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

December 18, 2018

Contents

Abstract

This document is a skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. The skeleton was implemented in Markdown and can be compiled with Pandoc, using the TEX Live and or the MiKTEX LATEX compiler.

Chapter 1

Introduction

The introductory part contains the analysis of the diploma dissertation, its historical history, the justification of the task (description of the motivation), the solutions so far and the summary of the student's solution.

According to the introductory custom, it is closed with the structure of the diploma, that is, with a brief description of which chapter it deals with.

1.1 APPSTACLE

APPSTACLE or *open standard APplication Platform for carS and TrAnsportation vehiCLEs* is an international ITEA research project that aims at providing standardize platform for car-to-cloud connectivity, external cloud or in-vehicle applications and the use of open source software without compromising safety and security [11]. This document describes an in-vehicle application based on the open source software 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 [10]. 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 throught the different kuksa layers of the in-vehiche platform. These layers are: *meta-kuksa* adds the kuksa in-vehicle applications into the AGL image, *meta-kuksa-dev* contains all extra packages that are useful for the development process but aren't required in the production Image, and *meta-rover* holds all the needed packages to enable the development for the Rover [10]. 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 [6].

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 and independent platform development.

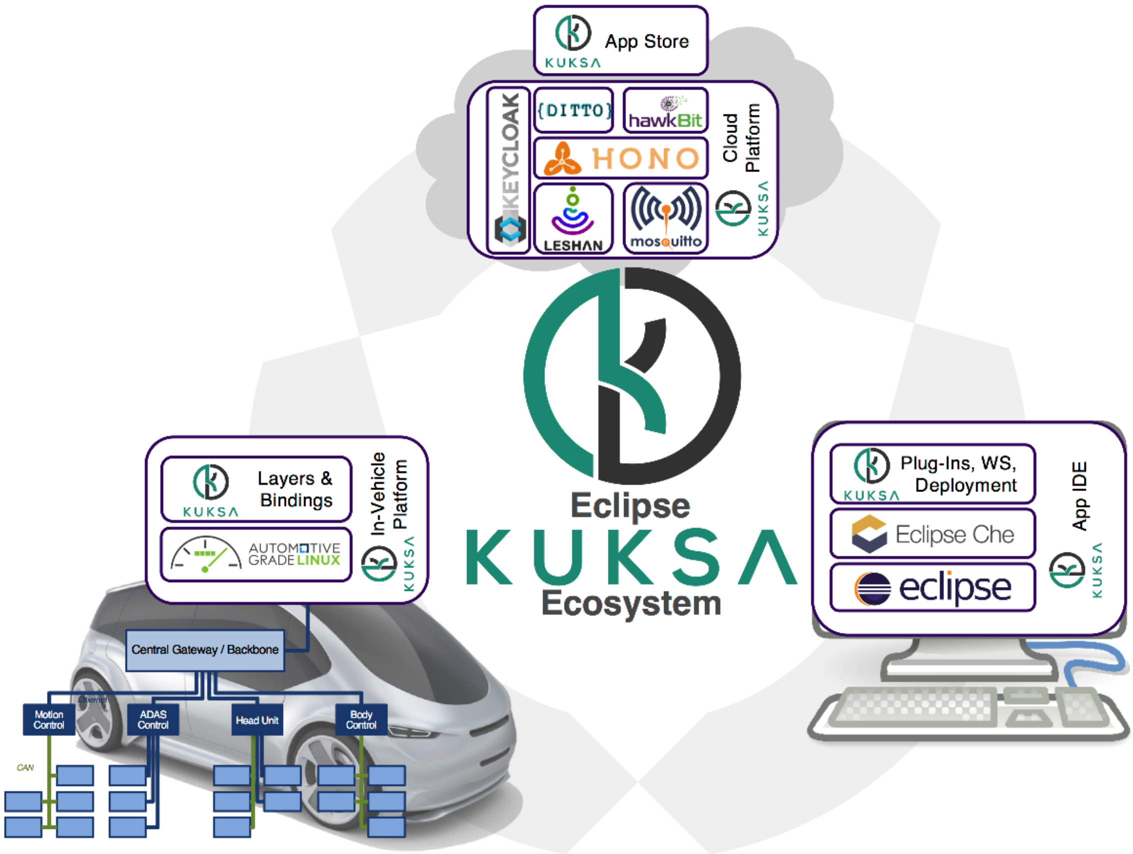


Figure 1.1: Eclipse Kuksa Ecosystem [10]

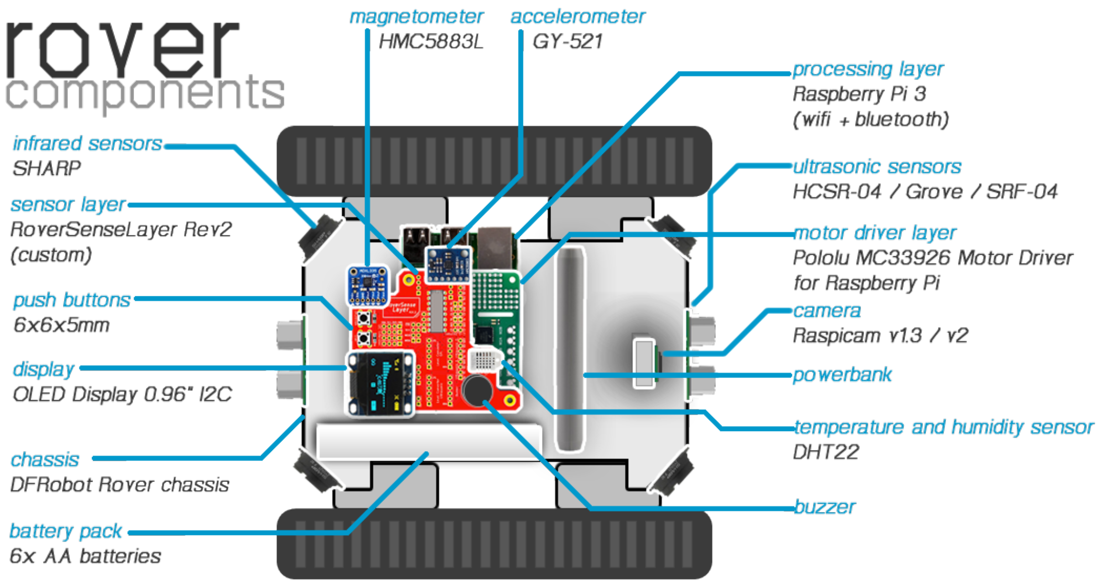


Figure 1.2: Rover Components [1]

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

1.3 In-vehiche Platform (Rover)

The in-vehicle platform 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 layers besides the raspberry pi. The *Motor driver* layer based on a MC33926 Motor Driver, and the *Sense* layer, a customly 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 runs on Linux-based embedded single board computers. These services provide intefaces 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 devlopment.

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 is complete, it's 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"
```

1.5 Rover Application

1.6 Related Work

Chapter 2

Design and Implementation

2.1 Use case description

In this research project we develop a use case for rover-apps. For the use case we are using two *Rovers*, a **Rover Leader** and a **Rover Follower**. 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 β and distance d with respect to the leader as is shown in figure 2.1, and follow the leader.

This chapter will be focused on the development of the behavior of the follower because is the only rover that must be completely autonomous. The following sections will describe requirements, main hardware parts involved and software tools required to implement follower's behavior, camera calibration and pose estimation theory, and software implementation details.

2.2 Requirements

The main requirements for the rover follower are listed in table 2.1.

Table 2.1: Statistics of estimated euler angles and distance to visual marker

Requirement	Description
Detect visual marker	The follower should detect an Aruco Marker using the PiCamera mounted on it. The camera field of vision should contain the marker, and this one should be within 100cm of radius.
Estimation angle and distance	The follower should estimate the angle and distance to the marker everytime the leader moves to another position.
Follower driving	The follower should steer based on the estimated values. It should first rotate based on the estimated angle, and then drive forward based on the estamited distance, but it should stop when is within 5cm of radius from the leader.

Requirement	Description
Leader movement	The leader should only move to a new position when the follower reaches it.
Autonomous driving	The follower should be completely autonomous. Detection of the marker and driving should be done with no human intervention other than turning on the rover and inital positioning.
Operating system	The follower should run Rasbian Jessie VX.X as operation system.
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.
Rover-App library	The follower code should be based on the services such as sensor reading and driving provided by the rover-app library.
Mantainability	The follower code should be mantainable following principles of modularity and encapsulation, and avoiding code duplication code.
Reusable	The follower code should be reusable. Subroutines or functions should be well defined, and it's design should take into account orthogonality and extensibility.
Understanbility	the follower 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).

2.3 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 [13]. If you hold that box in front of you in a dimly lit room, with the pinhole facing some light source you see an inverted image appearing on the translucent plat [5]. 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* will have it's correspondence in the image plane or *camera frame*. The distance from the pinhole to the image plane is called focal lenght.

The mathematical model of a pinhole camera can be devired 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 $\hat{\mathbf{m}} = \mathbf{P}\hat{\mathbf{M}}$.

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

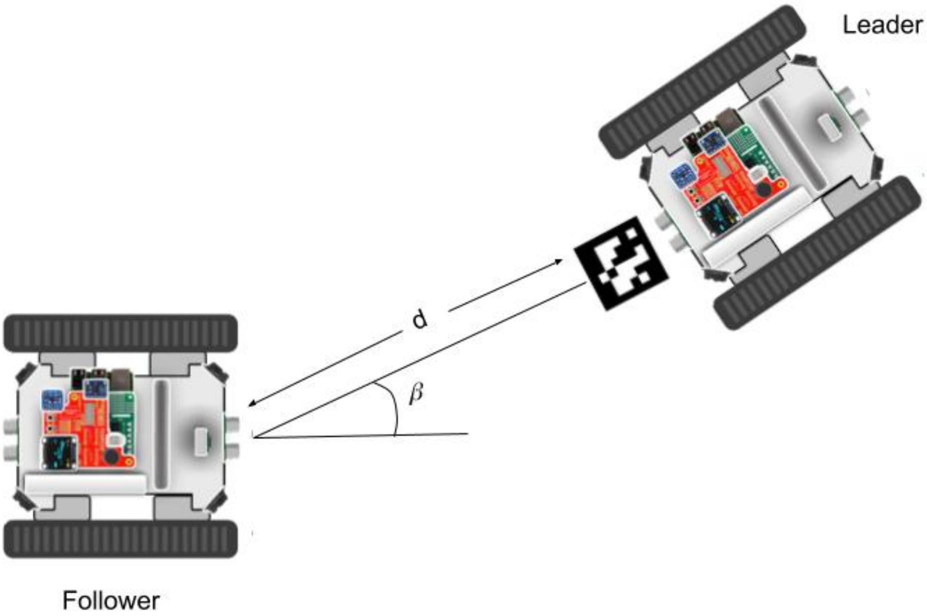


Figure 2.1: Rover Use Case

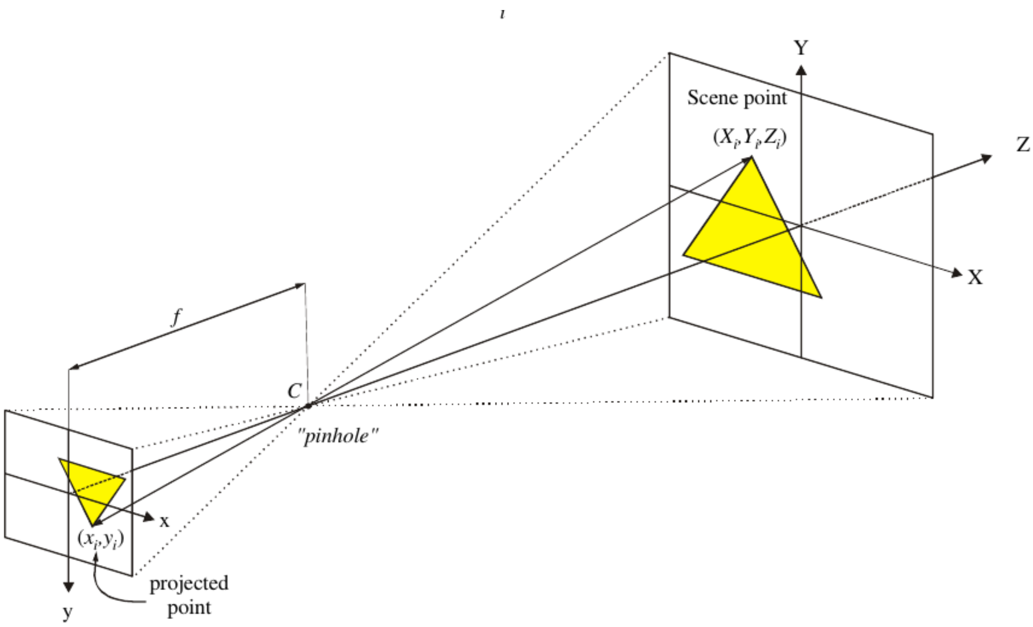


Figure 2.2: The pinhole imaging model [8].

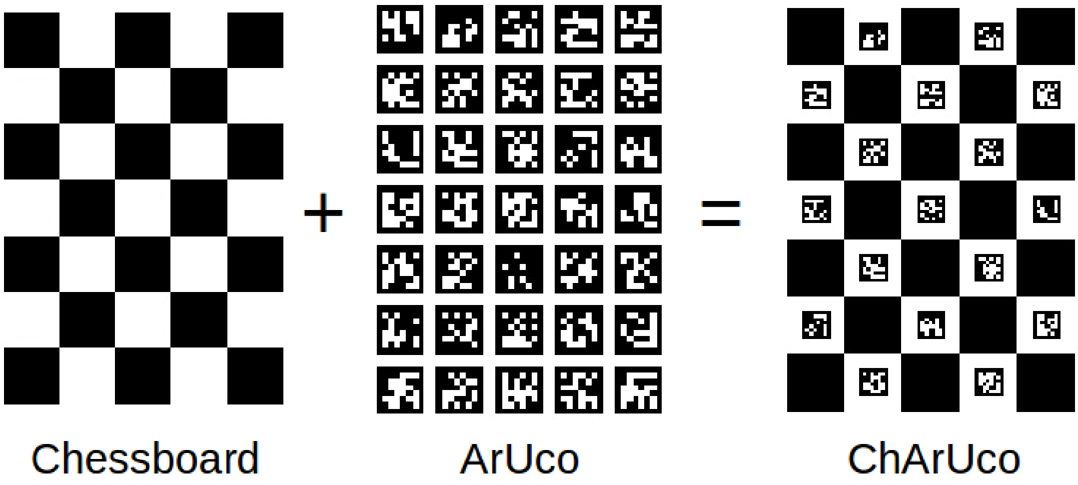


Figure 2.3: Plannar Patterns [3]

The camera intrinsic matrix **A** contains information about the internal parameters of the camera: focal lenght, image sensor format and principal point or image center. The coordinates of the principal point is described by (x_0, y_0), α_x and α_y represent the focal lenght in terms of pixels on the axis x and y, and γ is the skew of image.

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

The camera extrinsic parameters are given by the rotation matrix **R** and translation vector **t**. which are used to project an image on the world frame to camera frame. There is also a scale transformation, but it's already given by α_x and α_y.

Current cameras are equipped with lenses that produce some distortions on the images, however, the pinhole model is still a good aproximation for our case since we are using a **PiCamera** which has minimal distortions.

The camera calibration has been done with using `OpenCV`. This library implementation is based on the technique described by [14] and the matlab implementation done by [9]. The calibration technique in [14] requires the camera to observed a planar pattern, usually a chessboard pattern, at different orientations, the more the better the estimation of the intrinsic parameters. The calibration algorithm minimize the reprojection error which is the distance between observed feature points on the planer pattern and the projected using the estimates parameters.

For calibration we used a *ChArUco* board instead of the clasical chessboard because it generates a better estimation of the pamateres [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`. **A detailed explantion can be found in the Appendix (not sure but is a remainder)**.

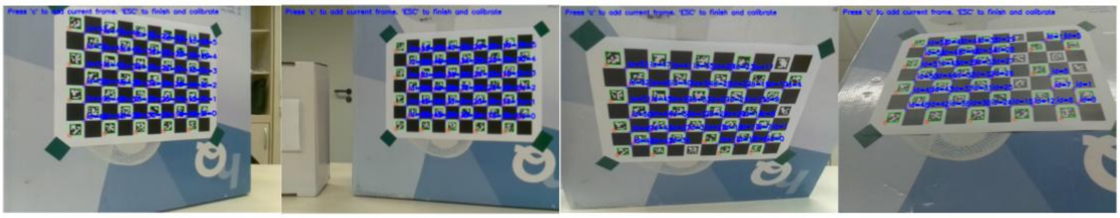


Figure 2.4: Some camera views used for calibration

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:

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

2.4 Extrinsic Parameters

As it was mention 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 [12].

However, a rotation can be also represented by a rotation axis $\mathbf{k} = [k_x, k_y, k_z]^T$ and an angle θ, or equivalenly 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.

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

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

2.5 Rotation matrix to Euler angles

In order to get the angles related a rotation whose yaw, pitch and roll angles are ϕ, ρ and ψ. 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

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

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

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

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

Given the given sequence of rotations and the algorithm described by [7], 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

2.6 Use case implementation details

As it was mentioned before, the follower should be completely autonomous. In order to do so, the follower will need to read data from sensors, process that data and generate movement based on the processed data.

In figure 2.1 is shown a diagram of our use case. The main sensor is a PiCamera, the processing part performs the marker detection on the captured frames and the calculation of the euler angles and distance based on the equations described in the last section, and finally the movement is generated using the driving rover services.

Video Processing with OpenCV

Just like in the case of camera calibration, we use `OpenCV` and the submodule `aruco` to capture video and process the frames in order to extract information from the visual markers. 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.

```
// 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 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 ω and the translation vector $\mathbf{t}$. The size of the marker is an input parameter of the `OpenCV` function `cv::aruco::estimatePoseSingleMarkers`. In our case the marker size is 7cm.

```
// image: input frame
// corners: detected corners
// rvect: rotation vector
// tvect: translation vector
cv::aruco::detectMarkers(image, dictionary, corners, ids);
cv::aruco::estimatePoseSingleMarkers( corners, 0.07,
                                      cameraMatrix, 0, rvec, tvec);
```

The next step is calculating the Euler angles by using function `cv::Rodrigues` and Slabaugh's algorithm. The `cv:Rodrigues` function is a direct implementation of equations **??** and **??**.

```
// rmat: rotation matrix
// angle: Euler angles
cv::Rodrigues(rvec.row, rmat);
rotationMatrixToEulerAngles(rmat, angle)
```

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, ψ is near 180 because the image is facing the camera as result the blue axis points towards the camera.

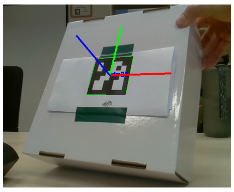


Figure 2.5: Camera axis

A basic code for video processing 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);
```

However, the estimated angles can not be used directly because estimations have small errors. In figure 2.6 can be observed the values of the rotation angles in a span of 1000 samples and in table 2.1 the stadistics of those samples. The ground truth values for euler angles were $[0, 0, 0]$, and for distance were 47.5 and 16 centimeters respectively.

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

Data	Mean	σ	Median
ψ deg	0.719853	0.162262	0.723000
ρ deg	0.584508	0.165382	0.507618
ϕ deg	1.155499	0.046940	1.157000

Figure 2.6: Angles in X-axis ψ, Y-axis ρ and Z-axis ϕ

Data	Mean	σ	Median
d cm	45.51706	0.033564	45.50772
d cm	15.65339	0.007401	15.65401

The results of standard deviation σ 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 pike 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 datain memory, but because the results showed that the mean and the median of euler angles are similar, thus it is reasonable to think that outliers has small influence on the data. In other words, the mean filter is a simple and effective option againts outliers problem. Its implementation is straighforward and requires no memory to save previous values. A pseudocode is as follows:

```
estAngle = 0;
for( i=0; i<samples; i++)
    estAngle += new_value;
estAngle /= samples;
```

Rover rotations

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

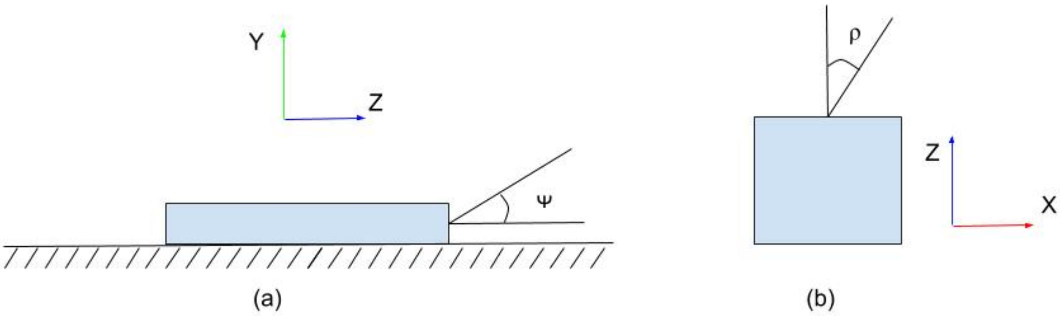


Figure 2.7: Rotation rotations (a) Rotation in X-axis (b) Rotation in Y-axis

in Y-Axis are possible. Aas shown in figure 2.7 rotations in X-axis are not possible since the rover can not fly or go underground. The same applies to rotations in Z-axis. In other words, the only relevant information from the estimated euler angles is ρ, or the angle related 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 diferrent axis, taking the "earth" or gravity acceleration as reference. On the other hand, the gyroscope measures angular speed relative to itself or 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 taking 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 inital position the angle will always be 0.

The gyroscope measures angular speed in all axis, in particular the angular speed in Y-axis or ω_y. The angular displacement ρ is just the integral of ω_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 Δt 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 beginnig of this chapter, in the pin hole camera model the 2D point $\hat{\mathbf{m}}$ and the 3D $\hat{\mathbf{M}}$ are related throught a projection matrix $\mathbf{P}$, 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 position, we must find $\hat{\mathbf{M}}$ given $\hat{\mathbf{m}}$.

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

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

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

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 ther rotate in order to reach to leader position. On the other hand, since the follower only rotates in Y-axis we only need to change the direction of the rotation and translation.

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

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

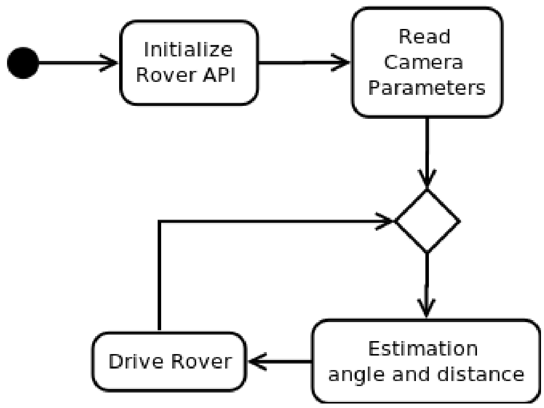


Figure 2.8: Activity diagram

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 inital set up, a infinity loop starts. During the loop we estimate the angle ρ and distance d, the later 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 ρ degrees. The current distance is measured using the front ultrasonic ranging module HC-SC04. The rover-API can only give accuarate information for distances lower than 40 cm [2], for distances greater than 40cm the API returns always 40cm. However, when the follower approaches the leader eventually will be in the measurable range.

Once the follower reaches the leader, it stops and wait until the leader moves again. A pseudocode 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

Results and Discussion

In this chapter, we evaluate the behavior of the follower in an controlled environment. We present the results of three test cases: known distance and angle, difference between ultrasonic sensor and camera, and fail cases. Our experiments consisted in three steps: - Set a known inital and final position. - Turn on follower rover - Measure the deviation from expected final position We measured the deviation using the front ultrasonic sensor, the Picamera and a ruler, because extreme precition, less than milimiters, are not required.

3.1 Known distance and angle

In this test case we evaluated followers accuaracy to reach the desired position and orientation. The different setting and initial camera estimations are presented on table 3.1.

Table 3.1: Inital 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 ρ	$30°$	$-30°$	$16°$	$-16°$	$0°$
Est. Angle ρ	$32.43°$	$-27.47°$	$15.78°$	$-15.73°$	$-2.84°$
Absolute Error	$2.43°$	$2.53°$	$0.22°$	$0.27°$	$2.84°$
Relative Error	8.1%	8.43%	1.38%	1.69%	∞

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

	1	2	3	4	5
Relative Error	14.17%	10.48%	7.95%	10.00%	9.15%
prox. Angle ρ	1.54°	2.75°	1.71°	0.95°	2.10°
Est. Angle ρ	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%

3.2 Difference between ultrasonic sensor and camera

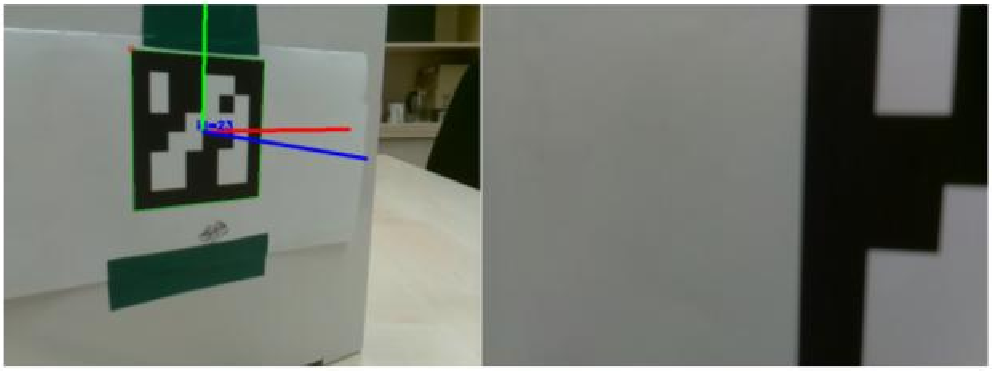
Table 3.3: Measurements and final estimations

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

3.3 Fail

-45 deg the ultrasonic sensors fails - Error due to there is no way to actually measure distance travelled - 11.png

Sometimes the follower is not able to detect the marker anymore since it is very close to it. An example is shown in 3.1. It is observed that at the final position $d = 3cm$, the follower lost track of the marker.



(a) (b)

Figure 3.1: (a) Initial position $d = 22.73cm$, $\rho = -33.41°$ (b) Final Position

Chapter 4

Challenges and issues

4.1 Camera vision field

The PiCamera mounted on the Rover has a limit field of vision. As observed in figure "circular shape ref", the camera only sense from $-X$ degrees up to X degrees, also note the ρ angles under those limits are -33 degrees and 28 degress respectively.

In addition, there is a problem with the estimation of extrinsic camera parameters. In figure "vision field" is presented three diferrent positions

4.2 Maximum rotation

There is a maximun ρ angle in clockwise and counterclockwise directions that can be detected. In figure "rotations" is shown the position and angle of the roverleader so the marker can be still detectable. For counterclockwise rotations the maximum angle is $\rho = -77$ degrees and for clockwise rotations is $\rho = 72$ degrees.

4.3 Building AGL...

4.4 Switching to Raspbian due to AGL building issues

Chapter 5

Conclusions

None so far

Chapter 6

Summary

A summary of the diploma

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] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach.* Prentice Hall Professional Technical Reference, 2002.

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

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

[8] 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.

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

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

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

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

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

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

Appendix A

Appendix

A.1 Calibration of the picam in raspbian

The `OpenCV` version we are using is 3.4.1.

bla bla bla... more details here

A.2 Answer to the "Life, the Universe, and All" questions

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42.$$

A Faraday-indukciós törvényből levezetve

$$\mathrm{rot}E = -\frac{dB}{dt} \quad \longrightarrow \quad U_i = \oint_{\mathbf{L}} \mathbf{Edl} = -\frac{d}{dt}\int_A \mathbf{Bda} = 42.$$