Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Prof. Dr.-Ing. Hakan Kayal
Professorship for Space Technology
Chair for Computer Science VIII

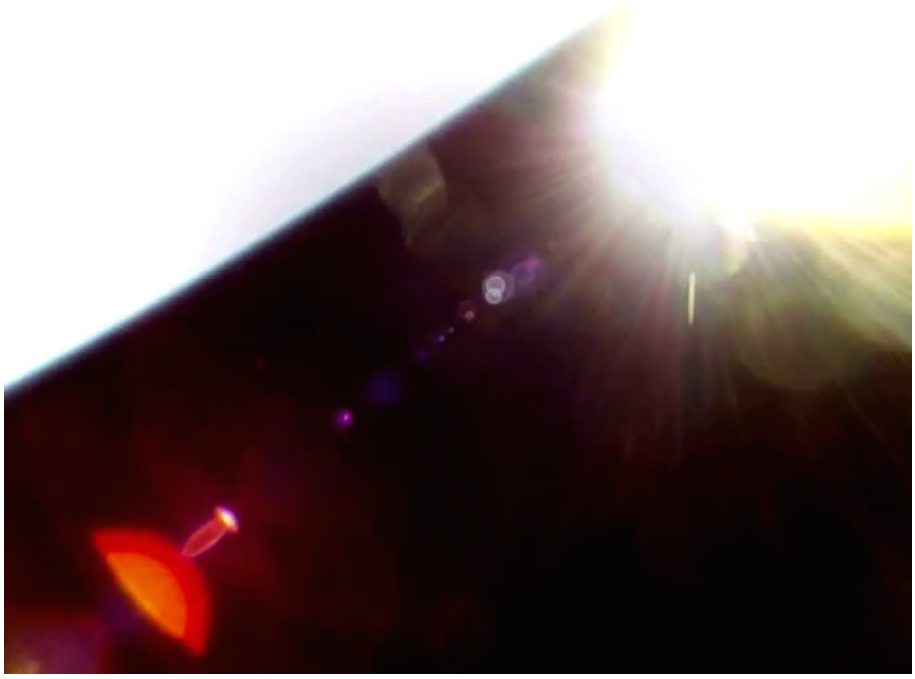
BACHELOR THESIS

DEVELOPMENT AND IMPLEMENTATION OF AN HORIZON SENSING ALGORITHM BASED ON IMAGE PROCESSING TECHNOLOGIES

Thesis submitted by

Jochen Barf

to the Department of Mathematics and Computer Science in Partial Fulfillment of the Requirements of the Degree of Bachelor of Science



Abstract

The purpose of this bachelor thesis is to develop an autonomous algorithm that is able to calculate a vector to the center of the earth from ordinary image data. The introduction shows the general purpose of attitude sensor systems and the HORACE project is explained. The first part of the development is a detailed elaboration of the algorithmic approach with its inputs, outputs and every step of the image processing. The next part is the implementation of this approach as payload for the HORACE project using the OpenCV runtime library. The last part is a further development of this implementation towards an application in embedded systems. Therefore all used functions from OpenCV are implemented as member functions of the algorithm using standard c++ libraries only.At the end of this thesis an outlook to the future use of this horizon sensor is given.

This thesis work is dedicated to my loving parents
Dietlind and **Gunter Barf**
for their outstanding support during my studies.

Contents

Acronyms

ACS attitude control system 7, 8

ADCS attitude determination & control system 7, 11

ADS attitude determination system 7–9, 11, 47

BEXUS Balloon Experiment for University Students 10

CAM camera 11, 12, 28–30

CMOS Complementary metal-oxide-semiconductor 28

CS core system 11, 28, 30

DLR German Aerospace Center 10

EGSE electrical ground support equipment 11

ESA European Space Agency 10

FOG Fiber Optic Gyro 9

FS flight segment 11, 12

GNSS global navigation satellite system 8

GS ground segment 11

HORACE Horizon Acquisition Experiment 1, 3, 10–12, 28, 29, 33, 34, 42, 44, 45

MGSE mechanical ground support equipment 11

MORABA Mobile Rocket Base 10

MU measurement unit 11

OpenCV Open Source Computer Vision Library 28–30, 33, 34, 40

PDU power distribution unit 11

REXUS Rocket Experiment for University Students 10, 12, 30, 45

RLG Ring Laser Gyro 9

RTTI run-time type information 34

SATA Serial Advanced Technology Attachment 28

SNSB Swedish National Space Board 10

SSC Swedish Space Corporation 10

SSD solid state disk 28, 31

ZARM Center of Applied Space Technologie and Microgravity 10

1 Introduction

Every space mission has different objectives and constraints which result in a very unique set of requirements. All subsystems must be build in order to fulfill these requirements and thus are custom made. However, all missions can be divided into three general segments, their elements and subsystems.

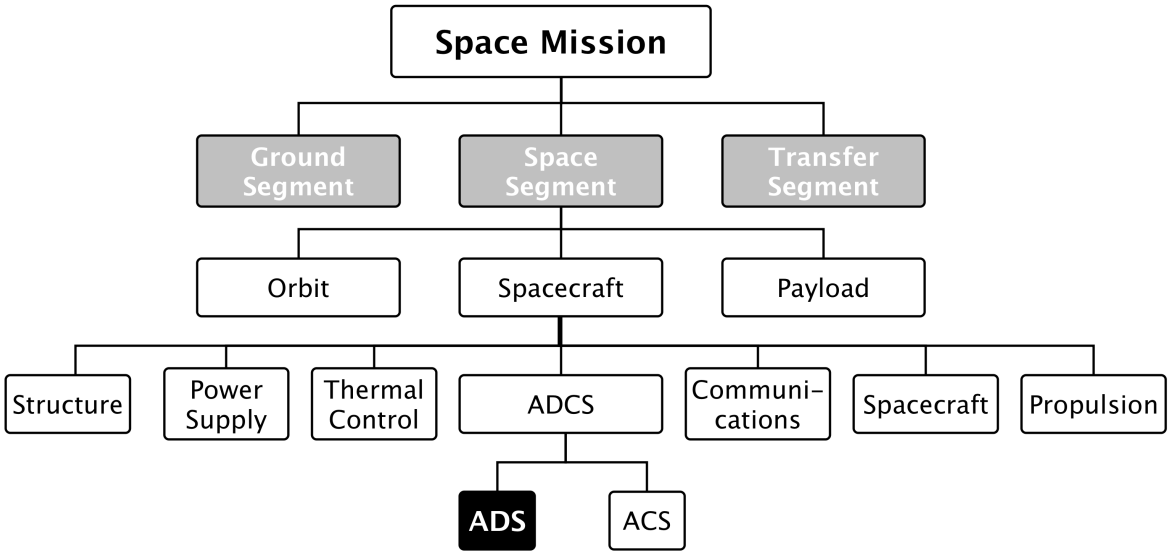


Figure 1.1: role of the attitude determination system (ADS) in a space mission [LWH09, 1.2.1]

A space mission basically consists of the transfer segment, the ground segment and the **space segment**. The space segment describes the device that is launched by the transfer segment and is controlled by the ground segment. This segment itself has the elements payload, orbit and **spacecraft**. The segments spacecraft and orbit create the needed environment for the payload and thus are highly influenced by it. The spacecraft, also known as space vehicle platform or (satellite) bus, has seven subsystems: structure, power supply, thermal control, communications, data processing, propulsion and **attitude control**. The attitude determination & control system (ADCS) is responsible for the orientation of the spacecraft and may not be mistaken with the propulsion system which, controls the orbit (cf. [LWH09, 1.2.1]). Figure 1.2 illustrates the typical control loop of an active ADCS. In most common applications the main task of the ADCS is to preserve a desired attitude but changes are nevertheless required for adjustments of antennas or cameras. This desired attitude is compared with the measured attitude by the ADS. The deviation is used in the attitude control system (ACS) computer to calculate the needed rotations to obtain the desired attitude. This information is send to the actuators that generate corresponding torques. Due to the limited accuracy of the actuators and external disturbance torques, the spacecraft's actual attitude status differs from the desired attitude. Disturbance torques can be caused by the magnetic field of the planet, the inhomogeneous gravity field of the

planet, solar radiation and residual atmosphere. The attitude status is measured again by the sensors of the ADS and the closed control loop starts from the beginning.

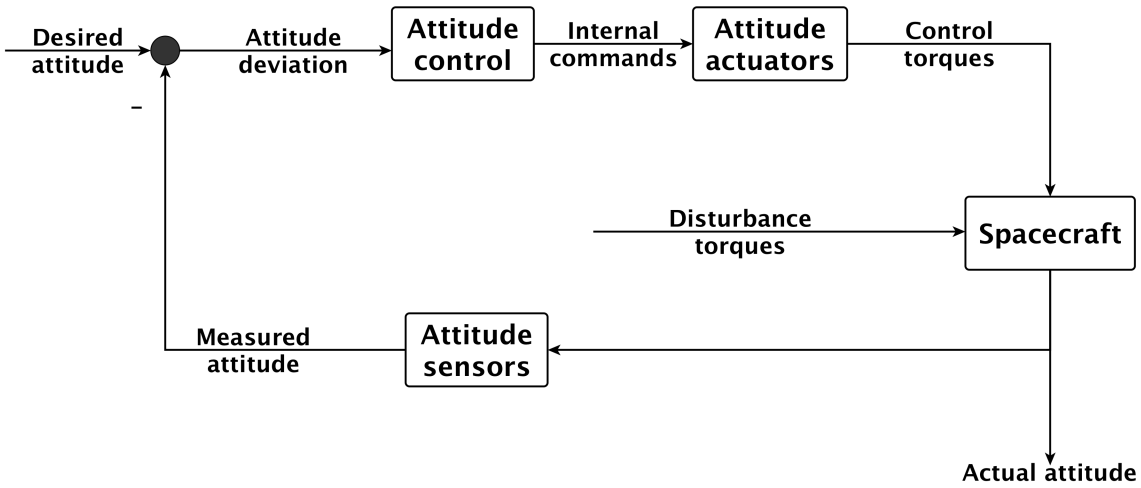


Figure 1.2: Attitude control loop [LWH09, Figure 4.5.4]

1.1 Attitude Determination

The attitude of a spacecraft describes the orientation of a vehicle in space but does not give any information about the position. This orientation is always given with respect to a reference system either as direction vectors or as angles. The data of the spacecraft's current attitude status, needed by the ACS, is provided by the ADS. The ADS can be realized by a wide range of sensors and is influenced by the accuracy requirements of the ACS. Satellite builders have several sensor systems at hand to design the satellite's ADS (cf. [LWH09, 1.2.1]):

- star sensors
- sun sensors
- earth sensors
- magnetometers
- gyroscopes
- global navigation satellite system (GNSS) for attitude determination

Star sensors are well known for their high accuracy of a few arc seconds and the three-axis attitude information. They consist of a camera and an electronic part for the image processing. By comparing stars in the image with a star catalog they are able to identify star patterns and to determine the inertial attitude. Due to the sensor's high light sensitivity, not only a very steady picture must be available but furthermore, the sensor suffers from hard radiation and must be cooled to lower the degradation.
Sun sensors are two-axis sensors and can be divided in coarse and fine sun sensors. The coarse sensors are basically solar cells which are mounted on several positions of

the spacecraft. By comparing the currents of the cells the attitude status is measured with a $20° - 10°$ accuracy. Fine sun sensors are either an array of photo cells or a CCD chip and provide an accuracy of $0.01°$.
Earth sensors detect the horizon of the earth in the thermal infrared spectrum at a wavelength of about $15\mu m$ and determine the attitude angles roll and pitch. The attitude information is available during the whole orbit but is temporarily disturbed by the sun and the moon, if in the field of view. These sensors can be divided in two kinds, the static and the scanning sensors. The static earth sensor is a pair of two germanium lenses which produce a current according to their exposure. By comparing these currents the attitude information can be provided with $1°$ accuracy. For a two-axis attitude information, at least two static sensors are necessary. The scanning sensor has a rotating mirror and provides a two-axis attitude information with an accuracy of $0.05°$ by measuring the transitions of the horizon signal.
Magnetometers use the magnetic field of the earth to extract an attitude information. This cheap and reliable sensor provides an accuracy of $0.1° - 1°$. Disturbances can occur by the magnetic field of the satellite. Therefore, the magnetometer must be placed in a distance outside the satellite.
Gyroscopes measure the rotations in the inertial reference frame without any external sources. The attitude information can be provided as angular velocity or as an angular increment of the last measurement span. There are three types used: the mechanical gyro, the Fiber Optic Gyro (FOG) and the Ring Laser Gyro (RLG). The mechanical gyro has a rotating mass which is torques-free, gimbal mounted and thus forms a inertial reference system. The attitude angles can then be directly measured with an accuracy of $0.01°/h$. Both, the FOG and the RLG use the interference of two laser beams to measure the angular rotation. The FOG reaches an accuracy of $1°/h$ whereas the RLG reaches an accuracy of $0.01°/h$.
Although, GPS and GLONASS were build for position determination on earth, the signals can be used to get a three-axis attitude information in space. The accuracy of $0.1°$ is reached by positioning several antennas on different places of the spacecraft and determining the time discrepancy of the signal received at the antennas (cf. [LWH09, 4.5.6]).

1.2 Horizon Sensor

The horizon sensor, whose software development is part of this work, is meant to extend the list of available ADS sensors. The clou of this two-axis earth sensor is that it works in the visible spectrum of light. Its desired design is either a software package for satellites that already carry a camera or a stand-alone system with a build-in camera and an electronic part for the image processing. Either way, this system has many advantages:
If used as a software package there is no additional mass and even the stand-alone version does not exceed the estimated mass of 0.3 kg since similar systems are even lighter (e.g. STELLA, University of Würzburg). Since no special hardware, like germanium lenses in the static earth sensor, is used, the costs of this system are estimated to be relatively low. The field of operation must not only be in an earth orbit but can also be in an orbit of any other planet. The star sensor needs a very steady picture to function whereas the horizon sensor also works at a rotation with high rates. So does the sun sensor, but the sun is a small object that is likely to be missed while rotating.

However, when orbiting a planet, it is the biggest object visible and thus very likely to be in the field of view. In contrast to the scanning earth sensor, the horizon sensor has no moving parts and is less sensible to the degradation of the optical sensor than the star sensor. Sun and moon in the field of view do not affect the performance of the sensor.
But along with the advantages there also come disadvantages: Due to the sensitivity to visible light, the horizon sensor does not function during the whole orbit. The output data rate is estimated to be lower than the rate of conventional earth sensors.

1.3 HORACE

1.3.1 Basic Idea

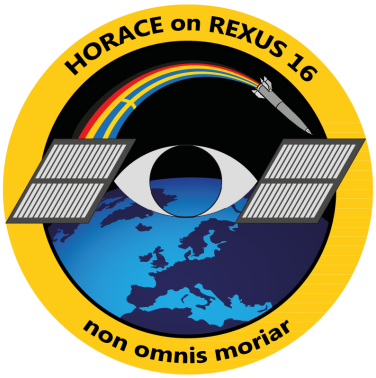


Figure 1.3: HORACE mission Patch

HORACE stands for Horizon Acquisition Experiment and is a student project in the REXUS programme.

1.3.2 REXUS

"The Rocket Experiment for University Students (REXUS) programme allows students from universities and higher education colleges across Europe to carry out scientific and technological experiments on sounding rockets. Each year, two rockets are launched, carrying up to 10 experiments designed and built by student teams. The basic idea behind REXUS is to provide an experimental space platform for students in the field of aerospace technology.
The REXUS/BEXUS programme is realised under a bilateral Agency Agreement between the German Aerospace Center (DLR) and the Swedish National Space Board (SNSB). The Swedish share of the payload has been made available to students from other European countries through a collaboration with the European Space Agency (ESA). EuroLaunch, a cooperation between the Esrange Space Center of the Swedish Space Corporation (SSC) and the Mobile Rocket Base (MORABA) of DLR, is responsible for the campaign management and operations of the launch vehicles. Experts from DLR, SSC, ESA and the Center of Applied Space Technologie and Microgravity (ZARM) provide technical support to the student teams throughout the project. REXUS and Balloon Experiment for University Students (BEXUS) are launched from SSC, Esrange Space Center in northern Sweden." [RX/14, REXUS]

The ADCS of a satellite must work autonomously not only during nominal phases of the mission, but also in unexpected situations or emergency cases. An emergency case could occur when the satellite is spinning and tumbling uncontrolled at high rates. It is therefore necessary to have an ADS sensor that is able to operate in stress conditions and is also affordable for smaller satellites and missions. Almost all missions are situated in the orbit of a massive central body which in most cases is the earth. It is nearly impossible that the satellite would spin and tumble in a mode during which the central body is never visible. Thus it is reasonable to use the central body for attitude determination. In contrast to existing earth sensors, that detect the earth's IR radiation, HORACE uses an optical sensor for the horizon detection, which is sensitive to the visible spectrum, to keep expenses low and to emphasis the image processing software-components of the system. (cf. [RGW+14, 1.1])
Figure 1.4 defines some terms and provides an overview of the hierarchy within the project. HORACE is divided in two segments, the flight segment (FS) and the ground segment (GS). The FS consists of the subsystems core system (CS), measurement unit (MU), power distribution unit (PDU), camera (CAM) and the mechanical structure. The GS is compromised of the ground station, the mechanical ground support equipment (MGSE) and the electrical ground support equipment (EGSE).

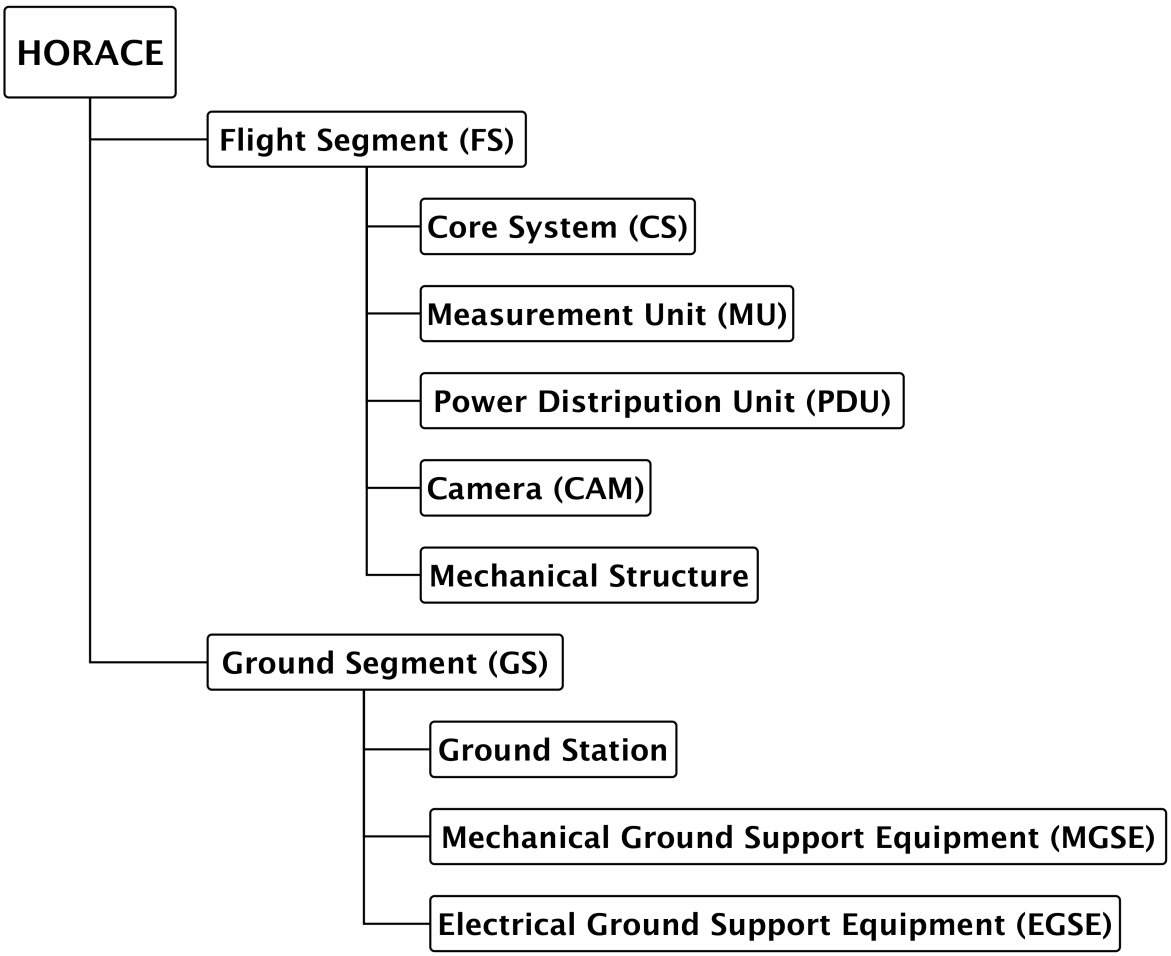


Figure 1.4: HORACE hierarchy [RGW+14, Figure 1-1]

1.3.3 Mission Statement

"HORACE on REXUS 16 is a technology demonstration mission for autonomous earth detection on satellites. The aim is to prove or disprove the general technical feasibility of the outlined approach. During the mission the functionality and robustness of the general approach is tested under realistic, space-like conditions, by means of the HORACE flight segment. After post flight evaluation it shall be determined whether the approach of autonomous horizon acquisition with a camera in conjunction with image processing algorithms running on an embedded system connected to the CAM is indeed apt to (re)acquire a satellite's attitude under nominal or stress conditions." [RGW+14, 1.2]

1.3.4 Mission Objectives

"With HORACE, whose development will be part of the mission, the following **primary objectives** shall be reached:

- Investigate whether horizon acquisition can be performed accurately enough for attitude determination.
- Determine whether the very dynamic and time-critical problem can be solved with an embedded system with reasonable time resolution and power consumption.

Secondary objectives are:

- to show physical or systematic limits and problems of the general approach.
- to determine, if a future attitude determination system following the general approach would be applicable also for small satellites." [RGW+14, 1.3]

1.3.5 Requirements

Table 1.1: brief software requirements of the HORACE project [RGW+14, 2.2]

ID	Requirement text
F-S-02	The FS shall calculate the 2D vector to the 2D projection of the earth center.
F-S-03	The FS shall save the experiment data with global timestamp.
D-S-07	Of the calculated data the FS shall save the stop of calculation timestamp.
P-S-01	The 2D vector to the earth center should be calculated with 2 digits.
P-S-04	When the rocket is spinning with low rates (< 0.3Hz) AND if there are no image disturbances the results of horizon acquisition should be successful in 90% of those cases.
P-S-09	When the rocket is spinning with high rates (> 1.0Hz) AND if there are many image disturbances the results of horizon acquisition should be successful in 30% of those cases.
P-S-10	The amount of false negative horizon acquisitions should be less than 10%.

In table 1.1 important software requirements, concerning the algorithm are listed. False negatives in this context are failed detection where a horizon is clearly visible whereas

false positives are detections where no horizon is visible. The mentioned image disturbances are phenomena like

- sun in the image
- lens flares
- too dark illumination
- too bright illumination.

A horizon acquisition is successful if

- the ratio between the calculated earth radius and the real earth radius $\frac{r}{R}$ holds $0.9 < \frac{r}{R} < 1.1$
- the error of the calculation of the center of earth e (euclidean distance) related to the real earth radius R holds $\frac{e}{R} < 0.1$

(cf. [RGW+14, 2.2])

2 Algorithmic Approach

2.1 Basic Idea

When looking at a picture like the one in figure 2.1 it is easy for the human mind to identify the earth, the sun and even sun flares. We can tell where the center of the earth must be at first glance. That is because our brain can recognize objects and is able to compare them with familiar objects from the memory. But how can a computer manage this? To make it possible for machines to acquire the earth's horizon,

Figure 2.1: Image of the earth's horizon and the sun taken from a REXUS rocket by the team EXPLORE

this approach, illustrated in 2.2 uses the high contrast between the earth (bright) and space (dark). The first step in the preprocessing is to convert the coloured image into a grayscale image. This image is then converted to a black-and-white image also known as binary image. Now bright areas of the picture are represented by pixels holding 1, respectively dark areas by pixels holding 0. The border between a 1 area and a 0 area is called an edge. Since the contrast between earth and space is high, the horizon

is an edge in the picture. Unfortunately, it can happen, that the horizon is not the only edge in the picture. Therefore, it is necessary to find the edge that represents the horizon. Having found this edge, the horizon line still is a set of pixels that are part of a circle. To find this circle, represented by its center and the radius, this algorithm finds a circle that fits the best to all data points. Having the center of the circle that represents the earth's horizon it is obvious how to calculate the direction vector from the image frame origin to the center.

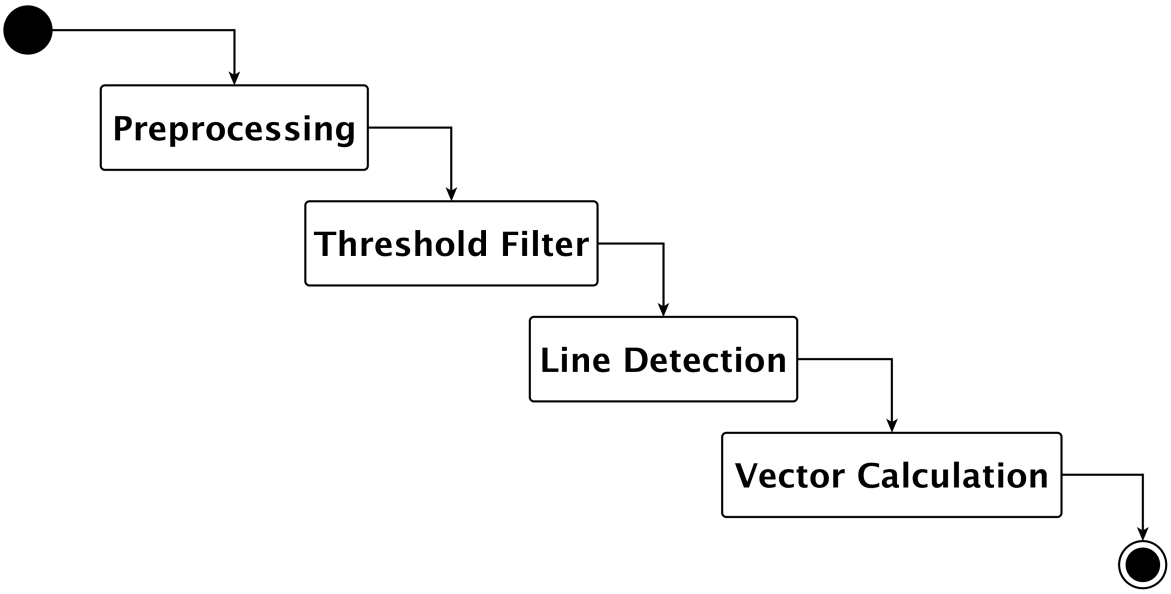


Figure 2.2: activity diagram oft the algorithm

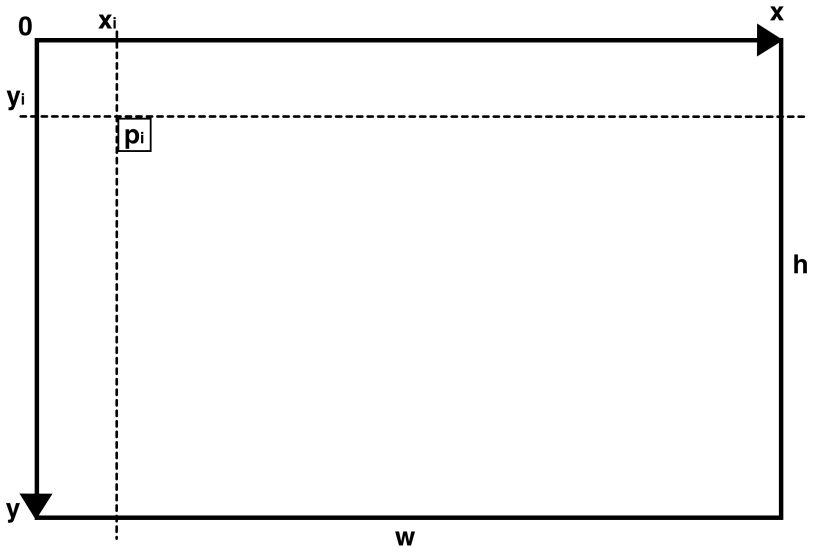


Figure 2.3: image frame coordinate system

2.2 Detailed Steps

To simplify further explanations there is a list of all assumptions. All quantities are measured in pixels:

- let the origin of the coordinate system, with the axes x an y, be in the top-left corner of the image (see fig. 2.3)
- the x-coordinate increases by moving right
- the y-coordinate increases by moving down
- let h_s be the height (y-direction) of image s
- let w_s be the width (x-direction) of image s
- let k_s be the number of channels in image s
- let b_s be the brightness of image s determined in 2.2.1
- let $b_s(i)$ be the brightness of pixel i in the image s
- let p_i be a point with the coordinates (x_i, y_i) where $i \in \mathbb{N}^+$, $i < h_s \cdot w_s$ is the index of a pixel i
- let $f_s(p, c)$ be the content of the channel c of image s at position p
- $f_s(p)$ is used as abbreviation for $f_s(p, 0)$ and is only valid if $k = 1$
- let $f_s(l, m) \leftarrow u$ be an assignment of the value u to the value of m at point l in image s

2.2.1 Preprocessing

The preprocessing step checks if the image is worth calculating. If the horizon is not visible in the picture, the calculation will clearly not succeed and the time, spend on the calculation, was pointless. Thus, it is reasonable to check whether the horizon is visible as long as the check does not cost more time than a futile calculation. This is done by determining the brightness of the image and excluding images with very high or low values from further calculation. The brightness is determined in the following manner:

- let $0 \leq a \leq 1$ be the ratio of pixels that are gathered for brightness determination
- let $n = h_s \cdot w_s \cdot a$ be the number of pixels that are gathered for brightness determination
- let $r() \in \mathbb{N}^+$, $r() \leq h_s \cdot w_s$ be a different random number every time it is used
- let b_{max} be the maximal brightness value
- let b_{min} be the minimal brightness value

- the brightness $b_s(i)$ of pixel i in image s is defined as the arithmetic average of the values of its channels:

$$b_s(i) := \frac{\sum_{c=1}^{k_s} f_s(p_i, c)}{k_s} \tag{2.1}$$

- the brightness b_s of image s is then defined as the arithmetic average of its random pixels.

$$b_s := \frac{\sum_{i=1}^{n} b_s(r())}{n} \tag{2.2}$$

- decide according to the pattern:

$$instruction = \begin{cases} skip, & b_s > b_{max} \\ accept, & b_{max} \geq b_s \geq b_{min} \\ skip, & b_s < b_{min} \end{cases} \tag{2.3}$$

2.2.2 Threshold Filter

The threshold filter converts a multi channel image s like the one in figure 2.1 into a binary image t (fig. 2.4), that means there is only one channel ($k_t = 1$) and there are only two valid values of a pixel 1 (white) and 0 (black). In this thesis two methods are defined to apply a threshold filter, the static and the dynamic method.

Method 1: static

The static method compares the brightness $b_s(i)$ of a pixel i in an image s with the threshold value and decides whether the value of the new image t is 1 or 0 according to the following scheme:

- let b_{thresh} be the static threshold value
- for all $i \in \mathbb{N}^+, i \leq h_s \cdot w_s$ is

$$f_t(p_i) \leftarrow \begin{cases} 1, & b_s(i) \geq b_{thresh} \\ 0, & else \end{cases}$$

Method 2: dynamic

In contrast to the static method, the dynamic method compares the brightness $b_s(i)$ (2.1) of a pixel i in an image s with the product of the the brightness of the image b_s (2.2) and a factor. Then it is decided whether the value of the new image t is 1 or 0 according to the following scheme:

- let b_{factor} be the dynamic threshold factor
- for all i

$$f_t(p_i) \leftarrow \begin{cases} 1, & b_s(i) \geq b_s \cdot b_{factor} \\ 0, & else \end{cases}$$

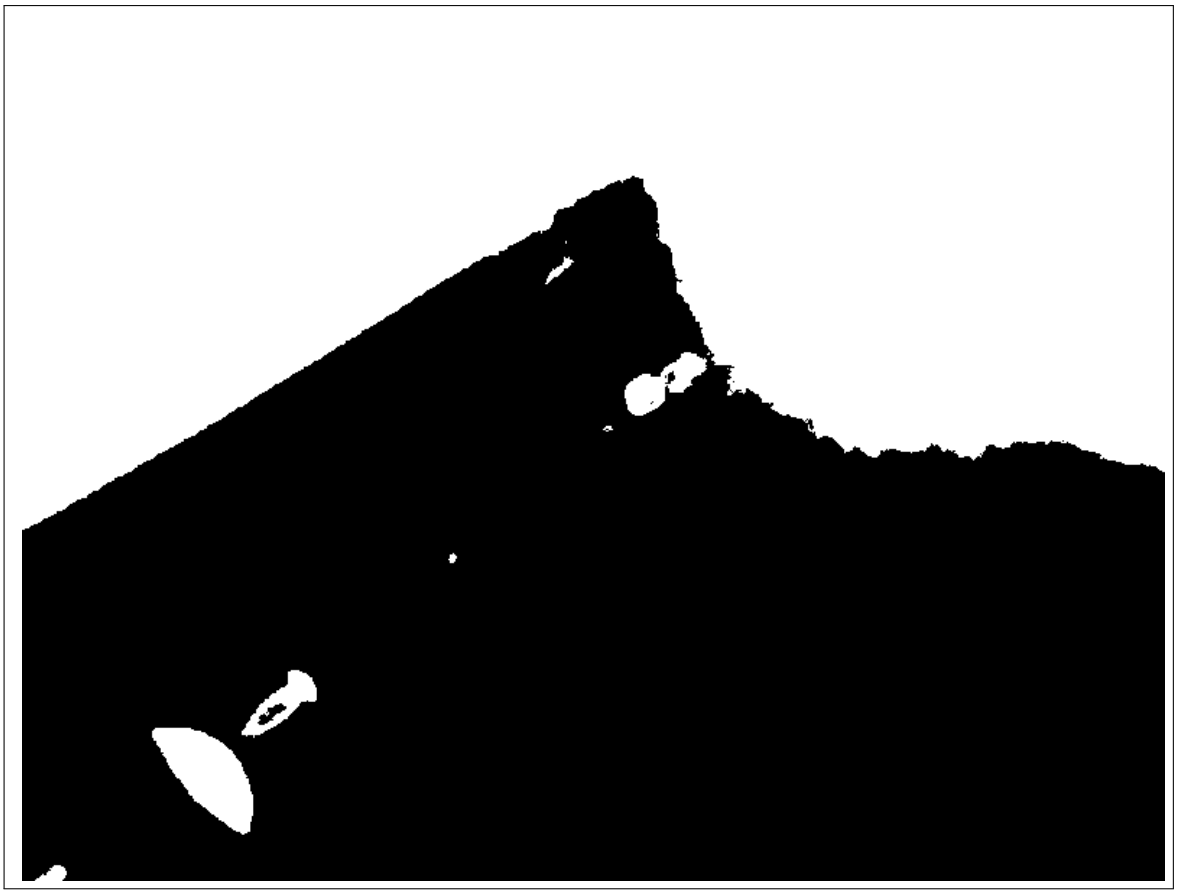


Figure 2.4: threshold Filter applied to image in fig 2.1 with the static method and threshold value 80

2.2.3 Line Detection

The line detection is an edge detection with a subsequent selection of a line. An edge is the border between 1 and 0 components of an image. If there is more than one edge a line detection is necessary to select one.
The following is assumed:

- a line $L = \{p, q, r...\}$ is a set of points
- $|L|$ is the length of the line L (number of elements)
- $L \leftarrow p_i$ expands the set L by the point p_i

Method 1: topological search

The topological search analyses the topological structure of binary images by border following. It first searches with a TV raster scan for a pixel that satisfies the criteria for a border pixel and then follows that border until it finds the pixel it started from. While following it marks the border pixels with a unique identifier. After being done with one border it resumes the TV raster scan to find the next border and repeat the procedure until it reaches the end of the image. Algorithm 2 in [SA83] is modified in a way that it only detects outer borders, that means if a border is enclosed by another

Figure 2.5: edge detection applied to image in 2.4 with method topological search

border, this border is not detected. The longest border is then selected as the horizon line, the line that is presumed to contain the horizon (see fig. 2.6).
This is the algorithm introduced by Satoshi Suzuki and Keiichi Abe in [SA83]. This method is the Algorithm 2 in [SA83] with the following extensions:

- let $m \in \mathbb{N}_0^+$ be the number of the border
- for every border m is $L_m \leftarrow (j_3, i_3)$, for $m = |NBD|$ (j_3, i_3 and NBD from [SA83, Algorithm 2]) if and only if
  - case (a) or (b) in substep (3.4) is accepted
  - $x \neq 0$
  - $y \neq 0$
  - $x \neq w_t$
  - $y \neq h_t$
- select horizon line $H_t = \{L_q \; for \; q \in \mathbb{N}^+, |L_q| = max\}$

Method 2: erode

The method "erode" shrinks the 1 components of a binary image s and subtracts the resulting image from the original image. The remaining pixels are the edges of the 1

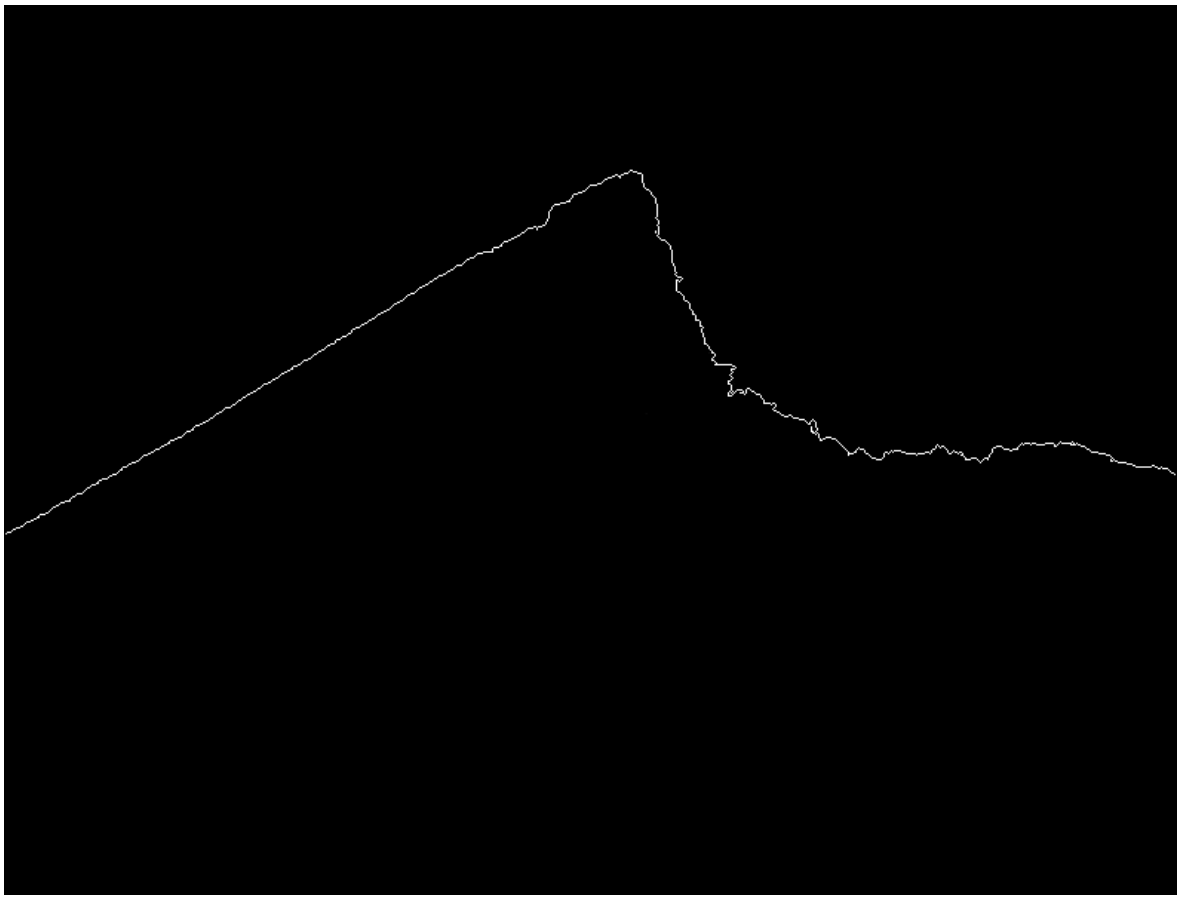


Figure 2.6: line detection applied to image in 2.5 with method topological search

components. Since it is not possible to distinguish the borders, all edges are one line and the line selection is obsolete. This means in detail:

- the *g-neighbourhood* of a pixel i are the pixels inside the square with center i and the edge length $g \in [3, 5, 7, ..., min(h_s, w_s)]$
- let

$$G_g(i) = \sum_{x=x_i-\lfloor \frac{g}{2} \rfloor}^{x_i+\lfloor \frac{g}{2} \rfloor} \sum_{y=y_i-\lfloor \frac{g}{2} \rfloor}^{y_i+\lfloor \frac{g}{2} \rfloor} f_s((x_i, y_i), 0) \tag{2.4}$$

  be the number of pixels in the *g-neighbourhood* that have the value 1 where $i \in \{k \in \mathbb{N}^+ | w_s - \lfloor \frac{g}{2} \rfloor > x_k > \lfloor \frac{g}{2} \rfloor \wedge h_s - \lfloor \frac{g}{2} \rfloor > y_k > \lfloor \frac{g}{2} \rfloor\}$
- let $a \in [1, 2, 3, ..., g \cdot g - 1]$ be the erode threshold value
- the horizon line

$$H = \{p_i | G_g(i) > 0 \wedge G_g(i) < a \wedge f_s(p_i) = 1\} \tag{2.5}$$

2.2.4 Vector Calculation

The desired vector is the direction vector from the center of the image to the center of the 2D projection of the earth in the image plain. The center of the earth is calculated

with the Least Square Method. It takes the horizon line H, which must contain at least γ points, as input and calculates the best fitting circle to these data points. Not only the center is calculated but also the radius of the earth in the image plain. This allows to check if the calculation returned a valid result by comparing the radius with the expected radius given by the height above ground.

Circle Fit

Let γ be the minimal number of points in the horizon line H.

$$instruction = \begin{cases} skip, & |H| < \gamma \\ accept & |H| \geq \gamma \end{cases} \tag{2.6}$$

The Least Square method minimizes the error $e(\lambda_1, \lambda_2, \lambda_3, ...) = f_d - f(\lambda_1, \lambda_2, \lambda_3, ...)$ of a function f to its desired value f_d by finding the optimal parameters λ_i. The equation for the least square fit is (cf. [Lea14])

$$\lambda_1^2 = (x - \lambda_2)^2 + (y - \lambda_3)^2 \tag{2.7}$$

where $p = (x, y) \in H$ is a data point,λ_1 is the radius and (λ_2, λ_3) are the coordinates of the center of the circle. The error e is then:

$$e(\lambda_1, \lambda_2, \lambda_3) = (x - \lambda_2)^2 + (y - \lambda_3)^2 - \lambda_1^2 \tag{2.8}$$

The non-linear equation 2.7 can be linearized by the substitutions

$$A := 2\lambda_2 \tag{2.9}$$
$$B := 2\lambda_3 \tag{2.10}$$
$$C := \lambda_1^2 - \lambda_2^2 - \lambda_3^2 \tag{2.11}$$

as follows (cf. [Lea14]) :

$$Ax + By + C = x^2 + y^2 \tag{2.12}$$

To minimize the error of the complete data set the sum of every error square

$$E(\lambda_1, \lambda_2, \lambda_3) = \sum_{i=1}^{|H|} e_i^2(\lambda_1, \lambda_2, \lambda_3) \tag{2.13}$$
$$= \sum_{i=1}^{|H|} [(x_i - \lambda_2)^2 + (y_i - \lambda_3)^2 - \lambda_1^2]^2 \tag{2.14}$$
$$= \sum_{i=1}^{|H|} [Ax_i + By_i + C - x_i^2 - y_i^2]^2 \tag{2.15}$$

must be minimized. This is done by solving the system (cf. [Lea14])

$$\begin{pmatrix} \frac{\partial E}{\partial \lambda_1} & \frac{\partial E}{\partial \lambda_2} & \frac{\partial E}{\partial \lambda_3} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \tag{2.16}$$

All following sums in this section without indicated limits are $\sum_{i=1}^{|H|}$. The solution of 2.16 is the following system (cf. [Lea14]) :

$$\begin{pmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i \\ \sum x_i y_i & \sum y_i^2 & \sum y_i \\ \sum x_i & \sum y_i & |H| \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} \sum x_i (x_i^2 + y_i^2) \\ \sum y_i (x_i^2 + y_i^2) \\ \sum x_i^2 + y_i^2 \end{pmatrix} \tag{2.17}$$

The substitutions 2.18 and 2.19 are done to simplify the view:

$$\begin{pmatrix} K & L & M \\ L & P & O \\ M & O & N \end{pmatrix} := \begin{pmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i \\ \sum x_i y_i & \sum y_i^2 & \sum y_i \\ \sum x_i & \sum y_i & |H| \end{pmatrix} \tag{2.18}$$

$$\begin{pmatrix} Q \\ R \\ S \end{pmatrix} := \begin{pmatrix} \sum x_i (x_i^2 + y_i^2) \\ \sum y_i (x_i^2 + y_i^2) \\ \sum x_i^2 + y_i^2 \end{pmatrix} \tag{2.19}$$

The equation 2.17 can be solved for A, B and C as follows:

$$A = \frac{O^2Q - NPQ + LNR - MOR - LOS + MPS}{L^2N - 2LMO + KO^2 + M^2P - KNP} \tag{2.20}$$

$$B = \frac{LNQ - MOQ + M^2R - KNR - LMS + KOS}{L^2N - 2LMO + KO^2 + M^2P - KNP} \tag{2.21}$$

$$C = \frac{MPQ - LOQ - LMR + KOR + L^2S - KPS}{L^2N - 2LMO + KO^2 + M^2P - KNP} \tag{2.22}$$

The desired values reveal then as:

$$\textit{radius } \lambda_1 = \frac{\sqrt{4C + A^2 + B^2}}{2} \tag{2.23}$$

$$\textit{center x-coordinate } \lambda_2 = \frac{A}{2} \tag{2.24}$$

$$\textit{center y-coordinate } \lambda_3 = \frac{B}{2} \tag{2.25}$$

Radius Check

To verify the calculation, the expected radius in the image is determined via the height above ground, the radius of the central body and the parameters of the optical system. An sketch of the system is given in figure 2.7. The point M is the center of the earth while point L defines the position of the lens. The line segment d defines the distance between the lens and the optical sensor whereas h represents the height above ground. Due to the characteristics of the optical system, only the part of the sphere between the two tangential boundary points T_1 and T_2 is visible. The radius of the central body is called r_e, the radius of the circle in the image is r_b and the visible radius of the central body is r_a.

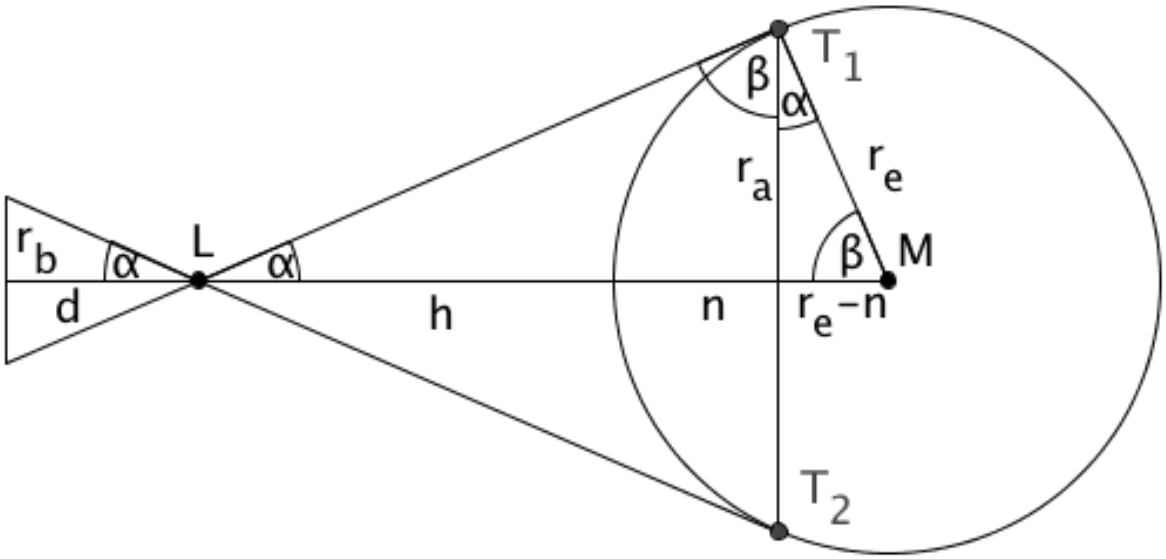


Figure 2.7: sketch of the optical system

It is obvious that

$$\tan\alpha = \frac{r_a}{h+n} = \frac{r_e - n}{r_a} \tag{2.26}$$

$$\sin\alpha = \frac{r_e - h}{r_e} = \frac{r_e}{h + r_e} \tag{2.27}$$

$$\Rightarrow n = \frac{r_a^2(h + r_e)}{r_e^2} - h \tag{2.28}$$

The theorem of intersecting lines states:

$$r_b = \frac{r_a d}{h+n} \tag{2.29}$$

Plugging formula 2.28 in 2.29 with use of the Pythagorean theorem

$$r_a^2 = r_e^2 - (r_e - n)^2 \tag{2.30}$$

shows:

$$r_b = \frac{r_e d}{\sqrt{h(h+2r_e)}} \tag{2.31}$$

To convert r_b in pixels, the width of one pixel of the optical sensor p must be given. The expected radius of the circle in pixels r_p is then

$$r_p = \frac{r_b}{p} \tag{2.32}$$

$$= \frac{r_e d}{p\sqrt{h(h+2r_e)}} \tag{2.33}$$

The following is assumed:

- let z be the margin of the radius check in pixels
- let h_{max} be the maximal height above ground (e.g. apogee of the orbit)
- let h_{min} be the minimal height above ground (e.g. perigee of the orbit)

The maximal radius $r_{p,max}$ and the minimal radius $r_{p,min}$ are then:

$$r_{p,max} = \frac{r_e d}{p\sqrt{h_{min}(h_{min}+2r_e)}} + z \tag{2.34}$$

$$r_{p,min} = \frac{r_e d}{p\sqrt{h_{max}(h_{max}+2r_e)}} - z \tag{2.35}$$

Decide as follows (with formula 2.23):

$$instruction = \begin{cases} skip, & \lambda_1 > r_{p,max} \\ accept, & r_{p,max} \geq \lambda_1 \geq r_{p,min} \\ skip, & \lambda_1 < r_{p,min} \end{cases} \tag{2.36}$$

2D Vector

The direction vector $\vec{v_d}$ from the center of the image s to the center of the 2D projection of the earth is then obviously (with formula 2.24 and 2.25) given by

$$\vec{v_d} = \frac{1}{\sqrt{\left(\lambda_2 - \frac{w_s}{2}\right)^2 + \left(\lambda_3 - \frac{h_s}{2}\right)^2}} \begin{pmatrix} \lambda_2 - \frac{w_s}{2} \\ \lambda_3 - \frac{h_s}{2} \end{pmatrix} \tag{2.37}$$

2.2.5 Height Calculation

The formula 2.33 already states a relation between the radius of the circle in the image in pixels and the height above ground. This equation is solved for h and reveals the height above ground for $h \geq 0$

$$h = r_e \left(\frac{\sqrt{d^2 + \lambda_1^2 p^2}}{p \cdot \lambda_1} - 1 \right) \tag{2.38}$$

where r_e is the radius of the central body, d is the distance between lens and optical sensor (cf. fig. 2.7) , p is the width of a pixel on the optical sensor and λ_1 is the radius of the circle in the image (cf. formula 2.23) .

2.2.6 Division

In case of the the image in fig. 2.6 the vector calculation (cf. 2.2.4) of λ_1, λ_2 and λ_3 delivers a circle similar to the one in figure 2.8. But the radius check registers the failed calculation and skips the image. This happens because the sun is in the image and the edges of the sun and the horizon overlap. But there is a part of the horizon's edge that is not influenced by the sun and would deliver a valid calculation. Thus, the image is divided in two parts every time a calculation failed by cutting the longer side in the middle. And that is also done for every resulting image until the image is either too small for further calculation or was skipped in the preprocessing step (cf. 2.2.1). In this example the image is divided two times and delivers four partial images (cf. figure 2.9). The image 4 is skipped by the preprocessing step in formula 2.3 because it is too

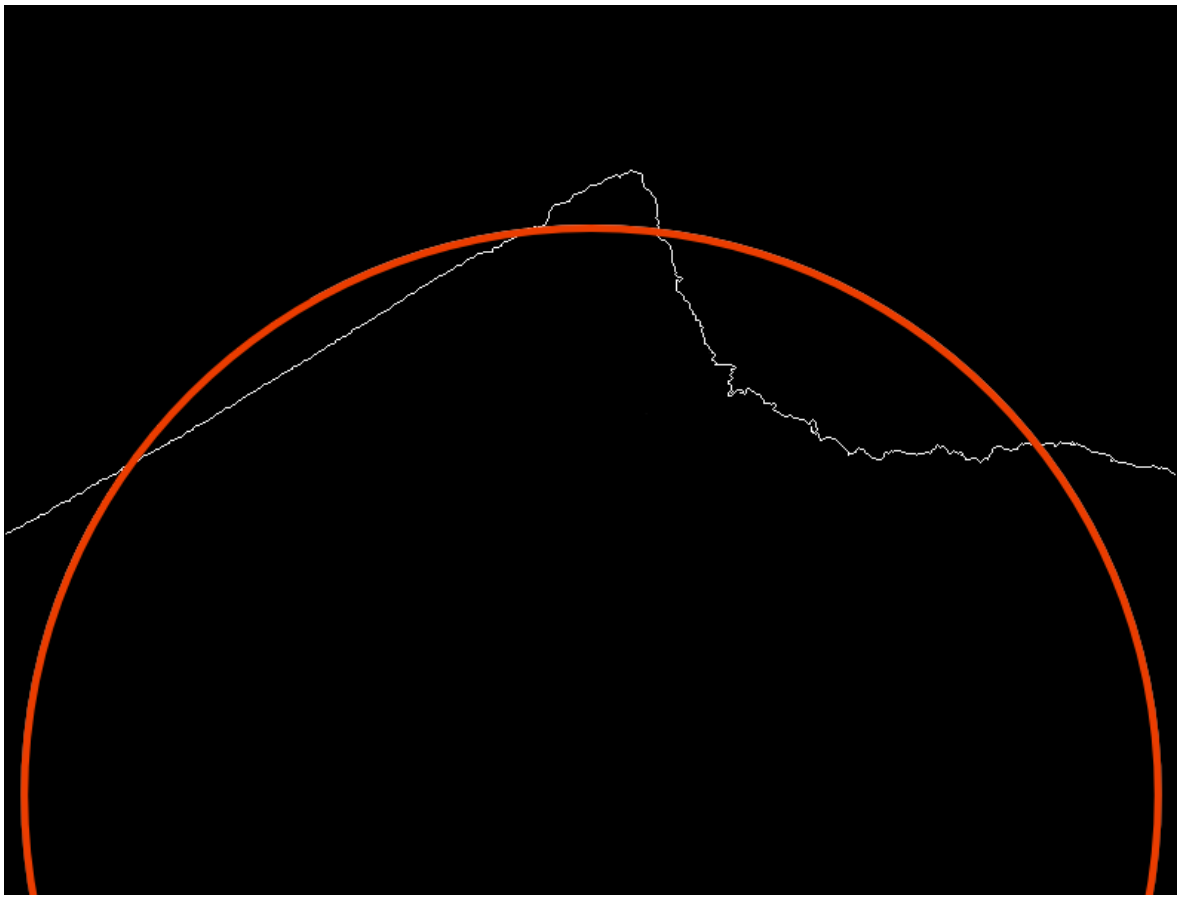


Figure 2.8: failed vector calculation of image 2.6

dark. Images 3 and 4 fail again during calculation, only image 1 (cf. 2.11) delivers a valid calculation. If there is more than one valid calculation, the result of the complete image is the median of all partial results (see fig 2.10).

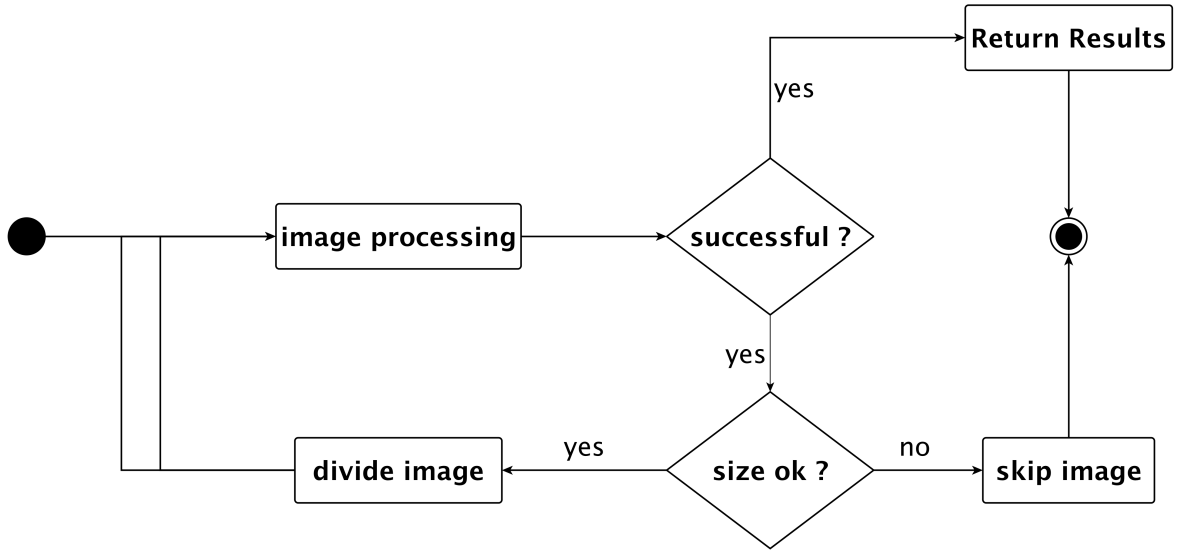


Figure 2.10: division activity diagram

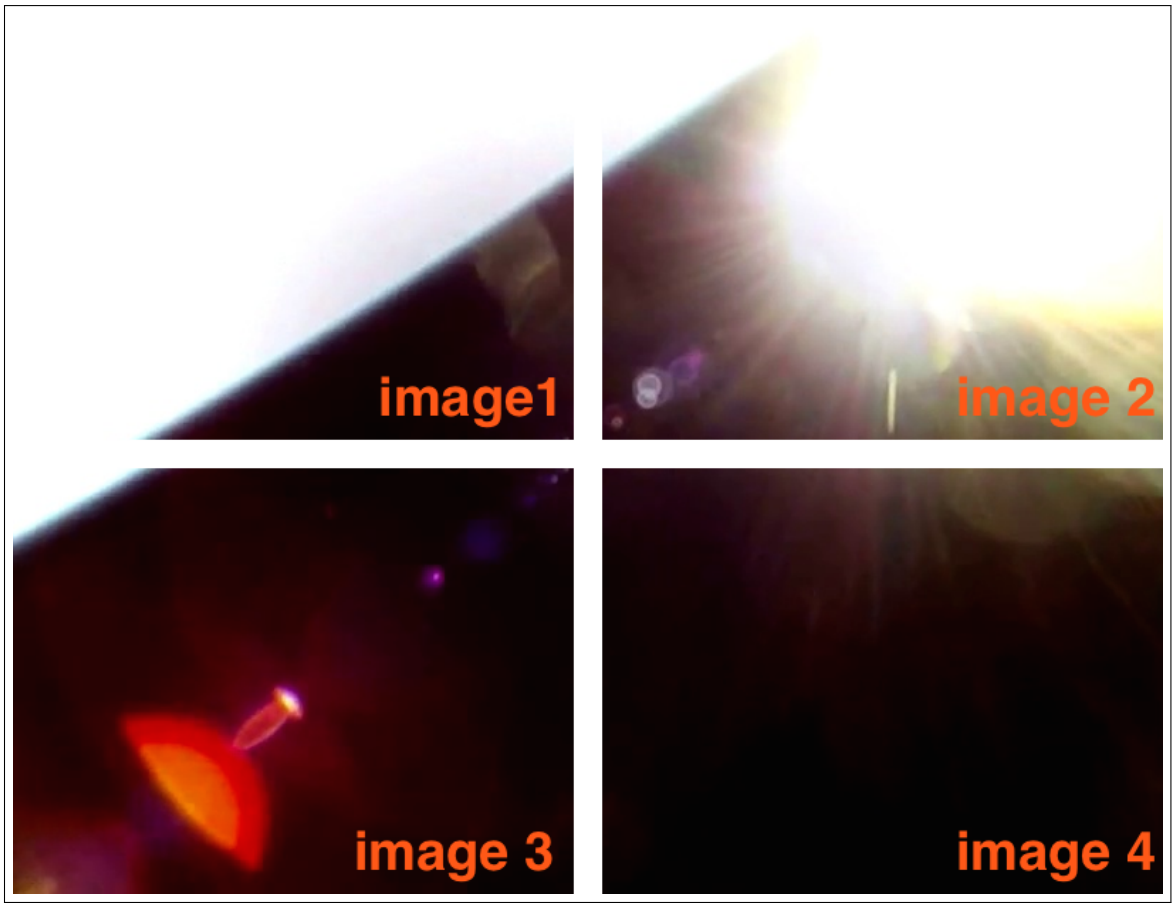


Figure 2.9: Image in 2.1 divided in four parts

The image is divided as follows:

- let the image s be the original image
- let the images t and u be the resulting partial images
- let $\delta \in \mathbb{N}^+, \delta \leq h_s \cdot w_s$ be the minimal number of pixels an image must have to be divided
- decide for image s as follows

$$instruction = \begin{cases} skip, & w_s \cdot h_s < \delta \wedge b_s > b_{max} \wedge b_s < b_{min} \\ accept & else \end{cases} \tag{2.39}$$

(2.40)

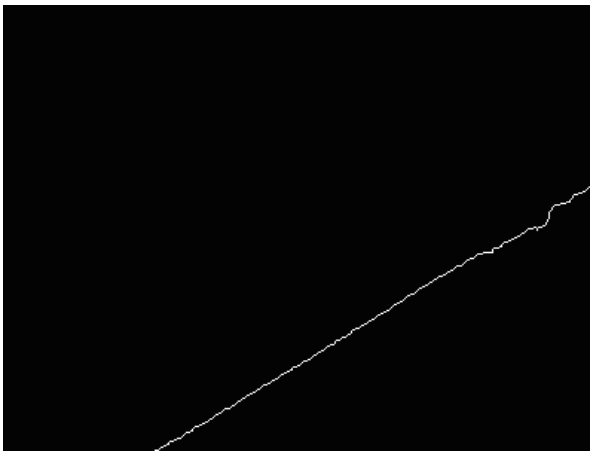


Figure 2.11: line detection of image 1 in fig. 2.9

- if $h_s < w_s$

$$w_t = w_u = \frac{w_s}{2} \tag{2.41}$$
$$h_t = h_u = h_s \tag{2.42}$$
$$f_u((x,y),0) \leftarrow f_s((x+w_u,y),0), f_t((x,y),0) \leftarrow f_s((x,y),0) \tag{2.43}$$
$$\forall x \in \mathbb{N}^+, x < w_u, y \in \mathbb{N}^+, y < h_u \tag{2.44}$$

- if $h_s \geq w_s$

$$w_t = w_u = w_s \tag{2.45}$$
$$h_t = h_u = \frac{h_s}{2} \tag{2.46}$$
$$f_u((x,y),0) = f_s((x,y+h_u),0), f_t((x,y),0) = f_s((x,y),0) \tag{2.47}$$
$$\forall x \in \mathbb{N}^+, x < w_u, y \in \mathbb{N}^+, y < h_u \tag{2.48}$$

- start with images u and t from the beginning (preprocessing)

3 HORACE Payload

3.1 OpenCV

"Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code. The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. [...] OpenCV's deployed uses span the range from stitching streetview images together, detection of swimming pool drowning accidents in Europe, inspecting labels on products in factories around the world [...] on to rapid face detection in Japan. It has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS." [Ope14, about] Especially because of its C++ interface, the Linux support and the wide range of algorithms, OpenCV is eminently suitable for an application in HORACE.

3.2 Hardware

3.2.1 Core System

The CS is the main computer of HORACE and provides among other functionalities the platform for the algorithm. The embedded computer, Pico-ITX MIO-2260 with an Intel Atom CPU, is connected to the CAM (see 3.2.2) via GigE Vision and stores data to a SSD using the Serial Advanced Technology Attachment (SATA) interface. The operating system Arch Linux is the basis for all tasks within the CS (cf. [RGW+14, 4.5.2]).

3.2.2 Camera

The camera which observes the outer environment is the industrial CMOS camera mvBlueCOUGAR-X102b manufactured by Matrix Vision. It provides the image data as consecutively and uniquely numbered frames via GigE-Vision interface to the CS (see 3.2.1) . This interface provides a fast data throughput to the CS. The CAM is set to deliver images with a resolution of (1024 x 768)px in 8bit RGB using automatic exposure. With a global shutter and a maximal blindness of 1/8.333 s after full illumination good pictures can be provided also under rough conditions like high spinning rates and looking regularly into sun (cf. [RGW+14, 4.5.1]).

3.2.3 Lens

The lens Lensagon BT8020N manufactured by LENSATION is screwed onto the CAM and can be focused to an object distance between 20cm and infinity. The focal length is 8mm and has a optical distortion less than -1.6% (cf. [LEN]).

3.3 Implementation

The HORACE algorithm was implemented using Apple's Xcode 5.1.1 on OSX 10.9.4 with OpenCV 2.4.9. Since the development started in a very early phase of the project when no hardware was available yet, the algorithm was designed as a stand-alone version to run on a Desktop PC and as a thread, embedded in the HORACE framework. By defining the preprocessor macro *DEBUG* in the build command and including a separate main-file *algorithm_main.cpp* the algorithm runs on the local machine including loading videos, showing results and creating videos. When compiling the complete HORACE project with the compiler g++4.7 on Ubuntu 12.04, the preprocessor macro *DEBUG* was omitted.
During implementation the algorithm in chapter 2 was developed. The steps preprocessing, threshold filter, line detection and vector calculation are all done in the function *process* in class *horace_algorithm*. OpenCV's functions *cvtColor* and *threshold* are used to convert the image first in a grayscale image and then to apply a dynamic threshold filter according to 2.2.2. The topological search was applied to the image by the function *findContours* in OpenCV with the parameter *CV_RETR_EXTERNAL* as mode to only detect outer borders. The circle fit and the vector calculation was implemented as described in 2.2.4. The step division is implemented with the function *computeFrame* and *devideAndConquer* as defined in 2.2.6 with one exception. The function *computeFrame* starts the calculation by scaling the image down to a defined size with OpenCV's function *resize*. This reduces the amount of data that has to be calculated but decreases the accuracy of the results. An optimal size of (533 x 400)px was determined experimentally.
The function *run* in the class *horace_algorithm* is an infinite loop that first captures a new image, starts the time measurement and then starts the actual calculation by calling the function *computeFrame*. In case of the stand-alone version, the time measurement is then stopped, the results of the calculation are shown and the procedure starts from the beginning. In the thread version the procedure is the same, except that the results of the calculation are not shown but saved and pushed to the downlink buffer.
The parameters are set according to table 3.1 and can be found in the file *globalDefines.h*. The parameters concerning the optical system have been copied from the data sheets of the lens (cf. [LEN]) and the camera (cf. [VIS]). The maximal and minimal height has been defined regarding the the expected position for promising image data. All other parameters have been determined experimentally.
The source files as well as the executable for OSX of the HORACE algorithm can be found in the appendix.

Table 3.1: parameters of the within the HORACE algorithm

Discription	**Name**	**Value**
dynamic threshold factor b_{factor}	THRESHOLD_FACTOR	0.75
width p of a pixel on the optical sensor	SENSOR_PIXEL_WIDTH	3.75 μm
minimal number of points γ in the horizon line	MIN_DOTS	5
distance d between the lens and the optical sensor	LENS_SENSOR_DISTANCE	8 mm
radius r_e of the central body	RADIUS_CENTRAL_BODY	6371.0 km
maximal height above ground h_{max}	APOGEE	100.0 km
minimal height above ground h_{min}	PERIGEE	20.0 km
maximal brightness value b_{max}	PREPROCESSING_ THRESHOLD_WHITE	235
maximal brightness value b_{min}	PREPROCESSING_ THRESHOLD_BLACK	5
margin of the radius check z	RADIUS_MARGIN	5000.0
minimal height an image must have to be divided	FIXED_CALC_IMG_HEIGHT	400

3.4 Evaluation

For the evaluation process after the flight, a special evaluation library was created whose main feature was to load images in the RAW format and export them as a video. Since the images are not fetched in a defined rate from the CAM but the video format AVI needs a constant rate, a program had to be developed that includes the time gaps between the images. Furthermore, the functionality to show saved calculation results was added to the program which can be found in the appendix with the name *raw2avi*.

3.4.1 Testing with Real Image Data

For the development of the algorithm and the testing of its performance, a video, taken during an earlier REXUS flight by the team EXPLORE, was used. Since no data about the optical system was available, the parameters for the maximal and minimal height above ground h_{max} and h_{min} were determined experimentally. The resulting video is available in the appendix (see table 6.1). The yellow line indicates the circle that represents the detected horizon. A red X means, that for this frame no vector was successfully calculated. The information about the radius and the center of the circle in pixels can be found in the bottom-left corner. The needed time per frame is also indicated but is not the time a calculation on the CS would take. The video is in the format (800 x 600)px and the simulation was executed on an MacBook Air with an 1.6 GHz, Intel Core 2 Duo processor, a 4 GB, 1067 MHz, DDR3 memory and the operating system OSX Mavericks using OpenCV 2.4.9. Besides, this video was taken with a GoPro which has a strong fish eye lens. Those distortions have been corrected manually but are still visible in some areas near to the edge of the image and may cause calculation failures.
Clearly visible is that the horizon is detected reliably if no disturbances like the sun

or lens flairs are in the picture. The position and radius of the circle are the same as if a human had mapped it. The amount of false positives and false negatives is very low. Even in images with disturbances, the horizon is correctly detected, however, the accuracy is lower.

3.4.2 Flight Data

Due to a camera failure described in [Rap14, 2.4] all image data is overexposed and a majority of images is completely white. At least the preprocessing step could be tested, since all image had to be skipped. According to [Rap14, 3.5.3] the number of false positives and the number of false negatives are both zero. As the distribution of

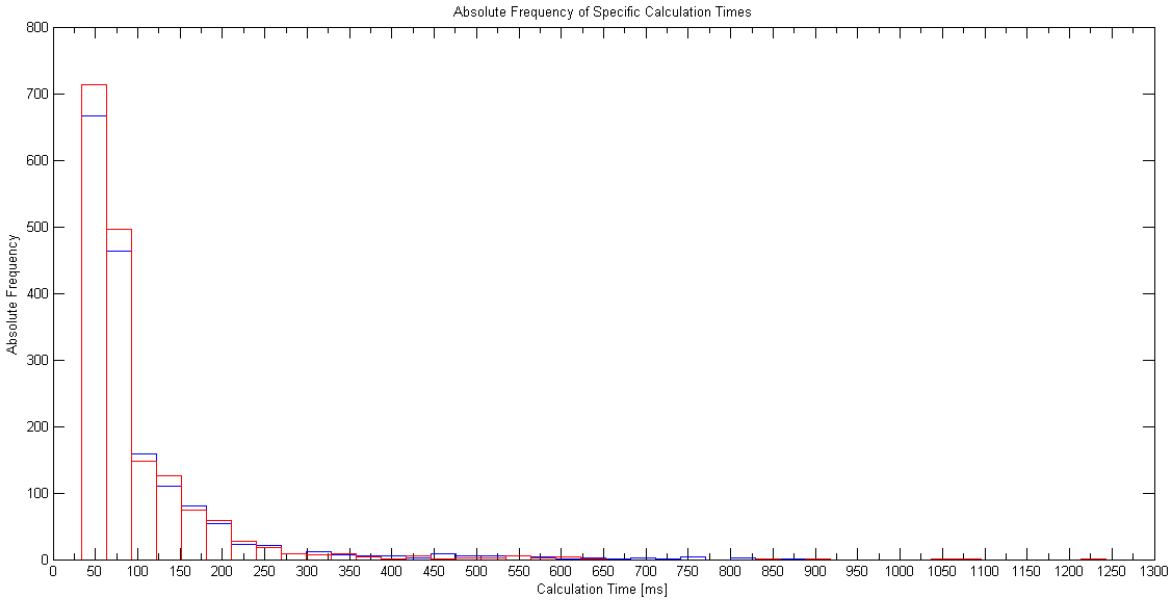


Figure 3.1: calculation times of images during flight (cf. [Rap14, figure 3-4])

the calculation time of the algorithm, illustrated in fig. 3.1, shows, are most frames calculated under 100 ms. Still, this number is not a reliable source to indicate the algorithm's performance since the images have not been completely calculated and the algorithm runs as thread on an operating system that saves a huge amount of data (raw image files) to a solid state disk (SSD) at the same time.

3.4.3 Simulation

The test with the video by team EXPLORE (see 3.4.1) showed that the approach works generally but the accuracy could not be determined. Therefore, a simulation video was created where the position, the orientation and the parameters of the optical system of the camera are predefined and thus known (cf. [Rap14, 3.5.4]). This video is also available in the appendix (see table 6.1). The angular difference between the vector given by the simulation and the vector calculated by the algorithm was determined and a plot of its frequency was created (fig. 3.2). An evaluation of this plot reveals that the accuracy of the direction vector is 0.6°, that means the requirement P-S-04 (see table 1.1) is accomplished. Furthermore, the length of the vector from the center of the image to the center of the 2D projection of the earth is compared with the same vector

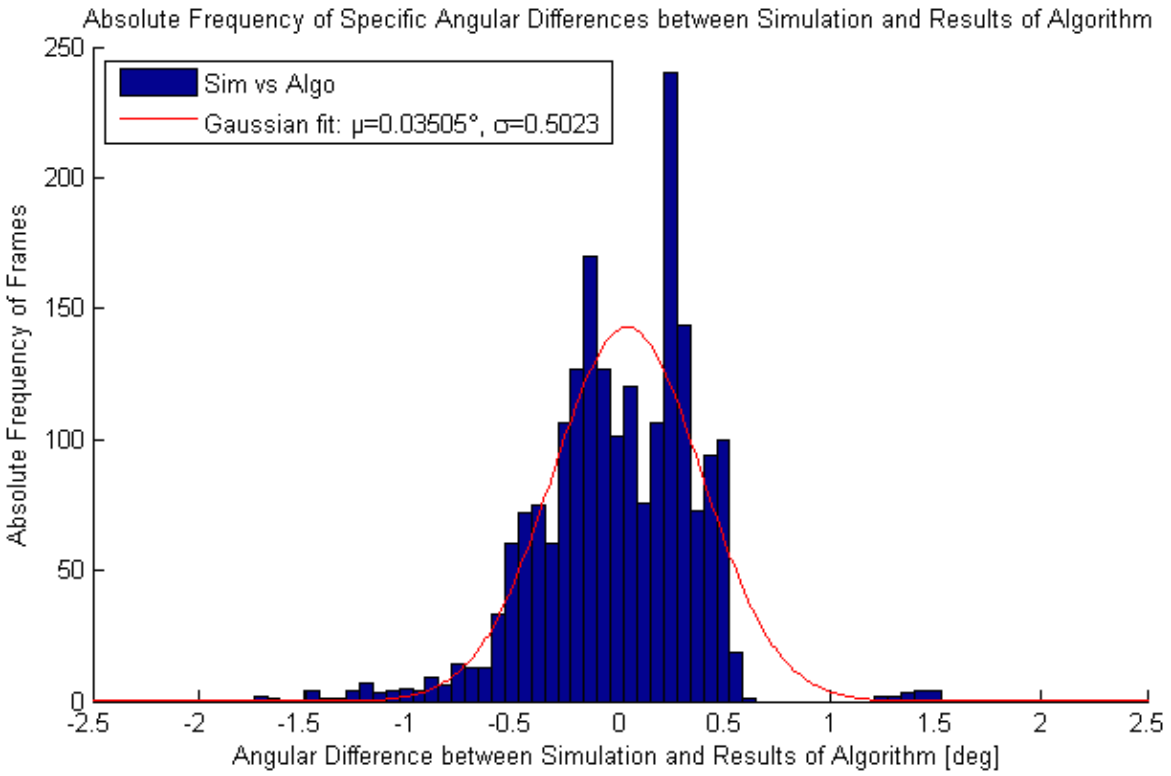


Figure 3.2: angular accuracy of the direction vector during simulation (cf. [Rap14, 3.5])

given by the simulation. Both values are plotted vs. the image ID and illustrated in figure 3.3. Although the values are not in the same unit and thus are not directly comparable, an effect of the dynamic threshold filter is clearly visible. The offset of the blue curve to the red curve is caused by the atmosphere. In the atmosphere the brightness smoothly fades to zero and it is, even for the human eye, hard to determine where the earth ends and the atmosphere starts. Thus, the threshold value is set to the brightness of a pixel somewhere in the atmosphere. This increases the radius of the circle in the binary image and thus the vector of the calculation. Since the dynamic threshold filter is used, the threshold value decreases if the overall brightness of the image decreases. The brightness is lower, the less of the earth is visible within the field of view and in this case the simulated vector is longer. That means, the higher the length of the simulation vector, the higher is the offset of the calculation vector. This effect is visible in figure 3.3 especially when comparing the frames 300 and 600. At frame 300 the offset is about 400 units, whereas the offset at frame 600, where the length is 400 km higher, is about 1300 units.
The ratio of false positives in this simulation was 0% and the ratio of false negatives was 10.29% which means the requirement P-S-10 (see table 1.1) is slightly not accomplished. The majority of false negatives occurred when the horizon was only slightly visible. Therefore it is assumed, that the amount of horizon points came below the minimal value (cf. [Rap14, 3.5.4]).

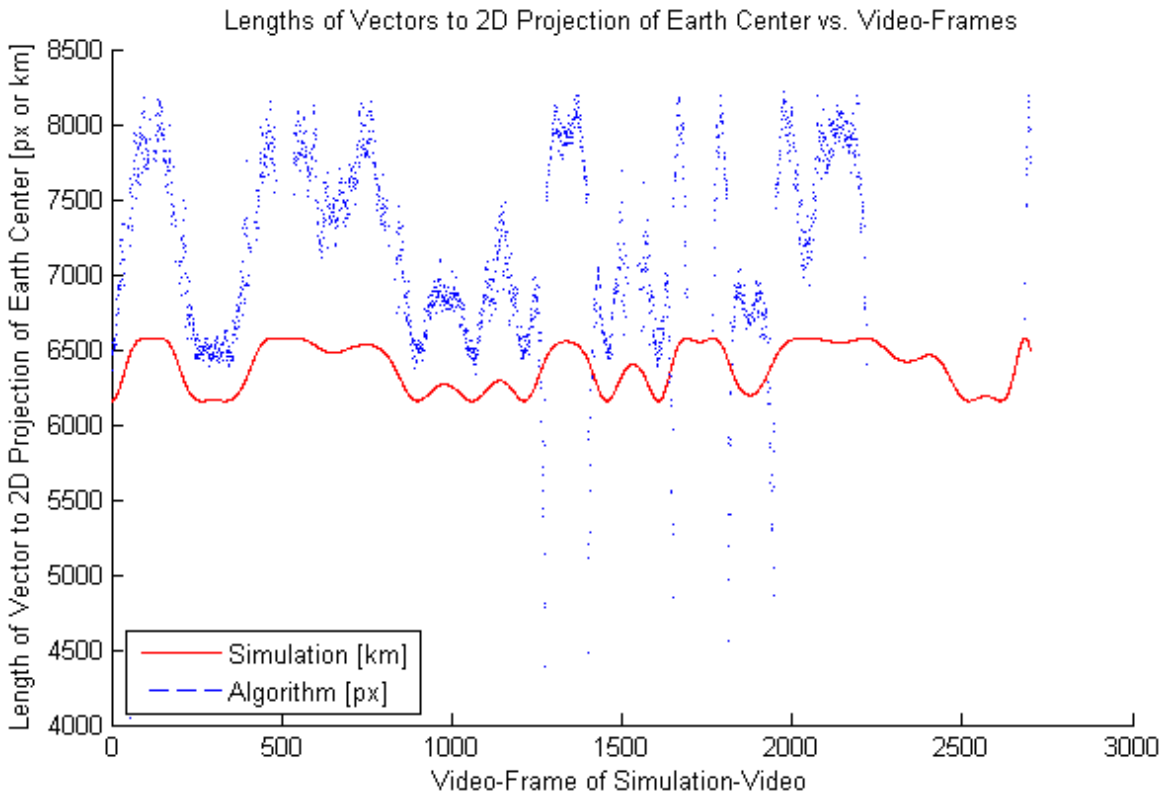


Figure 3.3: accuracy of the the length of the vector during simulation (cf. [Rap14, 3.5])

3.5 Documentation

A separate documentation for the HORACE algorithm and the evaluation library was not generated but the code was commented in detail.
All classes and variables belonging to the algorithm are placed in the namespace *algorithm* and the evaluation library in the namespace *evaluation*. Since the algorithm is not the only thread within the HORACE framework, an abstract class *ThreadClass* was created to unify all threads. The class *horace_algorithm*, which is the main class of the algorithm, is a derived class of *ThreadClass*. Here, all functions for time measurement, image fetching and image processing are defined. The class *ImageFrame* contains all information including the image data as OpenCV's image data type *Mat*. The class *PartialImageFrame*, which is derived from *ImageFrame*, represents a part of a divided image and contains, in addition to the information of the *ImageFrame*, information about its place inside the original image. At least, the class *Result* contains the results of a calculation including the horizon line.

4 The HorizonSensor

4.1 Further Development of HORACE

As mentioned in 1.3.3, HORACE was designed to implement a first prototype and to test the approach of a horizon sensor via image processing. To bring this system to the next level, which is an application in an embedded system, it is necessary to refine it regarding power consumption, mass, size, computational power consumption, accuracy, calculation speed and memory consumption. The last four items mentioned are issues of the software design and are approached in this work. OpenCV is a very powerful tool but was developed for PCs and needs a special run-time environment that is not simply portable to any embedded system. Thus, one of the tasks of this work is to implement all functions and data structures from OpenCV that are used in HORACE with C/C++ standard libraries only. But more important is to have an implementation that is easily reusable and extendable by multiple persons. So the main focus of this work is to create a platform for further development of the HorizonSensor and to create a first functioning algorithm.

4.2 Implementation

Since this algorithm is meant for an application in space systems, that are embedded systems, it is necessary to adapt the implementation to these systems. To accomplish this adaption the HorizonSensor is implemented regarding the Coding Directives v10.0 by Prof. Dr. Sergio Montenegro [Mon08]. That means in particular no run-time type information (RTTI), no exceptions, no inheritance and no data structures that allocate memory dynamically are used. Furthermore, the formal directives concerning the code structure were adopted to make the source code easier to understand. Since this library should function on as many systems as possible it is important that the implementation is compatible with as many compilers and dialects as possible. Therefore, it was made sure during development that the code is compilable with the dialects c++98, gnu++98, c++11 and gnu++11 using the C++ standard libraries libstdc++ and libc++. The tested compilers are g++4.6, g++4.7, g++4.8 and clang++503.0.40 on osx 9.4. The binary for each compiler is available in the appendix (see table 6.1). The development environment was Apple's Xcode 5.1.1 on OSX 10.9.4 with OpenCV 2.4.9. All source files of the HorizonSensor can be found in the appendix (see table 6.1).

4.2.1 User Interface

In the following context, the programmer embedding the HorizonSensor in his software is called user. The user needs to include the file *HorizonSensor.h* and keep all other files that belong to the HorizonSensor Library in the same directory as *HorizonSensor.h*.

The user interface is carried out with three classes *HorizonSensor*, *ImageFrame* and *Result*. The image data is passed as an *unsigned char ** along with the image width, the image height and the channels to the constructor of *ImageFrame*. The channel represents the number of colors that are used in the image data, e.g. in a RGB picture the number of channels is three. Due to the data format *unsigned char*, the color depth is limited to 8 bit. The image data is a pointer to an array containing the image information in the following format. All pixels are subsequently ordered row-by-row, as if one would take every row of the image from top to bottom and put them in one line. The pixel itself has its channels ordered also subsequently. An RGB image would then result in an array like $R_1G_1B_1R_2G_2B_2R_3$.... This *ImageFrame* is passed to the method *findHorizon(ImageFrame)* in *HorizonSensor*. The function executes the calculation on the image data given in ImageFrame and returns the result as a *Result* object. There, the user can find getter for the direction vector in pixels, the height above ground in meters, the radius in pixels and the center in pixels.

Figure 4.1: changing parameters

```
HorizonSensor sensor;
sensor.param.setThresholdMethod(STATIC);

ImageFrame frame = ImageFrame(myImageData,
                              myWidth,
                              myHeight,
                              myChannels);

Result result;
result=sensor.findHorizon(frame);

Vector2f vec;
vec=result.get2DDirectionVector();
```

Figure 4.1 shows an example on how to use the HorizonSensor. To adapt the algorithm for a certain application it is essential to be able to easily change parameters and methods. In this implementation it is possible to adjust the calculation by simply calling one function with one parameter even during run-time. In the shown example the user changes the method, that converts the image into a binary image, to the *STATIC* method (line 2 in fig. 4.1). The HorizonSensor then executes the *STATIC* threshold method (cf. 2.4) for all successive calculations (see figure 4.3).

4.2.2 Code Structure

In this example the modular structure of the algorithm is nicely visible. Every step in the calculation can be done somehow different and it depends on the application which of them is the most suitable. Thus, there are functions called super functions in the following context, that have got the task to execute the chosen method by calling the corresponding function which is referred to as subfunction. In case of the threshold filter there are currently two methods available, the *STATIC* and the *DYNAMIC* one (for more information see section 2.2.2). Every single step has a corresponding

Figure 4.2: threshold method enumeration

```
/*!
 \brief defines all possible threshold methods
 */
enum ThresholdMethod{

    /*!
     \brief the threshold value is set according to the
         brightness of the image
     */
    DYNAMIC,
    /*!
     \brief the threshold value is fixed
     */
    STATIC,

};
```

enumeration (here *ThresholdMethod* see fig. 4.2), a super function (here *thresholdFilter(ImageFrame& _imageFrame)* see fig. 4.3) and an entry in the *Parameter* class. When the HorizonSensor executes the threshold filter (super function), the function chooses the method (subfunction) according to the value in *Parameter*. All parameters are accessible through setters and getters in the class *Paramter* which is a member variable (*param*) in *HorizonSensor*.
This procedure also makes it easy to add new methods because it only requires a new entry in the enumeration (see figure 4.2) and another case in the switch statement (see figure 4.3).

4.2.3 Overall Design

Since a detailed description is given in the appendix, only the main structures are explained here. The complete HorizonSensor (including HorizonSensorTest) is situated in the namespace *hs*. The class *HorizonSensor* is basically a library of functions that are needed for the calculation (see 4.5). The function *findHorizon()* is the only public member function (except the constructor) because this function is executed by the user. All other functions are executed internally by the function *processFrame()*. Because there are several methods for every single step of the algorithm (as mentioned in 4.2), every step has a corresponding function that executes the method chosen by the user. In the following context these methods are called subfunctions and functions that execute the subfunctions are called super funtions. Every subfunction performs a modification on the *ImageFrame* it gets passed. For example, the subfunction *determineRandomBrightness()* determines the brightness of the image and stores the result in the *ImageFrame*.
The subfunction uses the parameters, given in the public member variable *param* to perform the modification.All parameters in the class *Parameter* are accessible through setters and getters, hence it is easy for the user to find and change a certain parameter.

Figure 4.3: super function of the threshold filter

```
}

bool HorizonSensor::thresholdFilter(ImageFrame& _imageFrame
   ){

    switch (param.getThresholdMethod()) {

        case DYNAMIC:
            return dynamicThresholdFilter(_imageFrame);
            break;
        case STATIC:
            return staticThresholdFilter(_imageFrame);
            break;

        default:
            return false;
```

The class *ImageFrame* (see fig. 4.4) contains all information about an image and provides functions to access this data. Because this class is part of the user interface, all functions that modify calculated data are set to private. This requires the class *HorizonSensor* to be a *friend class* of the class *ImageFrame*.
The private member variable result of the type *Result* is accessible via a getter and contains the results of the calculation of this *ImageFrame*. The results and the used parameters are then accessible with getters.
Points and vectors are implemented with integer and floating point values in two and three dimensions with a separate class each (*Vector2i*, *Vector2f*, *Vector3i*, *Vector3f*,*Point2i*, *Point2f*, *Point3i*, *Point3f*). They are equipped with operators for addition, subtraction, multiplication, division and comparison where reasonable. The class *RegionOfInterest* serves as a data interface for the subfunction *topologicalSearch()*. Since this function needs different data structures than given in *ImageData* the class provides all needed data structures and contains functions to simplify the data access.

4.2.4 Preprocessing

Since the preprocessing step consists only of the brightness determination, one super function named *determineBrightness* and one subfunction *determineRandomBrightness* was created. Since the image data in *ImageFrame* is accessed with the parameters x-coordinate, y-coordinate and the channel, the algorithm was changed. Instead of using one random number that represents one pixel, two random numbers for each of the coordinates were used. The parameter a was implemented as *float* and named *brightnessReferencePixelRatio*. The maximal and minimal brightness values, b_{max} and b_{min}, are named *brightnessThresholdWhite* and *brightnessThresholdBlack*. The result of the *brightness* calculation is stored in the variable *brightness* in class *ImageFrame*.

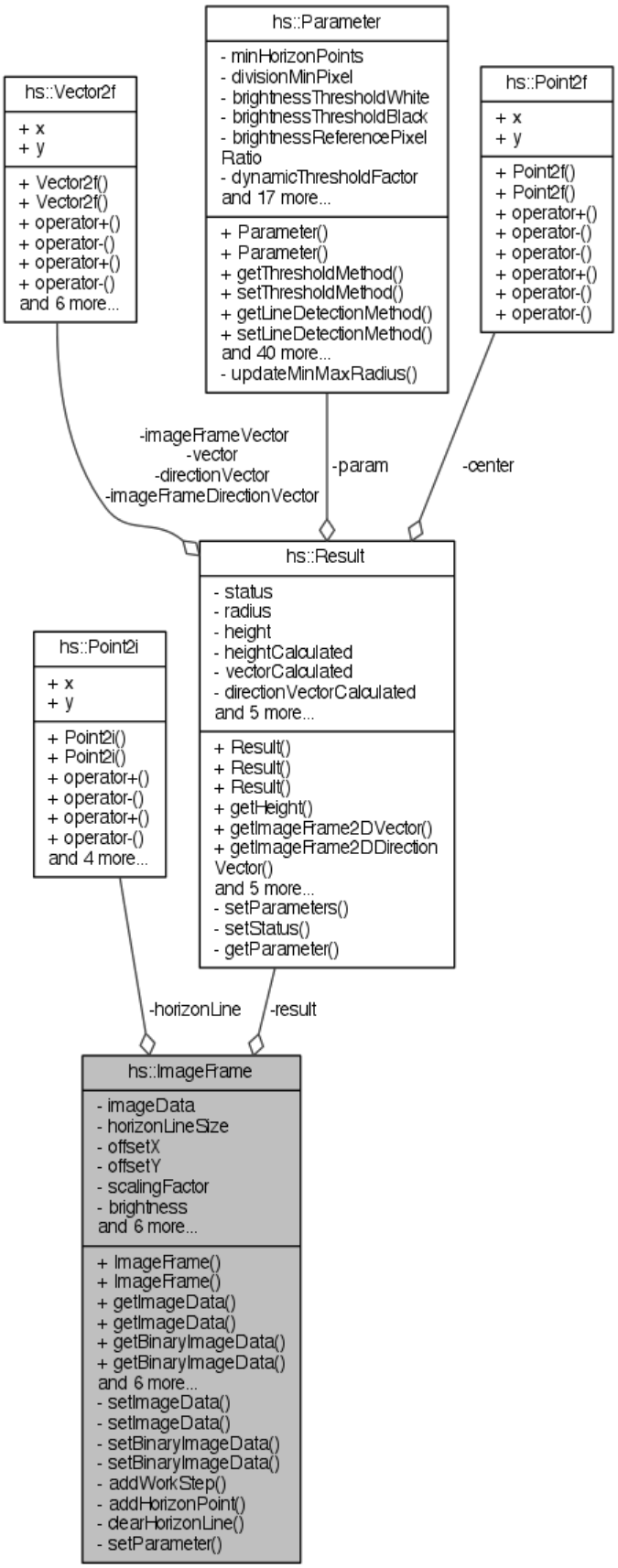


Figure 4.4: Collaboration diagram for class ImageFrame

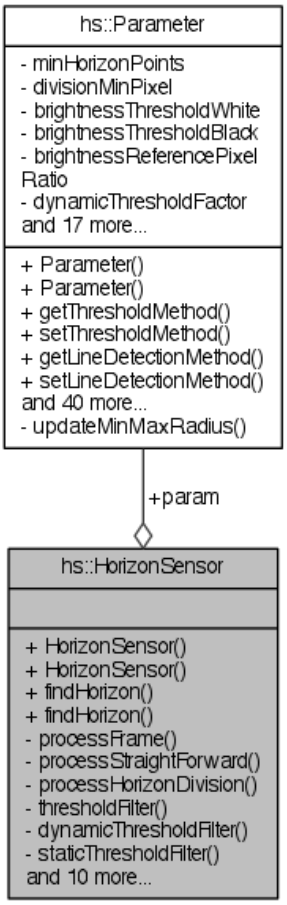


Figure 4.5: collaboration diagram for class HorizonSensor

4.2.5 Threshold Filter

For the step threshold filter the super function *thresholdFilter*, the subfunctions *staticThresholdFilter* and *dynamicThresholdFilter* have been created. Both subfunctions use in-situ memory, that means they modify the existing array *imageData* in *ImageFrame* and no additional array is created. The threshold value b_{thresh} of the static method is named *staticThresholdValue* and the threshold factor b_{factor} of the dynamic method is named *dynamicThresholdFactor*.

4.2.6 Line Detection

The line detection has one super function called *lineDetection* and two subfunctions called *topologicalSearch* and *erodeImage*. Since the focus of this work is to create a platform for further development the implementation of the algorithm 2 of [SA83] was waived. The method used in *topologicalSearch* is the algorithm 1 in [SA83] but with the extensions mentioned in 2.2.3. That means the topological search detects not only outer borders but also inner borders. This must be changed while further development. The neighbourhood edge length g of method erode is named *erodeEdgeLength* and the threshold value *a erodeThresholdValue*. The horizon line is stored in the array *horizonLine* in *ImageFrame*.

4.2.7 Vector Calculation

The step vector calculation is only partially done in class *HorizonSensor*. The calculation of the 2D direction vector is done in class *Result* because there are several results available but not all are always needed. This way, only the results that are utilized are calculated and some calculation time is saved. The other two substeps *circle fit* and *radius check* are implemented in the subfunction *circleFit* which is called by the super function *calculateVector*. The radius λ_1 is stored in the variable *radius* in class *Result* and the center in variable *center* in class *Result*. The parameters of the formulas 2.34 and 2.35 are named as follows:
The minimal number of points in the horizon line γ is named *minHorizonPoints*. The margin of the radius check in pixels z is named *radiusPixelMargin*. The maximal height above ground h_{max} is named *apogee*. The minimal height above ground h_{min} is named *perigee*. The radius of the central body r_e is named *radiusCentralBody*. The width of one pixel on the optical sensor p is named *sensorPixelWidth*. The distance between the lens and the optical sensor d is named *lensSensorDistance*.

4.2.8 Division

Since this step only increases the performance of the algorithm regarding false negatives, the implementation was waived. Only the function that divides an image is implemented and tested. This step is meant to be implemented as subfunction *processDivideImage* for the super function *processFrame*. Instead a subfunction named *processStraightForward* that executes all steps subsequently without any division was implemented.

4.2.9 Height Calculation

The step height calculation is implemented in class *Result* and named *getHeight*.

4.3 Evaluation

To test the HorizonSensor a specially designed test environment - HorizonSensorTest - was implemented.

4.3.1 HorizonSensorTest

HorizonSensorTest uses the OpenCV framework (see 3.1) to load, show and store images and videos, whereas the HorizonSensor library itself is completely independent from this framework. OpenCV can easily be installed on Linux and OSX but also on Windows. On Linux the installation is done via the packet managing tool e.g. in Ubuntu apt-get, see fig. 4.6.

Figure 4.6: installing OpenCV on Linux

```
#linux
sudo apt-get install libopencv-core2.4 libopencv-highgui2.4
    libopencv-imgproc2.4
```

On OSX the installation using MacPorts is the easiest way. Therefore the free command line tool MacPorts (www.macports.org) must be installed and the command in figure 4.7 be executed.

Figure 4.7: installing OpenCV on OSX

```
#osx
sudo port install opencv
```

After the installation the HorizonSensorTest binary can be executed via command line with a variety of parameters in UNIX-style.

Figure 4.8: HorizonSensorTest parameters

```
[INFO] Parameters are:
[INFO]  -iv:    input video file
[INFO]  -ip:    input picture file or directory
[INFO]  -t:     Test case
[INFO]  -l:     log file
[INFO]  -ov:    output video file
[INFO]  -op:    output picture file or directory
[INFO]  -h:     Show this message
```

As stated in figure 4.8 the input can be a video of any type (*-iv*), a picture or a folder with pictures of any type (*-ip*). One of these options is mandatory. It is also possible to save the output as pictures (*-op*) or as a video (*-ov*), but this is optional. The test case can be chosen by using the *-t* option which is also mandatory. The available test cases can be displayed with the *-h* option. By using the *-l* option a log file is created. An example for the usage of HorizonSensorTest is given in figure 4.9.

Figure 4.9: HorizonSensor parameters

```
./HorizonSensorTest_osx9.4_clang503.0.40 -t circlefit -iv /
   Path/to/my/input/video.avi -ov /Path/to/my/output/video.
   avi -l /Path/to/my/logfile.txt
```

The test environment's basis consists of three classes *TestBench*, *TestCase* and *StopWatch*. The *TestBench* handles the user input and executes the chosen test case. The *TestCase* is an abstract class for all test cases and provides functions to get, show and save images as well as to simplify the debugging process during testing. The class *StopWatch* implements the functionalities of a stop watch. The user can start, stop and reset the counter in milli seconds accuracy. To add a new test case a new class that inherits from *TestCase* must be created and the functions *run()* and *getName()* must be overwritten (see fig. 4.10).

Figure 4.10: example of a new test case

```
class MyTest : public TestCase {

public:
    string getName(){ return "MyName"; }
private:
    bool run();

};
```

The function *getName()* must only return a string containing the name of the new test case. In the function *run()* the complete testing is done using the functions *getNextImage(Mat& _destination)* to get images and *showImages(vector<Mat> _images)* to show and store images. Eventually, the new test case must be included in the file *TestCases.cpp* and added to the the list of *TestCase*s with the line in figure 4.11 in the same file. If using private members of the HorizonSensor classes within the test, it is necessary to declare the test class as *friend class* in the used class.

Figure 4.11: adding *TestCase*s

```
testCases.push_back(new MyTest);
```

Some example tests have already been implemented and are available along with the HorizonSensorTest environment files in the appendix (see table 6.1).

4.3.2 Full Test

To test the HorizonSensor interface, as a user would use the library, a new test case, class *FullTest*, was implemented and is available along with the video in the appendix. The test is very similar to the code in fig. 4.1 with the following extensions. The topological search is used although it is not fully implemented but for this test satisfiable works, since the static threshold filter is used and the threshold value is set to 10 which is very low. The image data is the same as in the simulation for the HORACE algorithm in 3.4.3. In addition, a test with an image where the earth is fully visible (fig. 4.12) was conducted. The machine, this test was executed on, was a MacBook Air with an 1.6 GHz, Intel Core 2 Duo processor, a 4 GB, 1067 MHz, DDR3 memory and the operating system OSX Mavericks. The test revealed that the HorizonSensor

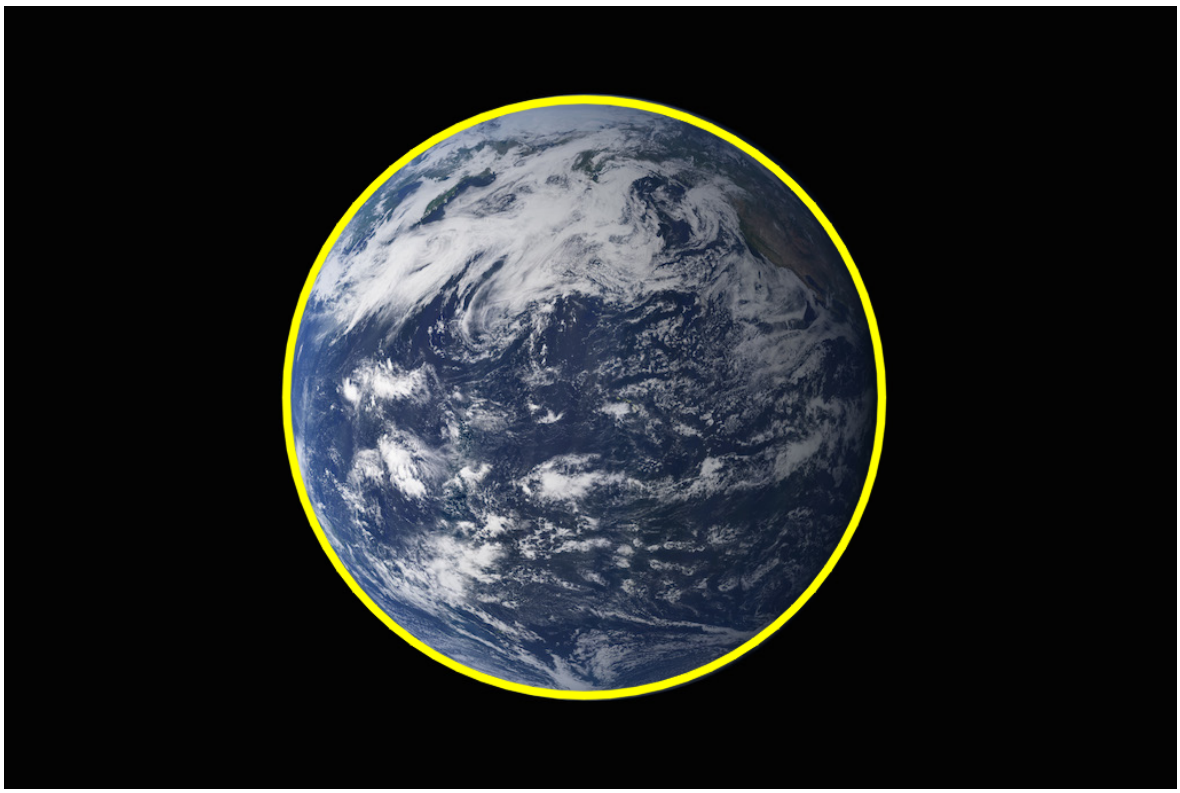


Figure 4.12: test on image with fully visible earth (source: NASA)

already works although not all steps are fully implemented. This is only possible because the simulation does not include image disturbances and the threshold value is set that low, so no inner borders can arise. For the test of the full earth image the parameter h_{max} was experimentally determined and the results are also excellent as can be seen in figure 4.12.

4.3.3 Speed Test

To compare the subfunctions regarding their calculation speed and to identify those steps with high calculation time impact, a test case - class *SpeedTest* - for the HorizonSensorTest environment was implemented. The used machine is the same as in 4.3.2. Thus, the absolute values of the measured time are only partly meaningful. However, comparing the values relatively to each other helps to compare functions and steps.

Table 4.1: average calculation time of all sub functions

Sub Function	**Avg. Calc. Time**
determineRandomBrightness	16 ms
staticThresholdFilter	89 ms
dynamicThresholdFilter	96 ms
erodeImage	360 ms
topologicalSearch	223 ms
circleFit	0 ms

Table 4.1 shows the results of this test, using the video that is also used in 4.3.2 and 3.4.3. The test clearly reveals that the determination of the brightness, in comparison with the other functions, is much less time consuming and is thus a qualified preprocessing method. According to this simulation the difference between the static and the dynamic threshold filter is very small but still noticeable. A much higher difference could be measured between the erode method and the topological search of the line detection. The topological search is 1.6 times faster than its competitor even though this method is not in its final implementation (algorithm 2 in [SA83]) which works, according to [SA83], even faster. Very remarkable is the result of the circle fit whose time consumption was lower than one millisecond. Comparing the steps results in a very clear statement: the most time consuming step is the line detection whereas the vector calculation costs almost no time. This knowledge can be used to optimize the HorizonSensor regarding calculation speed by concentrating the optimization on the line detection. Accuracy improvements could be achieved by reapplying the vector calculation step - since there are only small costs - to partial horizon lines.

4.4 Documentation

The code of the HorizonSensor is documented according to [Mon08]. That means every namespace, class, method or variable is tagged with doxygen comments. The doxygen tool, also recommended by [Mon08], is a free tool to create clear documentations for all kinds of programming languages and is available at www.doxygen.org.
It was used to auto generate an html and a pdf documentation which are available in the appendix (6.1). The documentation was written in a way that the user is able to utilise the HorizonSensor without knowing the internal procedures and structures but is also able to modify the algorithm if necessary. A special documentation for the HorizonSensorTest environment was not created but the functions, seen by the user are documented in detail.

5 Conclusion

Unfortunately, the most significant test of this new approach, the test under space conditions, was, due to a malfunctioning component (cf. [Rap14, 2.4]), only partly successful but that does not mean that the project was futile. On the contrary, HORACE produced a promising prototype of a functioning horizon sensor, even if this approach is not ultimately proven yet. The part of the software that was tested, that is the rejecting of bad images, worked excellently (cf. 3.4.2). During tests with real footage the algorithm showed an sovereign management of images with high disturbances (cf. 3.4.1) and a simulation determined an accuracy of 0.6° (cf. 3.4.3). Regarding that this sensor is still in an very early stage, it is presumable that it can keep up with conventional horizon sensors whose accuracies are between 1° and 0.05° (cf. 1.1). Although the HorizonSensor is not as sophisticated as its predecessor yet, it is, especially in combination with its purpose-built test environment HorizonSensorTest (cf. 4.3.1), a very powerful and helpful platform for further development. Even though there is some work to do, especially regarding calculation speed, it is worth the effort, because this sensor system could be, due to its reliability in stress conditions, a valuable extension to the current set of attitude sensors.

6 Outlook

Since this approach is not proven under space conditions yet, another attempt is already initiated to do so. A new team from the University of Würzburg is preparing the proposal for an experiment on REXUS 19/20, that ties on the work done by HORACE and this bachelor thesis. Part of that follow-up experiment might also be a further development of the software especially regarding accuracy and output data rate and the implementation of a second attitude system, like gyros, to verify collected attitude information. In this context, a step that scales the image down to a smaller image, could be added, to decrease the amount of data that is calculated. Furthermore, a different approach of the division step could be implemented, where, in contrast to the existing algorithm, not the complete image is divided and recalculated but only the horizon line is divided and the vector calculation is repeated. Since the vector calculation is very fast in comparison to the steps threshold filter and line detection, this new division method could save calculation time. Due the huge impact of the threshold value on the accuracy of the calculation, a threshold controller, that autonomously adjusts the threshold value according to the results of the last image, could be developed. Also, new test cases for the HorizonSensorTest environment could be developed to integrate the accuracy analysis as shown in [Rap14, 3.5.4] and thus to simplify the access to test results. New simulations with more realistic effects regarding atmosphere, sun, lens flairs and eclipse phases could improve the understanding of the impact of these effects on the algorithm's performance. Moreover a conversion of the direction vector to Euler angels could be implemented to offer another alternative to the user.
As long term evolution, an application in a Cubesat of the University of Würzburg is planned. Therefore, more improvements especially regarding mechanics and electronics have certainly to be made. Since the University already has experience with star sensors (STELLA), a combined sensor is an interesting idea, although or right because it is bold.

Bibliography

[Lea14] Least squares circle calculator, 2014. URL: http://www.had2know.com/academics/best-fit-circle-least-squares.html.

[LEN] LENSATION. Specification lensagon bt8020n. URL: http://www.lensation.de/images/PDF/BT8020N.pdf.

[LWH09] W. Ley, K. Wittmann, and W. Hallmann. *Handbook of Space Technology*. American Institute of Aeronautics & Astronautics. Wiley, 2009. URL: http://books.google.de/books?id=6dGFPwAACAAJ.

[Mon08] Prof. Dr. Sergio Retana Montenegro. C++ coding directives, version 10.0, 2008.

[Ope14] Opencv website @ONLINE, June 2014. URL: http://opencv.org.

[Rap14] Thomas Rapp. *Development and Post-Flight-Analysis of HORACE the Horizon Acquisition Experiment*. Julius-Maximilian University Wuerzburg,Am Hubland, D-97074 Wuerzburg, 20. August 2014.

[RGW+14] Thomas Rapp, Sven Geiger, Florian Wolz, Matthias Bergmann, Arthur Scharf, and Jochen Barf. Horace student experiment document v 4.0. This document was written in the context of a REXUS project, 2014.

[RX/14] REXUS/BEXUS programme, June 2014. URL: http://rexusbexus.net.

[SA83] Satoshi Suzuki and Keiichi Abe. *Topological Structural Analysis of Digitized Binary Images by Border Following*. Shizuoka University, Hamamatsu 432, Japan, 16. December 1983.

[VIS] MATRIX VISION. Technical details mvbluecougar-x. URL: http://www.matrix-vision.com/GigE-vision-kamera-mvbluecougar-x.html?file=tl_files/mv11/support/mvBlueCOUGAR/Datasheets/mvBlueCOUGAR-X_technical_details_2014-07.pdf.

List of Figures

List of Tables

Appendix

Table 6.1: appendix files

Appendix	**File**
HorizonSensor documentation	HS_DOC.PDF
HorizonSensor HTML documentation	HS_HTML_DOC.ZIP
HorizonSensor source code	HS_SOURCE.ZIP
HorizonSensorTest source code	HST_SOURCE.ZIP
HorizonSensorTest executables	HS_EXECUTABLES.ZIP
HORACE executables	HORACE_EXECUTABLES.ZIP
HORACE source code	HORACE_SOURCE.ZIP
HORACE evaluation executables	HORACE_EVAL_EXECUTABLES.ZIP
HORACE evaluation source code	HORACE_EVAL_SOURCE.ZIP
test videos	TEST_VIDEOS.ZIP

Declaration Of Authorship

I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University. Formulations and ideas taken from other sources are cited as such. This work has not been published.

Würzburg August 30, 2014

Signature