**Fachhochschule Dortmund**

Institut für die Digitalisierung von Arbeits- und Lebenswelten (IDiAL)

# Developing a Rover-Following Application for APPSTACLE

Research Project

*Author*
Daniel Paredes

*Supervisor*
Robert Höttger

February 23, 2019

# Contents

# Abstract

The purpose of this research project is to develop an application for the Rover in-vehicle platform in the context of the APPSTACLE project and the Eclipse Kuksa ecosystem. The application allows a Rover to follow a visual marker without human intervention, this opens the door to more advanced computer vision application.

`OpenCV` libraries and Eclipse Kuksa services are the main building blocks of the application. Experimental results show the accuracy of the application to estimate and reach the target position. The error in the experiments is in the range of 0.4 cm and 0.7 cm.
Moreover, challenge ,issues and possible solutions of the application are outlined.

# Chapter 1

# Introduction

Today, automotive software-intensive systems are developed in silos by each original equipment manufacturer (OEM) in-house [12] and this approach is not suitable for long-term challenges in the industry. For example, big data simulations and computer-aid modeling can lower development costs and speed up time to market, embedded data sensors should enable more precise monitoring of the performance of vehicles and components [9], and the car should be connected to monitor working parts and safety conditions around it, and communicate with other vehicles and with an increasingly intelligent roadway infrastructure [10].

## 1.1 APPSTACLE

APPSTACLE or *open standard APplication Platform for carS and TrAnsportation vehiCLEs* is an international ITEA research project that aims at providing standardized platforms for car-to-cloud connectivity, external cloud or in-vehicle applications and the use of open source software without compromising safety and security [12]. This document describes an in-vehicle application based on the open source software (Eclipse Kuksa) developed throughout the project.

## 1.2 Eclipse Kuksa

The result of APPSTACLE project is *Eclipse Kuksa* and it provides an example tooling stack for the connected vehicle domain [11]. The Eclipse Kuksa ecosystem consists of an in-vehicle platform, a cloud platform, and an app development IDE as shown in Figure 1.1.

It is possible to collect, store and analyze data through the different Kuksa layers of the in-vehicle platform. These layers are: *meta-kuksa* , which adds the kuksa in-vehicle applications into the AGL image; *meta-kuksa-dev*, which contains all extra packages that are useful for the development process but aren't required in the production Image; and *meta-rover*, which holds all the needed packages to enable the development for the Rover [11]. The in-vehicle platform runs on top of *Automotive Grade Linux* or AGL which is an open-source project from The Linux Foundation. The goal of AGL is to develop a GNU/Linux based operating system and a framework for automotive applications [5].


Figure 1.1: Eclipse Kuksa Ecosystem [11]

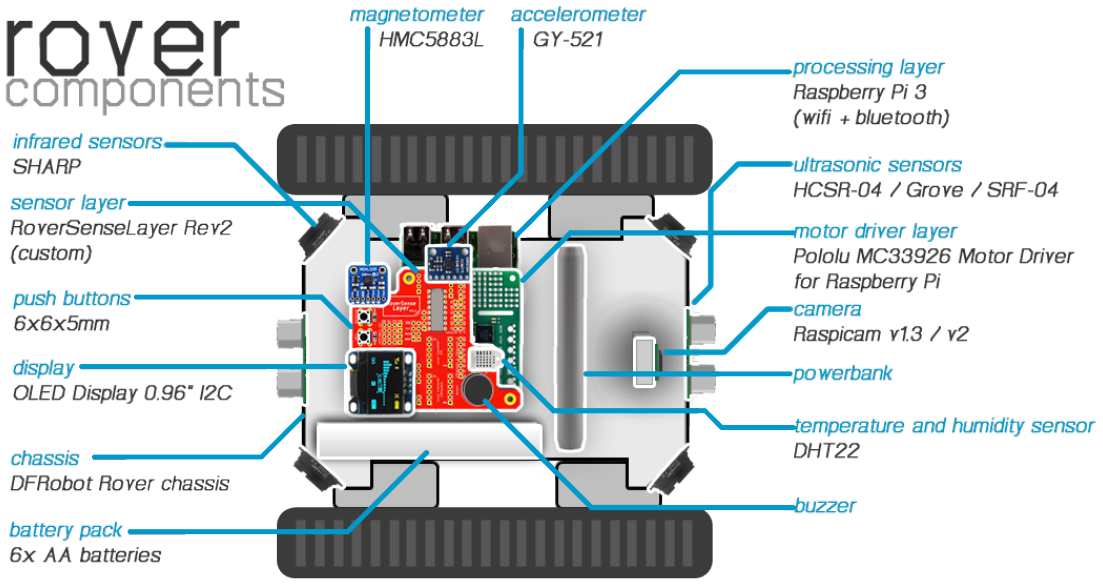



Figure 1.2: Rover Components [1]

The development of Eclipse Kuksa plug-ins or applications can be done using the web-browsed based IDE known as *Eclipse Che*. In order words, a complete toolchain is available as extensions to *Eclipse Che* which allows not only a fast, but also an independent platform development.

In addition, the cloud platform is built on top of other Eclipse frameworks such as Eclipse Hono, which is used in telemetry applications as backend cloud platform; and Eclipse Ditto, which is used to create a digital instance of the in-vehicle platform. It also provides the Kuksa app-store, so users can download an App and deploy it directly in their rovers.

## 1.3 In-vehicle Platform (Rover)

The in-vehicle platform prototype or Rover is based on a raspberry pi 3 that can run *Raspbian Jessie* or *AGL* as Operating system. The available hardware is shown in Figure 1.2. There are two hardware layers besides the raspberry pi. The *Motor driver* layer based on a MC33926 Motor Driver, and the *Sense* layer, a custom made circuit board that is designed as a shield on top of MC33926 [1]. The Sense layer provides interfaces for sensors (accelerometers, magnetometer, infrared, ultrasonic, humidity and temperature), buttons, buzzer and OLED display.

## 1.4 Rover Services

Rover Services are modules that run on Linux-based embedded single board computers. These services provide interfaces to interact with the in-vehicle hardware (sensors, camera, motors, buttons, buzzer) and cloud communication. One can think of services as libraries of an operation system which can be used for software development.

In the context of Eclipse Kuksa, new services can be added using a model-based approach using raml files containing information about hardware to be used and interfaces with its inputs and outputs. The `Raml2agl` tool [4] will generate the basic structure of required C++ files. Once the services are complete, they are compiled and added to the operating system libraries.

An example `RAML` file is shown below:

```
title: Rover Hello World
mediaType: application/json
version: v1
types:
  rover_sensor_id:
    enum:
      - front
      - rear
  rover_demo_id:
    enum:
      - driving
      - infrared
/print_hello_world:
  description: "Service test"
```

In this file is described the version of the new service (v1), rover services with which it interacts (driving and infrared) and its methods (`print_hello_world`).

### 1.5 Rover Application

It is possible to develop applications in Eclipse Kuksa based on rover services, these applications are called rover-apps. The development of rover-apps can be done locally on the user's laptop or desktop or on *Eclipse Che*. In this research project, we developed a rover-app for following a visual marker on another Rover without human intervention. Our application is based on `OpenCV` libraries, and camera, driving and sensor services. In addition, the development of our application was done locally because `OpenCV` is not integrated on the AGL software development kit embedded in Eclipse Che toolchain.

Finally, let us explain the organization of this research project. In chapter 2, we present the design and implementation of our rover-app with some theoretical background. In chapter 3, we show the experimental results. In chapter 4, we describe challenges and issues. At the end of the document, we summarize the conclusions.

# Chapter 2

# Design and Implementation

In this research project we develop an example of a rover-app. For the use case of our application we are using two *Rovers roles*, a **Rover Leader**, and a **Rover Follower** running our rover-app. Hereinafter, we will only use **Leader** or **Follower** to refer to them. The Leader has a visual marker, the Follower should detect it, estimate the angle $\beta$ and distance $d$ with respect to the Leader as is shown in Figure 2.1, and follow the Leader without any human intervention.

This chapter will focus on the design and development of our rover-app. In addition, we present the theoretical background required to understand design decisions. The following sections will describe the main requirements, camera calibration, pinhole model and pose estimation theory, and implementation details.

## 2.1 Requirements

The main requirements for the rover Follower are listed in Table 2.1.

Table 2.1: Most importante requirements

| Requirement | Description |
|---|---|
| 01.Detect visual marker | The Follower should detect the predefined visual marker, created by the ArUco library [3], using the PiCamera mounted on it. |
| 02.Estimation angle and distance | The Follower should estimate the angle and distance to the marker every time the Leader moves to another position. |
| 03.Follower driving | The Follower should steer based on the estimated angle and distance. |
| 04.Autonomous driving | The Follower should be completely autonomous. Detection of the marker and driving should be done with no human intervention other than turning the rover on and initial positioning. |
| 05.Operating system | The Follower should run Raspbian Jessie as operation system. |

| Requirement | Description |
|---|---|
| 06.OpenCV library | The Follower should use OpenCV 3.4.1 [3] for the video processing including reading video frames, marker detection, and estimation of the angle and distance to the marker. |
| 07.Rover-App library | The Follower's code should be based on the services such as sensor reading and driving provided by the rover-app library. |
| 08.Maintainability | The Follower's code should be maintainable following principles of modularity and encapsulation, and avoiding code duplication code. |
| 09.Reusable | The Follower's code should be reusable. Subroutines or functions should be well defined, and its design should take into account orthogonality and extensibility. |
| 10.Understandability | The Follower's code should be understandable. Comments should be relevant, variable and function names should be self explanatory, the code sections should be well defined (includes, global/static/volatile variables declarations, function definitions, main function). |

## 2.2 Camera calibration and the pinhole model

Camera calibration is a necessary step in 3D computer vision in order to extract metric information from 2D images [14]. The calibration process is based on the pinhole camera model shown in Figure 2.2. A 3D object (pyramid) is projected first on a scene plane, and then on the image plane. Each point in the scene plane or *world frame* has its correspondence in the image plane or *camera frame*. The distance from the pinhole to the image plane is called focal length.

The mathematical model of a pinhole camera can be derived using linear algebra and the visual representation shown in Figure 2.2.

Let's denote a 2D point $\hat{\mathbf{m}} = [x, y, 1]^T$, a 3D point $\hat{\mathbf{M}} = [X, Y, Z, 1]^T$, there exists a camera projection matrix $\mathbf{P}$ such that $\hat{\mathbf{m}} = \mathbf{P}\hat{\mathbf{M}}$. The camera projection matrix can be decomponsed in two matrices $\mathbf{A}$ and $[\mathbf{R} \quad \mathbf{t}]$.

$$\hat{\mathbf{m}} = \mathbf{P}\hat{\mathbf{M}} = \mathbf{A}[\mathbf{R} \quad \mathbf{t}]\hat{\mathbf{M}} \tag{2.1}$$

The camera intrinsic matrix $\mathbf{A}$ contains information about the internal parameters of the camera: focal length, image sensor format and principal point or image center. This matrix is the output of the calibration process. The coordinates of the principal point is described by $(x_0, y_0)$, $\alpha_x$ and $\alpha_y$ represent the focal length in terms of pixels on the axis $x$ and $y$, and $\gamma$ is the skew of the image.

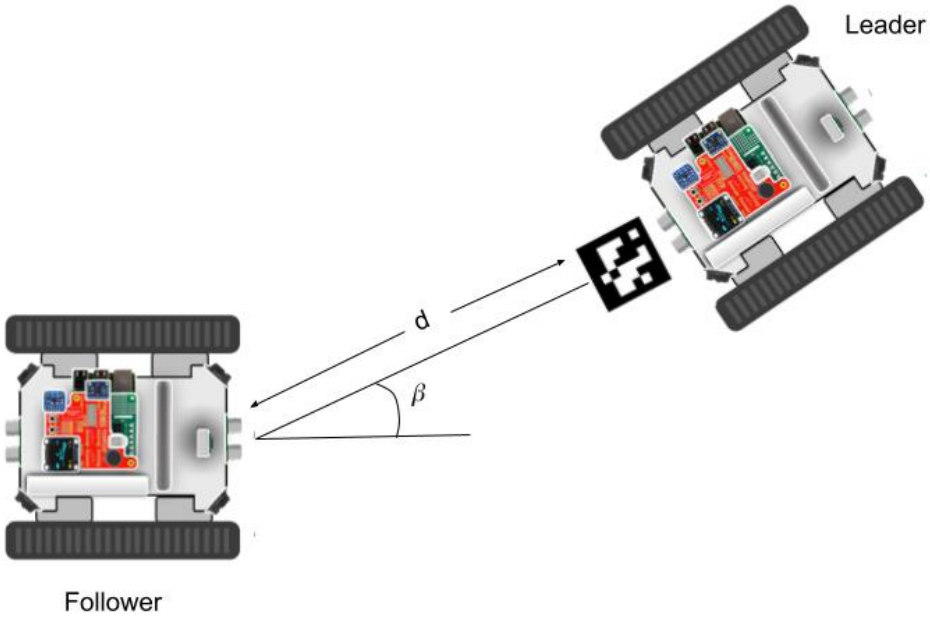



Figure 2.1: Rover Use Case

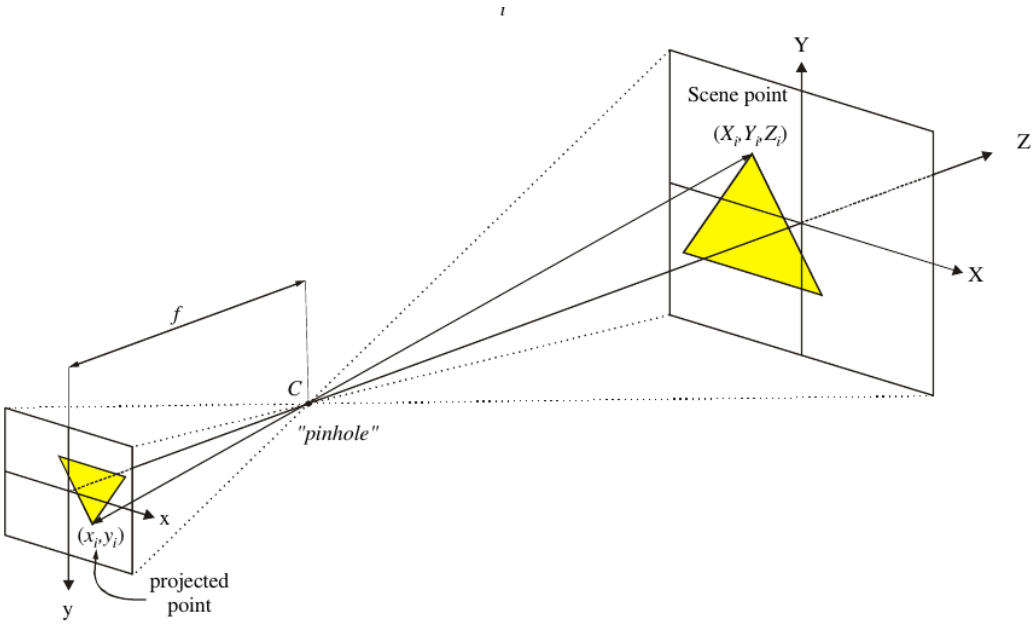



Figure 2.2: The pinhole imaging model [7].

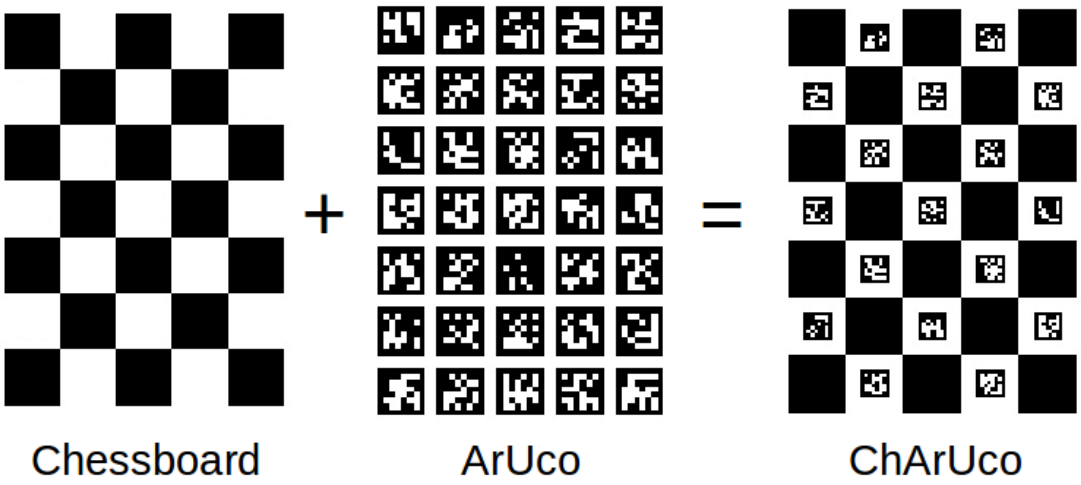



Figure 2.3: Planar Patterns [3]

$$\mathbf{A} = \begin{bmatrix} \alpha_x & \gamma & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

The camera extrinsic matrix are given by the rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$ which are used to project an image on the world frame to camera frame. Moreover, the scale transformation is given by $\alpha_x$ and $\alpha_y$. In the following section we will describe how to extract relevant information such as the rotation angles.

The camera calibration is performed using `OpenCV`. This library implementation is based on the technique described by [15] and the matlaparameters parameters b implementation done by [8]. parameters The calibration technique in [15] requires the camera to observe a planar pattern, usually a chessboard pattern, at different orientations. The more samples, the better the estimation of the intrinsic parameters. The calibration algorithm minimizes the reprojection error, which is the distance between observed feature points on the planer pattern and the projected using the estimated parameters.

For calibration, we used a *ChArUco* board instead of the classical chessboard because it generates a better estimation of the parameters [3].

The procedure to calibrate the PiCamera is straightforward with `OpenCV` and the sample codes found under `opencv_contrib-3.4.1/modules/aruco/samples`. The following list is necessary in order to perform a calibration process:

1. Create a charuco board, print it and paste it on a solid and planar surface.
2. Compile the example code `calibrate_camera_charuco.cpp` and run it
3. Place your pattern in different orientations and take pictures
4. When you are done, just close the program

In our case, the camera intrinsic matrix $\mathbf{A}$ is as following:

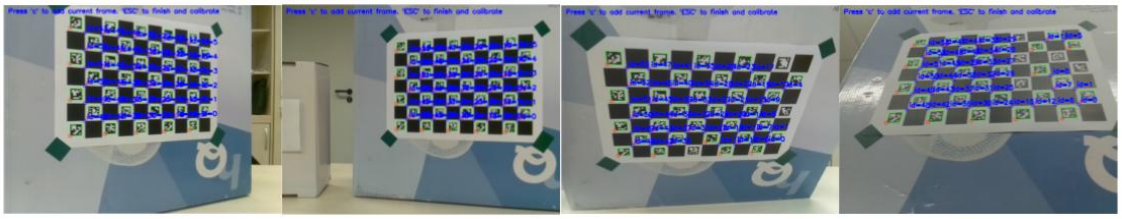


Figure 2.4: Some camera views used for calibration

$$\mathbf{A} = \begin{bmatrix} \alpha_x & \gamma & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 6.125e+02 & 0. & 3.216e+02 \\ 0 & 6.122e+02 & 2.365e+02 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

## 2.3 Rotation matrix to Euler angles

As mentioned before, the camera extrinsic parameters are given by the rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$. A rotation matrix can be formed as the product of three rotations around three cardinal axes, e.g., $x$, $y$, and $z$, or $x$, $y$, and $x$. This is generally a bad idea, as the result depends on the order in which the transforms applies [13].

However, a rotation can be also represented by a rotation axis $\mathbf{k} = [k_x, k_y, k_z]^T$ and an angle $\theta$, or equivalently by a vector $\omega = \theta\mathbf{k}$. In order to do the transformation from axis-angle representation to rotation matrix, the cross-product matrix $\mathbf{K}$ and Rodrigues' rotation formula can be used.

$$\mathbf{K} = \begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix} \tag{2.4}$$

$$\mathbf{R} = \mathbf{I} + (\sin\theta)\mathbf{K} + (1 - \cos\theta)\mathbf{K}^2 \tag{2.5}$$

In order to get the angles related a rotation whose yaw, pitch and roll angles are $\phi$, $\rho$ and $\psi$. These angles are rotations in $z$, $y$ and $x$ axis respectively. We will rotate first about the $x$-axis, then the $y$-axis, and finally the $z$-axis. Such a sequence of rotations can be represented as the matrix product

$$\mathbf{R} = R_z(\phi)\, R_y(\rho)\, R_x(\psi) \tag{2.6}$$

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \tag{2.7}$$

$$R_y(\rho) = \begin{bmatrix} \cos\rho & 0 & \sin\rho \\ 0 & 1 & 0 \\ -\sin\rho & 0 & \cos\rho \end{bmatrix} \tag{2.8}$$

$$R_z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

Following the sequence of rotations presented above and the algorithm described by [6], the angles can be found using algorithm 1.

**if** $R_{31} \neq \pm 1$ **then**
  $\phi = \arctan 2(R_{21}, R_{11})$
  $\rho = -\arcsin(R_{31})$
  $\psi = \arctan 2(R_{32}, R_{33})$
**else**
  $\phi = 0$
  $\rho = -R_{31}\pi/2$
  $\psi = \arctan 2(-R_{23}, R_{22})$
**end**

**Algorithm 1:** Slabaugh's algorithm

## 2.4 Implementation details

According to Requirement 04, the Follower should be completely autonomous. In order to do so, the Follower captures video frames, processes the frames with `OpenCV` a and generate movement based on the processed data.

### Video Processing with OpenCV

We use `OpenCV` and the submodule `aruco` to capture video and process the frames in order to extract information from the visual markers (Requirement 01, 02). To estimate and detect the marker at the beginning we should load the camera intrinsic parameters, saved in a YAML file, and the Aruco dictionary, composed by 250 markers and a marker size of 6x6 bits [3], to memory. These steps are shown in following code.

```
// cameraMatrix: camera intrinsic parameters
// dictionary: aruco dictionary
cv::FileStorage fs("calibration.yml", cv::FileStorage::READ);
fs["camera_matrix"] >> cameraMatrix;
cv::Ptr<cv::aruco::Dictionary> dictionary =
     cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);
```

Given a video frame, it is possible to detect Aruco markers if they are visible. When the marker is detected, we extract the four corners of the 7cm width marker using `cv::aruco::detectMarkers` function. The first corner is the top left corner, followed by the top right, bottom right and bottom left. The next step is to estimate the extrinsic camera parameters, which means the rotation vector $\omega$ and the translation vector $\mathbf{t}$. The size of the marker is an input parameter of the `OpenCV` function `cv::aruco::estimatePoseSingleMarkers`. Finally, we calculate the Euler angles by using function `cv::Rodrigues` and Slabaugh's algorithm, described in the previous section. The `cv:Rodrigues` function is a direct implementation of equations 2.4 and 2.5. Thus, a basic code for extracting the features from the marker is as follows:

```
// Initialization
cv::VideoCapture inputVideo(0);
cv::FileStorage fs("calibration.yml", cv::FileStorage::READ);
fs["camera_matrix"] >> cameraMatrix;
fs["distortion_coefficients"] >> distCoeffs;

inputVideo.open(0);
cv::Ptr<cv::aruco::Dictionary> dictionary =
    cv::aruco::getPredefinedDictionary(cv::aruco::DICT_6X6_250);
...

// Video processing
inputVideo.read(image);
cv::aruco::detectMarkers(
    image, dictionary, corners, ids);
cv::aruco::estimatePoseSingleMarkers(
    corners, 0.07, cameraMatrix,
    distCoeffs, rvec, tvec);
cv::Rodrigues(rvec, rmat);
rotationMatrixToEulerAngles(rmat, angles);
```

An example is shown in Figure 2.5. The Euler angles are $\psi = 165$, $\rho = 25$ and $\psi = 0$. The green, red and blue axes correspond to the X-axis, Y-axis and Z-axis respectively. As expected from the pin hole model in Section 2.2, $\psi$ is near 180 because the image is facing the camera as result the blue axis points towards the camera.

However, the estimated angles can not be used directly because estimations have small errors. Figure 2.6 shows the values values of the rotation angles in a span of 1000 samples and in Table 2.1 shows the statistics of those samples. The ground truth values for Euler angles were $[0, 0, 0]$, and for distance were 47.5 and 16 centimeters respectively.

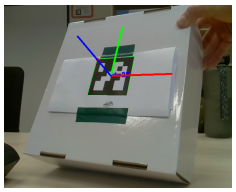


Figure 2.5: Camera axis

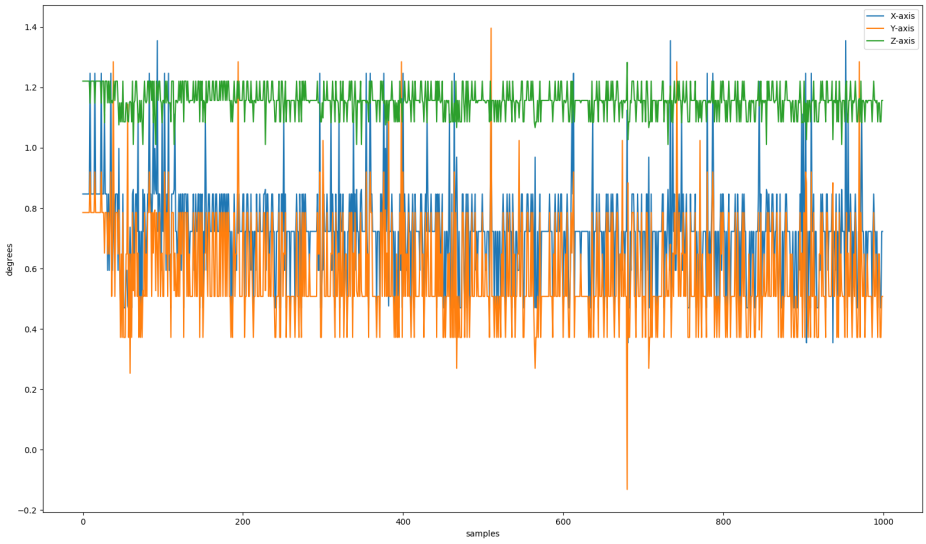



Figure 2.6: Angles in X-axis $\psi$, Y-axis $\rho$ and Z-axis $\phi$

Table 2.2: Statistics of estimated Euler angles and distance to visual marker

| Data | Mean | $\sigma$ | Median |
|---|---|---|---|
| $\psi$ deg | 0.719853 | 0.162262 | 0.723000 |
| $\rho$ deg | 0.584508 | 0.165382 | 0.507618 |
| $\phi$ deg | 1.155499 | 0.046940 | 1.157000 |
| $d$ cm | 45.51706 | 0.033564 | 45.50772 |
| $d$ cm | 15.65339 | 0.007401 | 15.65401 |

The results of the standard deviation $\sigma$ from table 2.2 suggest the estimated values can be stable ($\sigma < 0.16$ deg) overall, particularly in the case of distance to the marker ($\sigma < 0.04cm$). However, Figure 2.6 suggests the existence of peak values, thus we must filter the samples in order to minimize the effect of those outliers.

A median filter is highly effective removing outliers from data, but requires to save chunks of data in memory. However, the results showed that the mean and the median of Euler angles are similar, thus it is reasonable to think that outliers have small influence on the data. In other words, the mean filter is a simple and effective option against outliers problem. Its implementation is straightforward and requires no memory to save previous values. A pseudocode is as follows:



**Rover rotations**

Rover is a ground vehicle which means that it only steers in one axis. Thus, only rotations

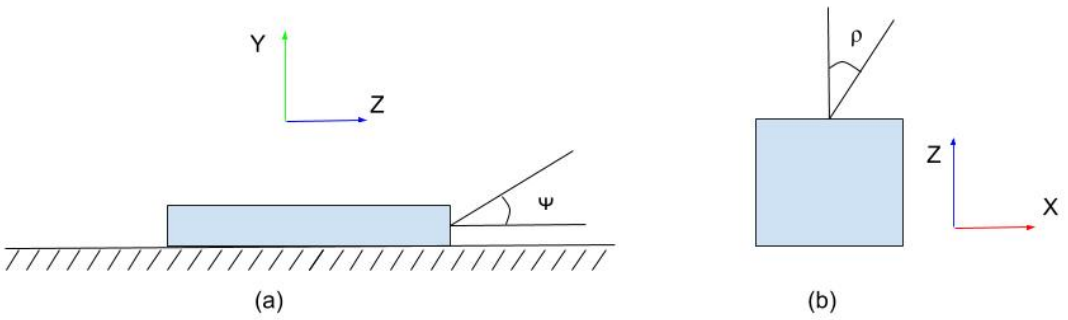



Figure 2.7: Rotation rotations (a) Rotation in X-axis (b) Rotation in Y-axis

words, the only relevant information from the estimated Euler angles is $\rho$, or the angle related to the Y-axis.

**Measuring angular displacement**

In order to move the Follower to a defined angular position, the CY-521 board is used. The CY-521 has an accelerometer and a gyroscope. The accelerometer works by measuring the components of gravity in the different axis, taking the "earth" or gravity acceleration as reference. On the other hand, the gyroscope measures angular speed relative to itself or its own rotation, using the inertial force called the Coriolis effect.

With that information, we could estimate the angular position of Follower. However, the values from the accelerometer are not taken into account because the gravity vector is parallel to the Y-axis. It is important to note that we want to measure relative rotations, thus in the initial position the angle will always be 0.

The gyroscope measures angular speed in all axes, in particular the angular speed in the Y-axis or $\omega_y$. The angular displacement $\rho$ is just the integral of $\omega_y$.

$$\rho = \int \omega_y(t)dt \tag{2.10}$$

$$\omega_y = \frac{\delta\rho}{\delta t} \tag{2.11}$$

Nonetheless, the calculation is done in a computer, thus we use the *Forward Euler Method* to solve the integral.

$$\rho[n+1] = \rho[n] + \Delta t\omega_y[n] \tag{2.12}$$

where $\Delta t$ is the sampling period between sensor readings and $\rho[0] = 0$. A pseudocode of the rotation routine is as follows:

```
current_angle = 0;

// clockwise rotation
```

```
if( desired_angle <=0 ) turnRight();
// counterclockwise rotation
else turnLeft();

while( abs(current_angle - desired_angle) > 0)
        wait(sampling_period);
        current_angle += getAngle()*sampling_period;

stop();
```

**Camera Projection and rover driving**

As mentioned at the beginning of this chapter, in the pin hole camera model, described in Section 2.2, the 2D point $\hat{\mathbf{m}}$ and the 3D $\hat{\mathbf{M}}$ are related through a projection matrix $\mathbf{P}$ [15], and the camera projection matrix $\mathbf{P}$ is formed by the combination of extrinsic and intrinsic camera parameters. In order to move the Follower to the Leader's position, we must find $\hat{\mathbf{M}}$ given $\hat{\mathbf{m}}$ (Requirement 03).

$$\hat{\mathbf{m}} = \mathbf{P}\hat{\mathbf{M}} = \mathbf{A}[\mathbf{R} \quad \mathbf{t}]\hat{\mathbf{M}} \tag{2.13}$$

$$\mathbf{P} = \overbrace{\mathbf{A}}^{\text{Intrinsic Matrix}} \times \overbrace{[\mathbf{R} \mid \mathbf{t}]}^{\text{Extrinsic Matrix}} \tag{2.14}$$

$$m_{2D} = \mathbf{A}^{-1}\hat{\mathbf{m}} = \mathbf{R}\hat{\mathbf{M}} + \mathbf{t} \tag{2.15}$$

The $m_{2D}$ is used by `OpenCV` to estimated rotation and translation vectors. In addition, $[\mathbf{R} \quad \mathbf{t}]\hat{\mathbf{M}}$ becomes $\mathbf{R}\hat{\mathbf{M}} + \mathbf{t}$ because the last element of $\hat{\mathbf{M}}$ is 1. Si

$$\mathbf{R}^{-1}(m_{2D} - \mathbf{t}) = \hat{\mathbf{M}} \tag{2.16}$$

Since rotation matrix is orthogonal:

$$\mathbf{R}^{T}(m_{2D} - \mathbf{t}) = \hat{\mathbf{M}} \tag{2.17}$$

Thus, the rover Follower first should drive forward and then rotate in order to reach to Leader's position. On the other hand, since the Follower only rotates in the Y-axis, we only need to change the direction of the rotation and translation.

$$R_y(\rho) = \begin{bmatrix} \cos\rho & 0 & \sin\rho \\ 0 & 1 & 0 \\ -\sin\rho & 0 & \cos\rho \end{bmatrix} \tag{2.18}$$

$$R_y(\rho)^T = R_y(-\rho) \tag{2.19}$$

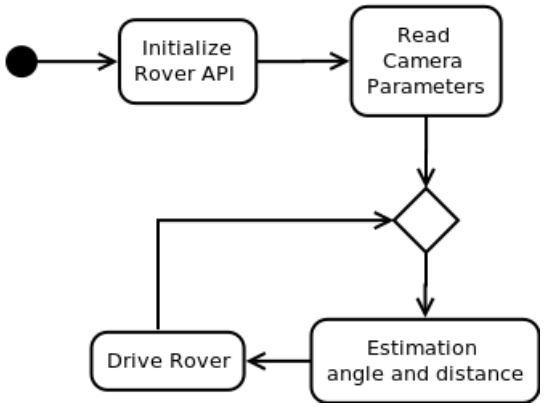



Figure 2.8: Activity diagram

**Implementation**

The activity diagram of use case is shown in Figure 2.8. First, the rover API is initialized, it also includes the motor and sensors, and the camera intrinsic parameters are load into memory as described before.

```
RoverBase r_base; /* Rover API */
RoverDriving r_driving;  /* Rover driving service */
RoverGY521 r_accel; /* gyro and accelerometer */
/* Ultrasonic sensors */
RoverHCSR04 r_front = RoverHCSR04(ROVER_FRONT);
RoverHCSR04 r_rear  = RoverHCSR04(ROVER_REAR);
```

After the initial set up, an infinity loop starts. During the loop, we estimate the angle $\rho$ and distance $d$, the latter is the norm of the translation vector $d = \|\mathbf{t}\|_2$. The motion is done in two steps: rotation and translation. The Follower goes forward $d$ centimeters, and when it is done, it rotates $\rho$ degrees. The current distance is measured using the front ultrasonic ranging module HC-SC04. The rover-API can only give accurate information for distances below 40 cm [2], for distances greater than 40cm the API always returns 40cm. However, when the Follower approaches the Leader, it will eventually be in the measurable range.

Once the Follower reaches the Leader, it stops and waits until the Leader moves again. A basic code of the loop is as follows:

```
while(1){
    // Initialization of the current values
    estimated_angle = 0;
    estimated_distance = 0;
```

```
    // Mean filter
    for(i=0; i<nSamples; i++){
        readFrame();
        [rvec, tvec] = getExtrinsicParameters();
        estimated_angle += getYrotation(rvec);
        }
    estimated_angle /= nSamples;
    estimated_distance = norm(tvec);

    // Driving routines
    moveForwoard(estimated_distance);
    rotateNdegrees(estimated_angle);
}
```

# Chapter 3

# Experimental Results

In this chapter, we evaluate the behavior of the Follower in an controlled environment. We present the results of two test cases: known distance and angle and difference between ultrasonic sensor and camera. Our experiments consisted in setting a known initial and final position, and measure the deviation from expected final position. We measure the deviation using the front ultrasonic sensor, the PiCamera and a ruler with smallest division of the scale of 1 mm, or estimated uncertainty of 0.5 mm.

## 3.1 Known distance and angle

In this test case we evaluated the Follower's accuracy to reach the desired position and orientation. The different setting and initial camera estimations are presented in Table 3.1. Distance values are in centimeters (cm) and angle values are in degrees. For the initial position, the results show that absolute error for both distance and angle is less than 4cm and 3° respectively, and that relative error is less than 5% for distance and less than 9% for angle if we ignore the relative error for 0°.

Table 3.1: Initial settings and estimations

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Distance | 70 | 70 | 70 | 70 | 100 |
| Est. Distance | 73.48 | 71.64 | 73.42 | 71.70 | 103.37 |
| Absolute Error | 3.48 | 1.64 | 3.42 | 1.70 | 3.37 |
| Relative Error | 4.97% | 2.34% | 4.89% | 2.43% | 3.37% |
| Angle $\rho$ | 30° | −30° | 16° | −16° | 0° |
| Est. Angle $\rho$ | 32.43° | −27.47° | 15.78° | −15.73° | −2.84° |
| Absolute Error | 2.43° | 2.53° | 0.22° | 0.27° | 2.84° |
| Relative Error | 8.1% | 8.43% | 1.38% | 1.69% | $\infty$ |

In Table 3.2 presents the results for the final position. The absolute error for distance is less than 0.7cm, and less than 2.75° for angle. On the other hand, the relative error for distance and angle is, in the worst case, 14.17% and 257.89% respectively.

Table 3.2: Measurements and final estimations

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Distance | 4.8 | 4.2 | 8.8 | 5.2 | 5.90 |
| Est. Distance | 5.48 | 4.64 | 9.50 | 5.72 | 6.44 |
| Absolute Error | 0.68 | 0.44 | 0.70 | 0.52 | 0.54 |
| Relative Error | 14.17% | 10.48% | 7.95% | 10.00% | 9.15% |
| Aprox. Angle $\rho$ | 1.54° | 2.75° | 1.71° | 0.95° | 2.10° |
| Est. Angle $\rho$ | 3.58° | 5.50° | 2.62° | 3.40° | 2.25° |
| Absolute Error | 2.04° | 2.75° | 0.91° | 2.45° | 0.15° |
| Relative Error | 132.47% | 100% | 53.22% | 257.89% | 7.14% |

Relative error values in the final position are bigger than in the initial position because the values of distance and angle are lower. However, absolute error values have improved in the case of distance, reducing from 3.48cm to 0.7cm, and have been kept stable in the case of angle.

## 3.2 Difference between ultrasonic sensor and camera

Ultrasonic sensor are commonly used to measure the distance to an object. We wanted to compare the accuracy of the ultrasonic ranging module HC-SC04 with PiCamera. In table 3.3 are presented the results of our experiments. The results show the chosen ultrasonic sensor is useless for distances larger than 40cm, while camera estimations have absolute errors lower than 1.52 cm. In [2] it is mentioned that for distances greater than 40cm, the sensor readings are not reliable. On the other hand, for distances smaller than 40cm, both ultrasonic readings and camera estimations presented absolute errors lower than 0.72cm and 1cm respectively.

Table 3.3: Accuracy comparison between ultrasonic ranging module HC-SC04 readings and PiCamera estimations.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Distance | 70 | 50 | 20 | 15 | 13 | 4.8 |
| PiCamera | 71.52 | 51.24 | 20.32 | 15.41 | 13.72 | 5.48 |
| Absolute Error | 1.52 | 1.24 | 0.32 | 0.41 | 0.72 | 0.68 |
| Relative Error | 2.17% | 2.48% | 1.60% | 2.73% | 5.54% | 14.17% |
| Ultrasonic | 40.00 | 40.00 | 20.00 | 14.00 | 13.00 | 5.00 |
| Absolute Error | 30 | 10 | 0 | 1 | 0 | 0.2 |
| Relative Error | 42.86% | 20.00% | 0% | 6.67% | 0% | 4.17% |

# Chapter 4

# Challenges and issues

In this chapter we will describe open challenges and unresolved issues that should be addresed or taken into account in further versions of the described rover-app. We describe the following issues and challenges: extrinsic parameters estimation, limited camera vision field, marker maximum detectable rotation, error measurements in the current distance sensing system, and the migration from Raspbian to AGL.

## 4.1 Camera vision field

The PiCamera mounted on the Rover has a limited vision field. As observed in Figure 4.1, the PiCamera can only sense from $-33$ degrees up to 28 degrees for a radius of 50cm, in other words the vision field has a $-33$ degree limit on the left side and a 28 degree limit on the right side. We could consider the vision field as a trapezoid as in Figure 4.2.
Thus, the closer the marker to the camera the smaller the vision field due to pinhole model and image projections. In addition, the limits are not the same because the PiCamera is not perfectly aligned with Rover axis.

## 4.2 Maximum detectable rotations

There is a maximun $\rho$ angle in clockwise and counterclockwise directions that can be detected. In Figure 4.3 the position and angle in which the marker can still be detectable is shown. For counterclockwise rotations the maximum $\rho$ angle is $-77$ degrees and for clockwise rotations is 72 degrees.

## 4.3 Problems measuring distance traveled

There is no sensor mounted on the Rover that accurately measures the distance traveled, such as GPS or encoders. We are using an ultrasonic sensor to measure the distance to the marker, however this approach only works when the rover Leader is within a 40cm radius and its orientation is not greater than 30 degrees given the way that sensors works. For angles greater than 30 degrees the bounced waved is not detectable. Due to the previous explained reasons and the fact that the estimated distance to the marker is known, the

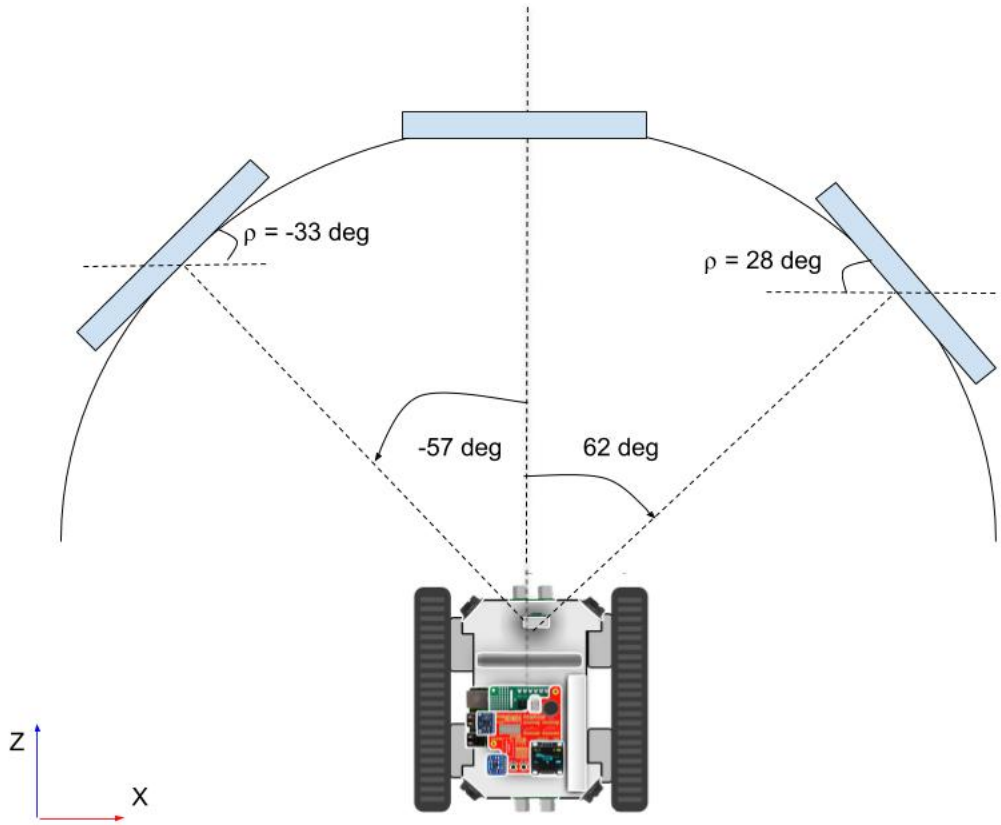



Figure 4.1: Limits of camera vision field

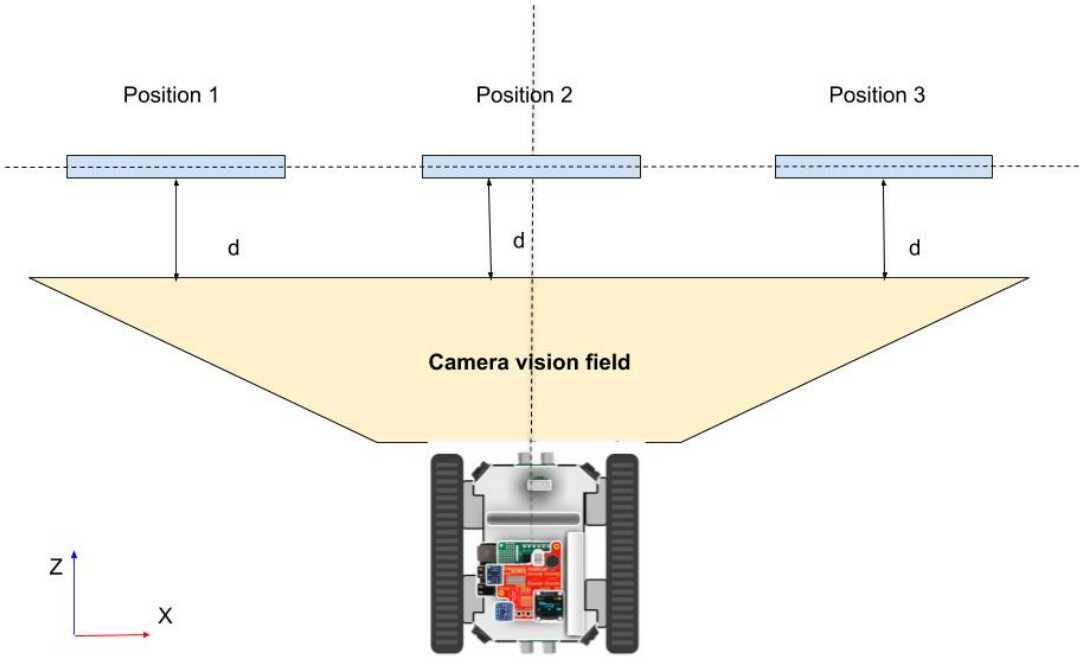



Figure 4.2: Camera vision field

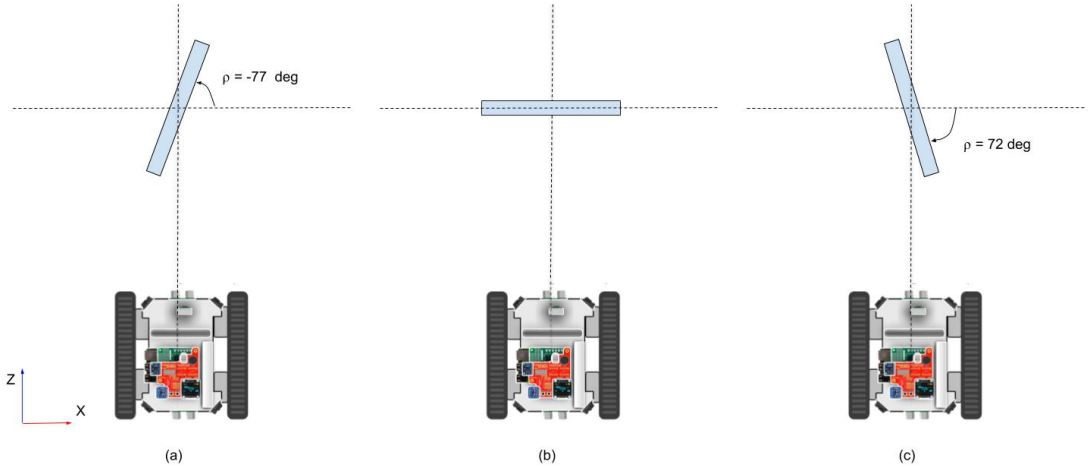


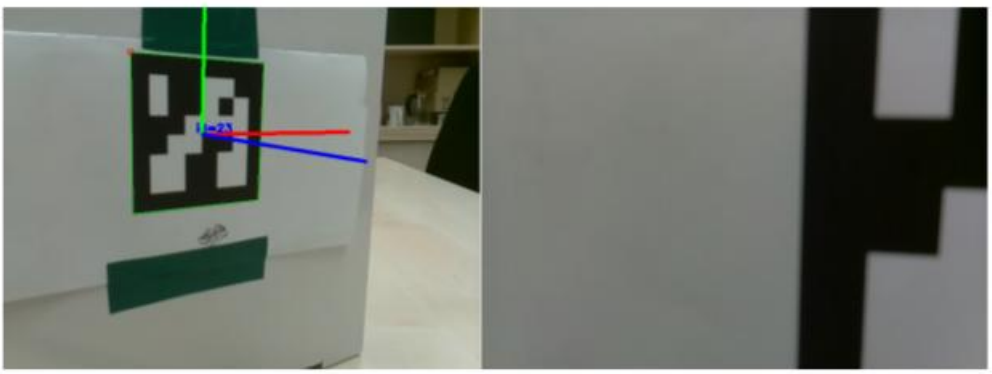

Figure 4.3: Maximum detectable rotations (a) counterclockwise, (b) no rotation and (c) clockwise

use of a sensor to measure distance travelled is needed, and also it will broad the range of possible applications.

In addition, sometimes the Follower is not able to detect the marker anymore since it is very close to it due to the fact that our system does not measure accurately the distance travelled. An example is shown in Figure 4.4. It is observed that at the final position $d = 3cm$, the Follower lost track of the marker. From the experiments, the Follower detected the marker from distances larger than 20cm.

## 4.4 Switching to Raspbian to AGL

The rover-app described in this document runs on top of the Raspbian operation system (Requirement 05). In further improvements the application should run on AGL. However, the current version of the Eclipse Kuksa software development kit (SDK) is not compatible with `OpenCV 3.4.1`, thus this issue should be addressed first.



(a) (b)

Figure 4.4: (a) Initial position $d = 22.73cm$, $\rho = -33.41°$ (b) Final Position

# Conclusions

The design and development of a rover-app has been presented. The application allows a Rover to follow without human intervention a visual marker on another Rover. The described rover-app is based on `OpenCV` libraries and rover-services to interface with camera, motors and sensors mounted on Rover.
Our experimental results show the high accuracy of our application to estimate the distance to the marker, and the existence of an almost constant error in the angle estimation. Moreover, we presented some challenges and issues such as maximum detectable rotations of the visual marker, limited camera visual field, errors measuring distance traveled and migration to AGL.

# List of Tables

# List of Figures

# Bibliography


[1] Eclipse APP4MC. Eclipse app4mc - rover documentation, 2017.
URL: https://app4mc-rover.github.io/rover-docs/.

[2] Eclipse APP4MC. Rover api documentation, 2017.
URL: https://app4mc-rover.github.io/rover-app/.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] Pedro Cuadra. Raml to agl, 2017.
URL: https://github.com/pjcuadra/raml2agl.

[5] The Linux Foundation. Automative grade linux, 2016.
URL: https://www.automotivelinux.org/.

[6] Gregory G. Slabaugh. Computing euler angles from a rotation matrix, 01 1999.
Technical Report URL: www.gregslabaugh.net/publications/euler.pdf.

[7] Rafael Garcia. *A proposal to estimate the motion of an underwater vehicle through visual mosaicking*. PhD thesis, University of Girona. Doctor of Philosophy Girona, 2001.

[8] J.Y.Bouguet. Matlab calibration tool (2018-11-20).
URL:http://www.vision.caltech.edu/bouguetj/calibdoc/.

[9] McKinsey and Company. A road map to the future for the auto industry, 2014.
URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/a-road-map-to-the-future-for-the-auto-industry.

[10] McKinsey and Company. What's driving the connected car, 2014.
URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/whats-driving-the-connected-car.

[11] APPSTACLE Project. Eclipse kuksa, 2016.
URL: https://www.eclipse.org/kuksa/.

[12] APPSTACLE Project. open standard application platform for cars and transportation vehicles, 2016.
URL: https://itea3.org/project/appstacle.html.

[13] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[14] Z. Zhang. Emerging topics in computer vision, chapter 2: Camera calibration. Upper Saddle River, NJ, USA, 2004. Prentice Hall PTR.

[15] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(11):1330–1334, November 2000.