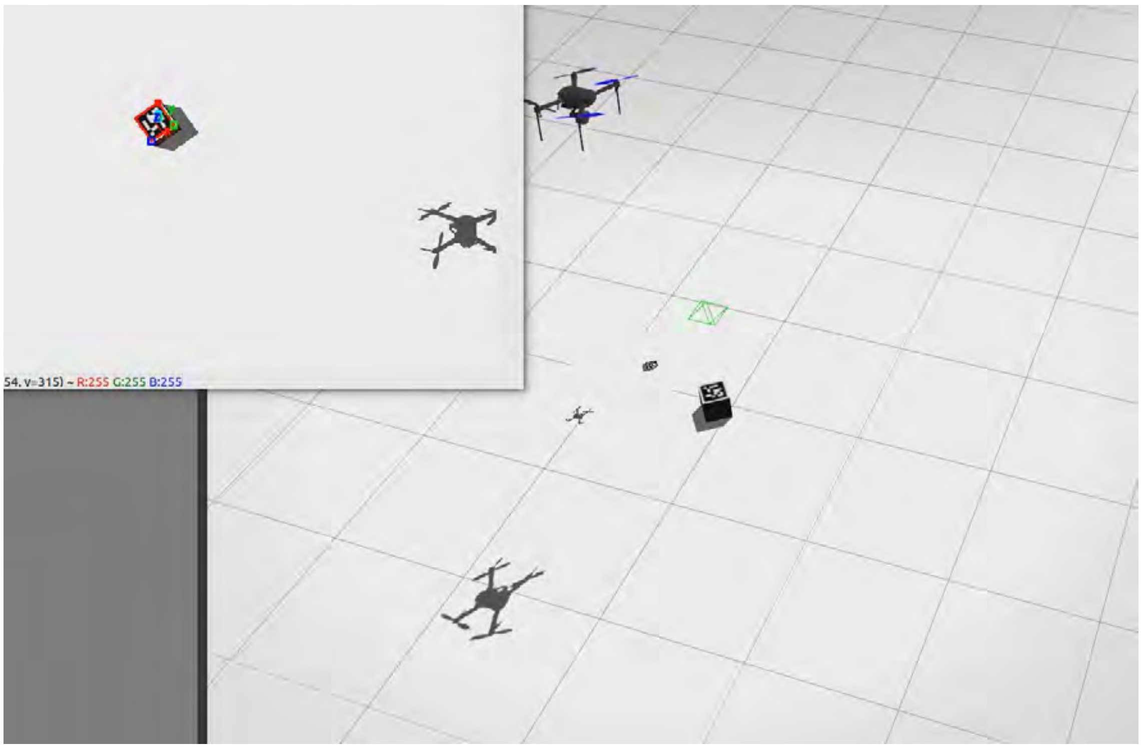


Figure 5.7: The Drone is hovering upon the visual marker. Just another screen snapshots from another perspective. The image window shows the detected visual marker.

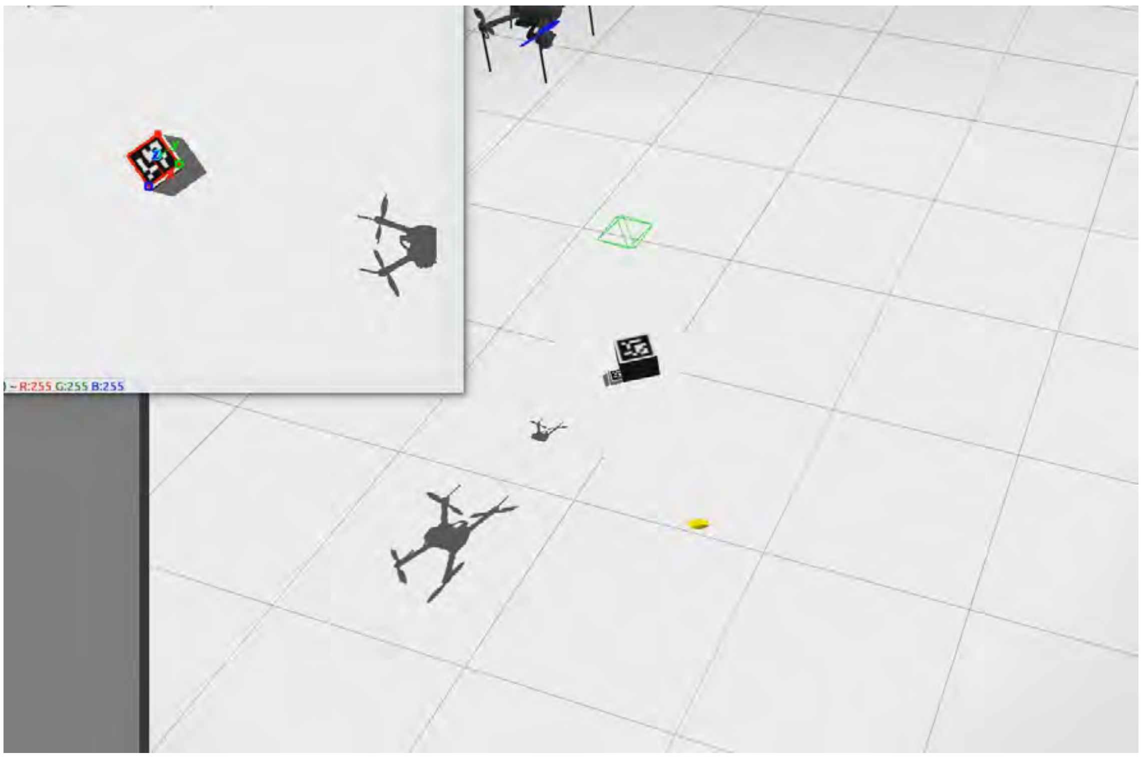


Figure 5.8: After some seconds the Drone starts approaching the visual marker with the translation vector. Thus, it increases the accuracy of the approach.

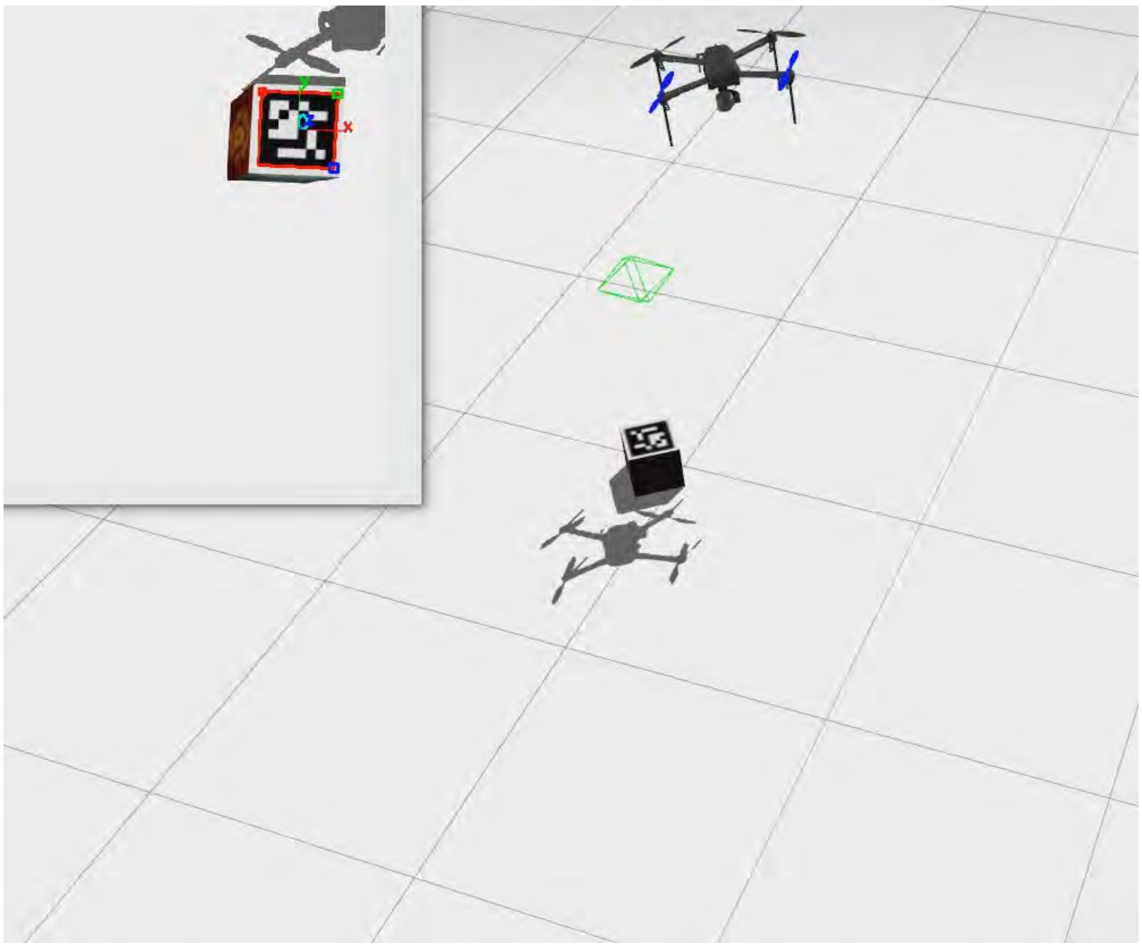


Figure 5.9: Meanwhile the Drone starts rotating (change it's heading - nose of the quadcopter) in order to align its axes with the marker's one.

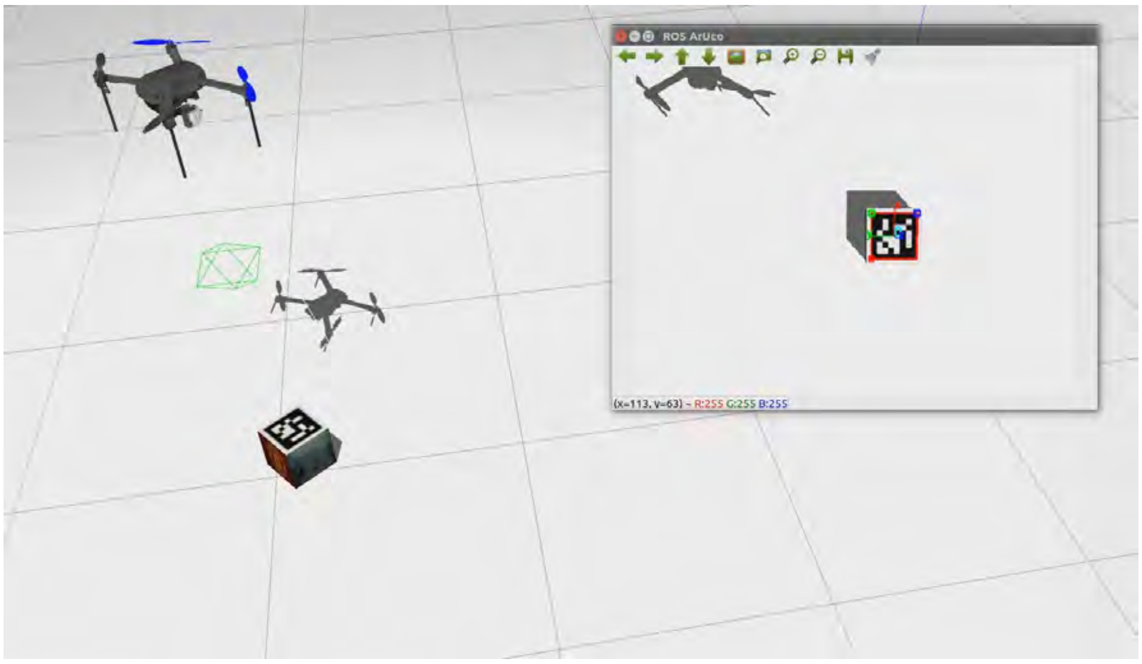


Figure 5.10: Illustration of the Drone's position and orientation after the completion of the translation and rotation commands. Note that the Drone decreases its altitude at the half of its initial altitude.

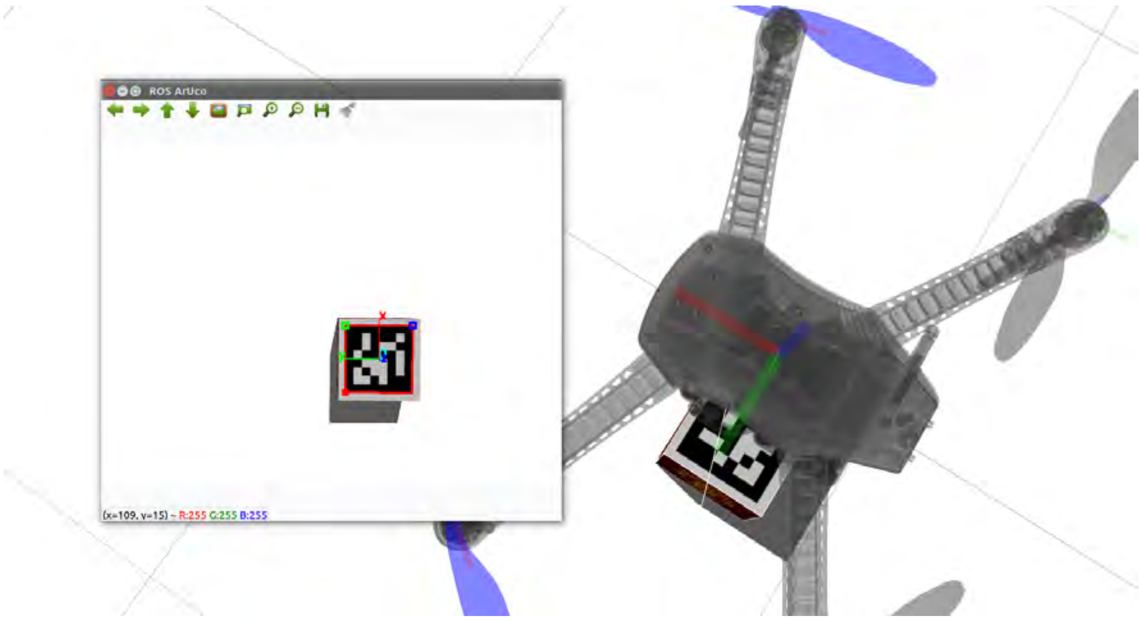


Figure 5.11: Illustration of the alignment of the Drone's axis with the Marker's one.

Chapter 6

Related Work

This chapter discusses similar works that have been done in the detection of a visual marker with the ArUco library in ROS. Moreover, it demonstrates the main differences between those works and this thesis.

6.1 Similar work

There are several packages in the ROS environment that can be used in order to detect fiducial markers with the ArUco library and the OpenCV in ROS. Some of these packages are the following.

6.1.1 ROS package aruco_ros

This package [21] has been developed by the *Ava group* of the University of Cordoba (Spain). It provides real-time marker based 3D pose estimation using AR markers. It is a software package and ROS wrappers of the ArUco Augmented Reality marker detector library.

More analytically, it uses the cv_bridge package in order to convert image raw data from ROS to OpenCV format. After that, the ArUco library is applied, so as to detect the marker and extract the pose of the camera with respect to a specific visual marker. Moreover, this package utilizes the *tf* ROS package in order to take and publish the pose of Ava's robot (Fig. 6.1) with respect to a stereo camera. Thus, within this way the package achieves visual servoing. Finally, it provides a generation of AR markers with a given size.

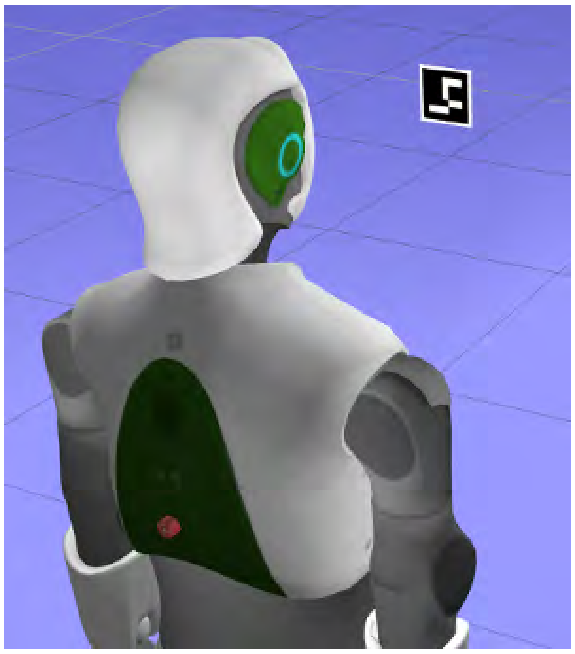


Figure 6.1: The robot of the pal_robotics.

6.1.2 ROS package aruco_detect

This package [22] has been developed by the *Ubiquity Robotics* team. It provides a node which finds ArUco markers in images stream and publishes their vertices (corner points), as well as the $3D$ pose estimation of the camera with respect to the fiducials. It is based on the Aruco contributed module to OpenCV. Similarly with the package that is provided by Ava group (subsection 6.1.1), it uses the *tf* ROS package to publish the translation and rotation. However, it publishes only the pose of the camera with respect to a marker and not to another frame of reference (e.g a robot).

6.1.3 ROS package asr_aruco_marker_recognition

This package [23] has been developed by the *Active Scene Recognition* team of the Humanoids and Intelligence Systems Lab (HIS), Karlsruhe Institute of Technology (KIT). It contains a recognition system for square, binary 2D-markers using the ArUco library. It can be used with a monocular or stereo camera system. Moreover, if a stereo camera system is used, the found markers are further processed. More specifically, the final marker pose is calculated based on the positions of the markers in the left and right image. That is to say, if a marker with the same id was found in each of them, the 3D corner points are calculated using triangulation of the 2D corner points in each image. Then, the published pose is calculated using ICP (Iterative Closest Point: is an algorithm employed to minimize the difference between two clouds of points and is used to reconstruct 2D or 3D surfaces from different scans, to localize robots and achieve optimal path planning). Finally, it provides marker image generation similarly to the package in the subsection 6.1.1.

6.2 Main differences

First of all, the above packages publish poses, but with rotations represented by quaternions rather than in euler angles format, as this thesis follows. It is crucial to extract euler angles rather than quaternions, since this thesis uses the DroneKit's API in order to guide the quadcopter within the commands that have been represented. Although someone could post-processing their outputs in order to transform quaternions to euler angles, the extracted euler angles will not to be the same. The last paragraph answers this statement.

Secondly, the first two packages (subsections 6.1.1, 6.1.2) make use of the tf ROS package. This package lets the user keep track of multiple coordinate frames over time. In other words, tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors between any two coordinate frames at any desired point in time. Thus, the first package (subsection 6.1.1), uses the tf in order to take the relationship between the body-robot coordinate frame and the stereo camera, so as to obtain the pose of the robot with respect to the marker.

However, in order to take advantage of the tf package (i.e to build the tree structure that holds all the relationships between coordinate frames), the description of the robots must be in a URDF format (Unified Robot Description Format is an XML file format used in ROS to describe all elements of a robot) rather than the

SDF format which is used by the Gazebo Simulator. Nevertheless, Gazebo deals with this issue by adding several elements to the URDF files, in order to work properly inside its simulator. In other words, Gazebo has the ability to read URDF files instead of SDF files within some modifications.

On the other hand, included models in the repository [20] (as discussed in the subsection 4.1.5), are in the SDF format. To deal with this problem, this thesis does not use the *tf* package to take the transformation between the body-quadcopter frame and the camera frame, and does an alternative as described in paragraph *"Different Coordinate Systems"* within the subsection 4.3.2. In other words, this thesis, is post-processing the pose of the camera, which the ArUco library returns, so as to take the pose of the quadcopter with respect to the marker. Thus, a user who desires to simulate a quadcopter, which does image processing in order to detect ArUco visual markers in Gazebo using an APM autopilot with SITL and ROS, will struggle to achieve it by using the above packages if he lacks of URDF model files and a knowledge about transformations between different coordinate frames.

Furthemore, the tf package, also, provides built-in functions to transform between rotation representation (i.e rotation matrix to quaternions, quaternions to axis-angle e.t.c). Since the ArUco library makes use of the OpenCV library, as discussed in this thesis, the returning rotation of the camera with respect to the marker is a rvec vector (see subsection 3.3.1). Thus, the first two packages (subsections 6.1.1, 6.1.2) in order to publish the final pose in quaternions form, they make use of the built-in functions that *tf* package provide. Tf package, also, provides built-in function to transform rotations from rotation matrix to euler angles but the user can not control neither the rotation sequence nor the axes of a body frame within those functions (see subsection 3.3.3), so he lacks of freedom to choose. Thus, this thesis, does not use built-in functions of the tf package and creates its own functions in order to generate euler angles, with the sequence of rotations and the axes of the body frame (quadcopter) as explained in previous chapters.

Chapter 7

Conclusions

7.1 Summary

The point of this thesis is the implementation of the detection and approachment of the visual markers with Drones within the Gazebo environment using ROS. The image processing is being executed by a ROS node, which is a subscriber and a publisher simultaneously, using the ArUco and OpenCV library. Moreover, during the extraction of the Drone's pose, a critical requirement of the localization of the drone and the approach of the target, pose data are being published to the ROS ecosystem. A second ROS node, now being a subscriber, grasps the publishing pose and makes use of the DroneKit API to communicate with the APM autopilot in order to guide the Drone safely.

In order to achieve this functionality, we had to fetch a repository [20], that provides SDF model files thus making it possible to represent the quadcopter and the gimbal in the Gazebo environment. This repository is recommended by the ArduPilot community and achieves the communication between the ArduPilot SITL and the Gazebo (Physics) Simulator. Finally, in order to make the Gazebo work with the ROS (for publishing Gazebo's virtual camera stream to a ROS topic), small modifications had to be made to the gimbal model SDF file and attach the ros_gazebo_camera plugin as well as the joint link of the quadcopter model SDF file with the gimbal model SDF file.

7.2 Possible improvements

Obvisously, there is still room for future improvements for increasing the performance of the system. First of all, what can be done is to test the system with a camera that has a better *FoV* (Field of View) and *image resolution* so that the Drone can search for visual markers at a larger distance. Secondly, the calibration for estimating the fundamental parameters of the camera can be improved and the re-projection error between the 2D image points and 3D word points must be decreased, so that the detection of the marker will be more accurate. Moreover, the SDF files can be transformed to URDF files so the *tf* ROS package can be used, for maintaining the relationship between the body-quadcopter coordinate frame and the camera coordinate frame in a tree structure. That being said, the camera can change its orientation in real-time (not pointing e.g always straight down or straight forward, but for changing its orientation within a flight searching mission) and the quadcopter can still extract its pose with respect to it. To conclude, one last modification that can be achieved is making the system run in real world conditions rather than the Gazebo environment.

Bibliography

[1] Francisco J. Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. "Speeded up detection of squared fiducial markers". In: *Image and Vision Computing* 76 (2018), pp. 38–47. ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2018.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0262885618300799.

[2] S. Garrido-Jurado et al. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014), pp. 2280–2292. ISSN: 0031-3203. DOI: http://dx.doi.org/10.1016/j.patcog.2014.01.005. URL: http://www.sciencedirect.com/science/article/pii/S0031320314000235.

[3] S. Garrido-Jurado et al. "Generation of fiducial marker dictionaries using mixed integer linear programming". In: *Pattern Recognition* 51 (2016), pp. 481–491. ISSN: 0031-3203. DOI: http://dx.doi.org/10.1016/j.patcog.2015.09.023. URL: http://www.sciencedirect.com/science/article/pii/S0031320315003544.

[4] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. Sept. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[5] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*. Kobe, Japan, May 2009.

[6] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA Workshop on Open Source Software*. 2009.

[7] *Ros.org. (2018). ROS.org — Powering the world’s robots.* `http://www.ros.org/`.

[8] J. Weng, P. Cohen, and M. Herniou. “Camera calibration with distortion models and accuracy evaluation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.10 (Oct. 1992), pp. 965–980. ISSN: 0162-8828. DOI: `10.1109/34.159901`.

[9] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

[10] Itseez. *Open Source Computer Vision Library.* `https://github.com/itseez/opencv`. 2015.

[11] *The OpenCV Reference Manual.* 2.4.9.0. Itseez. Apr. 2014.

[12] S. Birchfield and C. Tomasi. “A pixel dissimilarity measure that is insensitive to image sampling”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.4 (Apr. 1998), pp. 401–406. ISSN: 0162-8828. DOI: `10.1109/34.677269`.

[13] Z. Zhang. “A flexible new technique for camera calibration”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.11 (Nov. 2000), pp. 1330–1334. ISSN: 0162-8828. DOI: `10.1109/34.888718`.

[14] *Camera Calibration and 3D Reconstruction.* `https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html`.

[15] *EuclideanSpace - Mathematics and Computing.* `http://euclideanspace.com/maths/geometry/rotations/`.

[16] *The ArduPilot Open Source Project.* `https://github.com/ArduPilot/ardupilot`.

[17] *Ardupilot.org. ArduPilot Wiki.* `http://ardupilot.org/ardupilot/index.html`.

[18] *The DroneKit API reference. DroneKit — Your Aerial Platform.* `http://python.dronekit.io/automodule.html`.

[19] *The MAVLink ArduPilot Mission Command Messages.* `http://ardupilot.org/planner/docs/common-mavlink-mission-command-messages-mav_cmd.html`.

[20] *Ardupilot Gazebo Plugin and Models by Seunghwan Jo — A flight control engineer.* `https://github.com/SwiftGust/ardupilot_gazebo/`.

[21] *PAL Robotics S.L. Barcelona, Spain — ArUco ROS detection.* `https://github.com/pal-robotics/aruco_ros`.

[22] *Ubiquity Robotics — ArUco ROS detection.* `https://github.com/UbiquityRobotics/fiducials/tree/kinetic-devel/aruco_detect`.

[23] *Active Scene Recognition. DFG - Research Project at Humanoids and Intelligence Systems Lab (HIS), Karlsruhe Institute of Technology (KIT) — ArUco ROS detection.* `https://github.com/asr-ros/asr_aruco_marker_recognition`.