Star Tracker Lens Quality Assessment

30330 Image Analysis with Microcomputer

s132275 - Roberto Prestigiacomo
s180210 - Kasparas Simkus

Lyngby 2018

DTU Space
National Space Institute
Technical University of Denmark

Elektrovej, building 327+328 + 371 and
Ørsteds Plads, building 348
2800 Kgs. Lyngby, Denmark
Phone +45 4525 9500
www.space.dtu.dk

Contents

CHAPTER 1

Introduction

With the increase of space missions and objects sent in orbit, it becomes apparent that extremely accurate positioning determination has to be done on board. This is especially true as the distance from earth increases and technique like laser beacon becomes inefficient. Luckily such system exists and it is called *star tracker*. It consists of a camera on the spacecraft that acquire images of star fields and compare them to images already available to the system to provide data for orientation control of a spacecraft in $\mu 30s$ with accuracy of 0.7 arcseconds [1]. When position of stars is accurately and precisely identified the spacecraft can maintain desired orientation to ensure that any other direction sensitive instruments have optimal working conditions.

High quality results demand high quality production of instrument. To ensure that each camera has optimal focus and durability rigorous testing is performed. Lens focusing is crucial part of this process. A badly focused lens may introduce aberrations that make it more difficult to detect star position precisely. These tests require a lot of time, since camera has to be evaluated over multiple images and every adjustment compromises structure as it requires dis assembly, of certain degree, to adjust lens. Therefore it is desirable to have a better method that can improve efficiency of calibrating cameras.

The purpose of this paper is to examine a method for providing an assessment of the quality for star tracker camera lens focus based on images captured by a real system. A prototype program will be developed that will try to identify some forms of aberrations on images.

Principal component analysis will be used to determine the eigenvalues of the strongest star aberration which will be used to provide a quality value for the lens.

1.1 Problem statement

This work aimed at determining how accurately a lens focus ability can be quantified by using *Principal Component Analysis* (PCA) on regions holding a single star using data sets acquired with a real star tracker camera system.

This can be further broken down into the following sub problems:

- What preprocessing is necessary for PCA to properly work?
- What values of PCA are used to determine focus?
- How to visualize results for easy comprehension?

1.2 Limitations

Due to the fact that time frame to carry out this project was limited as well as the human resources (only two group components), a prototype is considered as a success. This means that the solution implemented can not be the most optimal, but other methods will be considered.

In addition, even though the course is called ”Image Analysis with Microcomputer” the performance on systems below a typical laptop will not be considered, and no investigation towards improving performance will be done. In-depth analysis of results might not be performed and a as a final result visualization of focus for image together with documentation of process will be provided.

CHAPTER 2

Theory

2.1 Scene analysis

For this project scene analysis can not be performed normally. Description of hardware will be given together with factors and interference that determine image quality.

Star tracker cameras are mostly made using *Charge Coupled Device* (CCD) [2].

- *Pixel* - pixels on CCD are made out of gate terminals that are on top of an insulator. The insulation layer prevents electrons from reaching the gate, since they are held at positive potential compared to the rest of the device they simply pool near gate. Beneath gate and insulator, photons coming in hit a silicon layer where due to photoelectric effect free electrons are created. The more light hits a pixel, the more electrons will be created. This also means that the larger each individual pixel is, the more light it can capture. These electrons are gathered into "wells" by gate which are then shifted towards the read out that is found on side of CCD.

- *Hardware* - Output of CCD pixel is read by analog to digital converter which measures voltage that was created due to photoelectric effect. To fully utilize sensor A/D must have a dynamic range that is the same as the sensor's. Electron values are read out at one side of CCD sensor in serial mode. They are shifted from top to bottom. A high end CCD will be split into multiple sections and each one will be read out at the same time to increase speed of conversion.

- *Dynamic range* - Each pixel of CCD can hold a set amount of electrons that is determined by physical size. This can vary between $10ke$ to $500ke$. If pixel is too saturated then it will spill over to neighbouring pixels and introduce *blooming*. This will degrade the final image since pixels will be capturing not only the intended light but also artifacts from saturated pixels.

- *Operation temperature* - if temperature is too high, it can excite electrons in the insulation layer and create unwanted electrons in it. This is called *dark current*. For every 6-7°C of cooling, there is about a 2X reduction in the total dark current generation rate. And cooling below -100 °C removes noise almost entirely[2] (**fig.2.1**).

- Quantum efficiency - photons falling upon CCD first have to pass through gate layer which can absorb or reflect wavelengths. Quantum efficiency indicates how many photons can be detected. It is typically presented as a graph which indicates quantum efficiency for various wavelengths. Typical quantum efficiency of CCD is in range of 25% to 95%.

Interference for star tracker tests on ground can be numerous, in this part the main focus is:

- *Atmosphere* - If a star tracker camera is pointed off zenith and especially to the horizon, there will be atmospheric refraction. Atmosphere causes light to bend and as a result the stars on the image will experience additional distortion. This is most

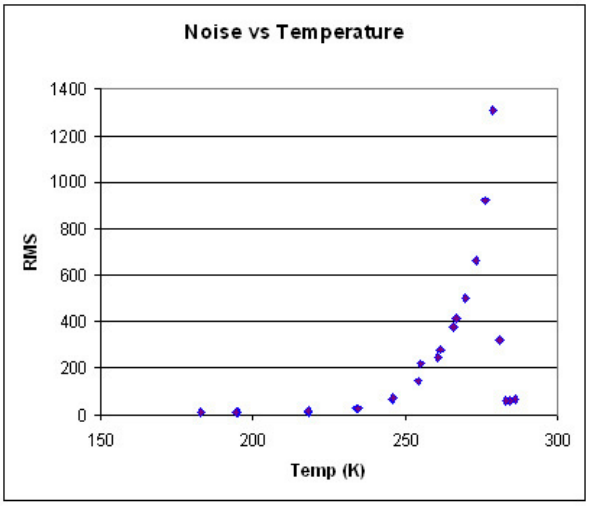


Figure 2.1: Cooling of CCD effects

likely to happen when testing camera by rotating it and pointing it at different sky scenes. If camera is tilted too much then one side can be under higher atmospheric refraction than another. This however can be mitigated by calculating refraction and taking it into account during the testing. To find atmospheric refraction this formula can be used[3]:

$$R = cot\left(h_a + \frac{7.31}{h_a + 4.4}\right) \tag{2.1}$$

here R is the refraction given in arc minutes and h_a is apparent altitude of astronomical body. This formula assumes temperature of $T = 283.15K$ and atmospheric pressure of $P = 101kPa$.

- *Light pollution* - when performing star tracker tests on ground, they are always expected to be carried out in remote areas where there is no light pollution. But in case this is not an option, then it has to be taken into account. Light pollution will introduce unwanted photons into pixels thus not allowing to capture full extent of star.

- *Temperature* - A heat plume can cause air ripple. This becomes problematic if this occurs and a part of field of view is inside it. The heat ripple will act as lens and vignetting might occur. In addition to this CCD sensors performs the best at lower temperatures.

- *Electromagnetic Disturbance* - When a camera is operated in areas with strong electromagnetic radiation, waves are visible in the image (this problem has been reported by photographers taking pictures from the roof of the Empire State Building, where antenna emissions are very strong and caused them to use film cameras).

Star Tracker cameras should not have this problem since they are intended to work in space where strong EM radiation can occur.

2.2 Lenses Distortions and Aberrations

Ideally, in a optical system, a monochromatic spot light source at infinity will form a spot image on a focal plane perpendicular to the optical axis, regardless of its position in the plane. A deviation from this ideal condition is said to be an *aberration*. In particular, some of these errors are termed as *errors of focus* (failure to produce a point image of point objects in the same focal plane) and others are *errors of magnification* (failure to produce the same image scale over the image area). Avoiding aberrations is the primary challenge in the design of a lens and such errors strongly affect the quality of an image produced by an optical system and this is a matter of interest in an application such as star trackers where this can influence matching algorithms between image acquired by the camera and available star fields images. Moreover, in the case of a star tracker the subjects are stars so far away that can be considered white point-light sources on a dark background and this result in a more pronounced visual effect compared to other camera applications.

In general, the effects of aberrations are more dramatic at the edges of an image, causing a distorted shape of stars. Wide angle lenses, especially at higher apertures (smaller f-number), are most affected as it will be stressed below.

It is assumed here, unless otherwise stated, that every optical element and the camera focal plane share the same optical axis (*perfect collimation*).

There are different types of aberrations:

- Spherical.
- Comatic (or simply called Coma).
- Astigmatism.
- Field curvature.
- Distortion.

Colour images also have *chromatic aberrations*, but this case is not considered here.

Sperical aberration is visually manifested as a uniform concentric spread of a point to a gradually blurred roughly circular area with lower contrast towards the edges.

The effect is caused by spherical lens elements focusing light rays parallel to the optical axis, at different aperture diameters, at different distances. This effect is uniform over the entire image frame, even close to the center of the image and in this sense this is different than other aberrations. For stars this effect is not detrimental unless it is not too dramatic because stars will still appear as spots.

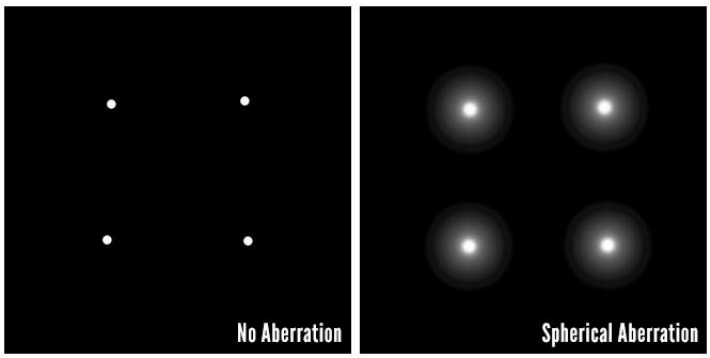


Figure 2.2: Spherical aberration [4].

Coma causes a point image to look like a wedge-shaped tail extended towards the radial direction from the optical axis (fig. Figure 2.3). It can be *external* if the the tail extends away from the optical axis and *internal* otherwise.
Coma is caused by non-uniform magnification of parallel, *abaxial* (not parallel to the optical axis) light rays passing through the centre of the lens and at the edges of the aperture. Light rays incident at higher aperture diameters are more distorted and so the effect is more visible at the edges of an image and higher apertures will give a worse effects.

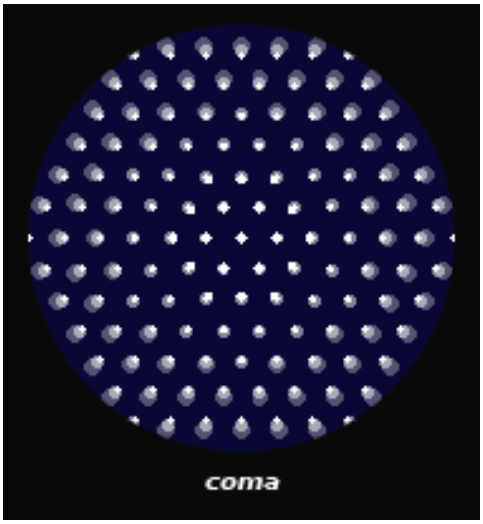


Figure 2.3: Comatic aberration [5].

The non-uniform magnification is caused by the difference in the refractive power at the sides of a lens due to different incidence angles. If an optical system is not perfectly collimated, the effect can be present also at the optical axis.

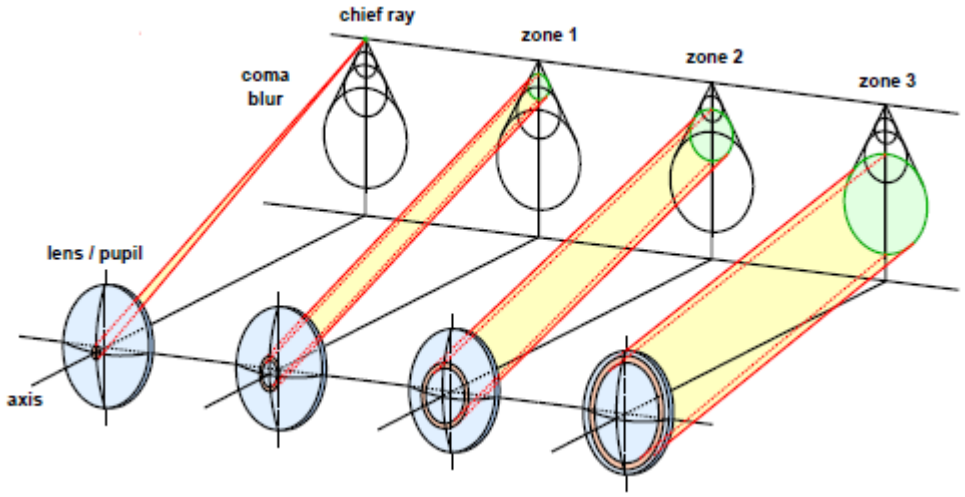


Figure 2.4: Distortion of point light source due to coma.

Astigmatism is both a focus and a magnification error. It is a form of aberration that is always present in off-axis point sources resulting in abaxial light rays. When these rays enter the lenses, they are subjected to different refracting power from the lens, as it happens in the case of Coma and this is the magnification error component resulting in a distortion of a point image into a stretched, thin ellipse resembling a line. This effect occurs along two different directions relative to the optical axis: *sagittal* and *tangential*. Moreover, there is a difference in the focal distances of light entering the lens in the sagittal and in the tangential planes. Either focal distance can be in front or behind the focal distance of *paraxial* rays (entering the lens parallel to the optical axis) resulting in the focus error component. Visually, the effect on an image is that point sources are distorted in lines that stretch radially from the optical axis and lines that are seemingly perpendicular to the radial direction to the optical axis.

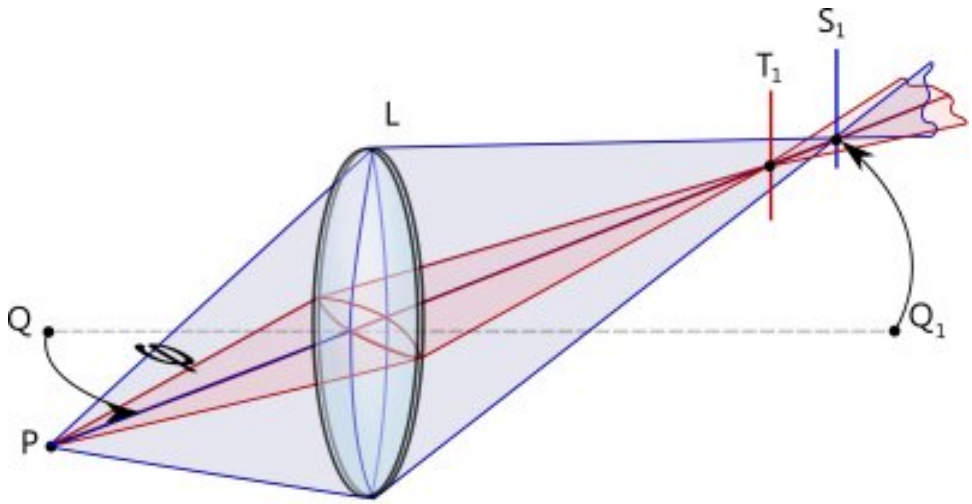


Figure 2.5: Astigmatism.

Astigmatism is very complex in the sense that either component can be more distorted than the other. The effect is always present for abaxial rays and is more evident on the edges of the image and particularly at the corners. On the optical axis there is no effect unless the optical system is not perfectly collimated. Astigmatism is the most difficult error to correct and even the most expensive lenses exhibit a certain degree of this type of aberration.

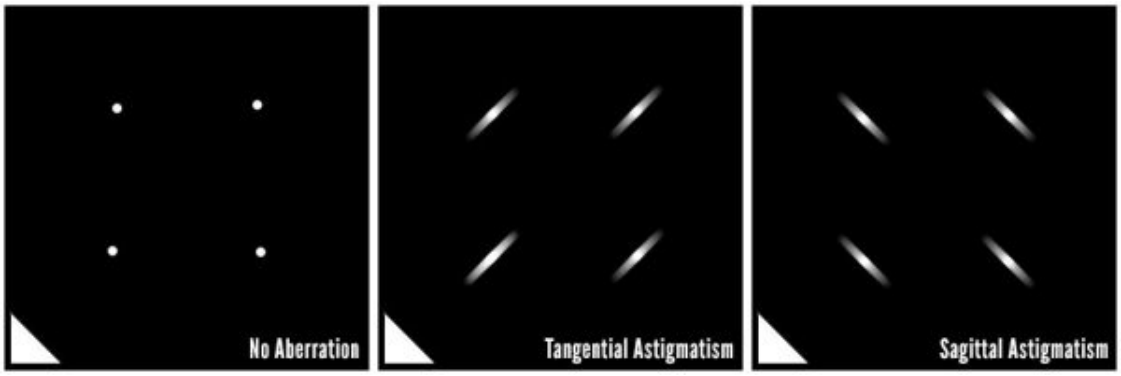


Figure 2.6: Effect of astigmtism (the white triangle represents the direction towards the optical axis) [4].

The combination of sagittal astigmatism and coma lead to an effect called *sagittal coma* that gives stars at the edges a typical shape.

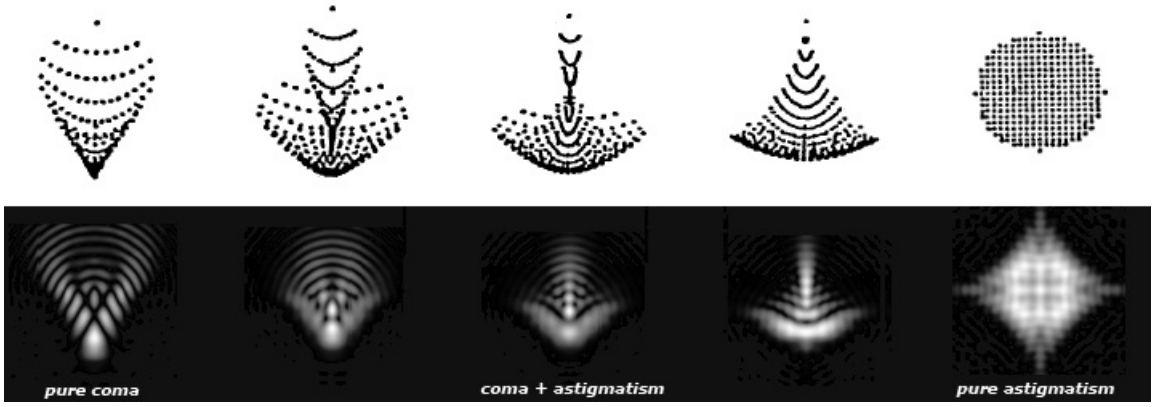


Figure 2.7: Coma and astigmatism together [5].

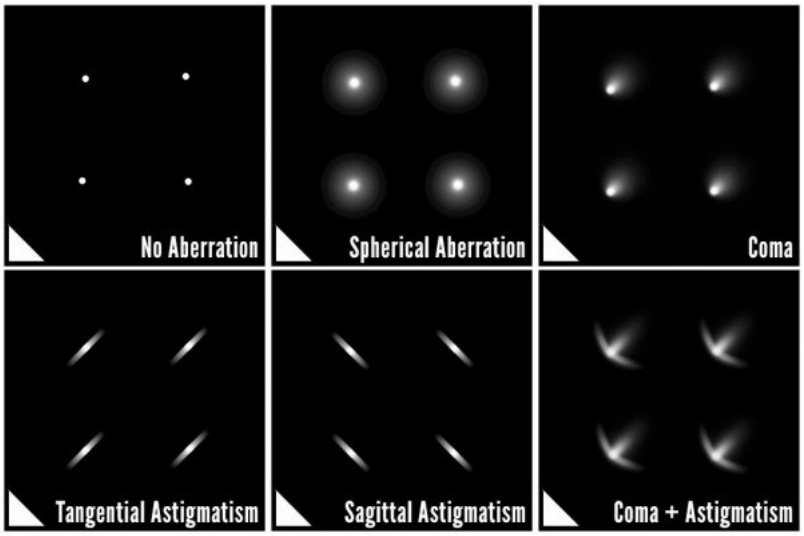


Figure 2.8: Summary of aberrations seen so far [4].

Finally and briefly, *distortion* is a projection of an image with a variable magnification towards the edges: higher magnification results in straight lines at the edges to be curved toward the center of the image (*positive*, or *rectilinear*, or *pincushion* distortion), lower magnification on the contrary determines outward-curved surfaces (*negative*, or *barrel* distortion). Also a combination of the two is possible (*moustache* distortion). The effect in star tracker images is a change in the position of stars.

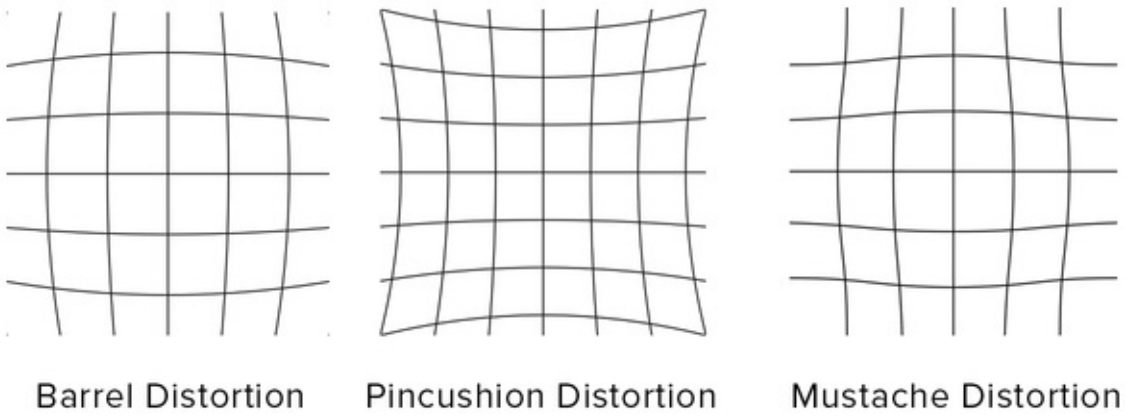


Figure 2.9: Distortions.

Field curvature consists of a focusing plane that is not flat. This causes changes in focus across an image since camera sensors are flat instead. The effect can be present either at the edge or at the central area of the image where the scene will be out of focus.

A formal theoretical discussion of the various types of aberrations, models, and their dependence on aperture and field of view is very complex and can be found in many texts of which **[6, 7, 8, 9]** are some examples.

2.3 Digital Images

In the following, a few relevant aspects of digital images are described. A thorough description of digital images and image formation is out of the scope of this report and will not be discussed here but it is widely available in literature. Here, we briefly describe what is relevant for the work that has been carried out.

2.3.1 Basics

A digital image is the result of the projection of the light coming from a scene onto an electronic sensor, through a system of optical elements that form the lens. A sensor is a grid of photosensitive elements that collect light and encode photon counts into a digital number describing an intensity level, for each element in the grid. The digital image is then a grid of intensity levels. With reference to a gray-scale, 8-bits image, these levels can assume values between 0 and 255.

The spacial distribution of pixel values, i.e. how intensity level vary moving from a pixel to another, is sometimes denoted denoted *spacial domain* whereas the distribution of the values that may be present in an image is sometimes denoted *range, or colour, domain*.

Digital images always come together with noise due to the electronic nature of the image formation process that can be affected by measuring and recording errors, and interference from external sources, as described above. This causes a random variation of intensity levels over the entire image that is superimposed to the useful signal levels. The amplitude of the disturbance may vary considerably. For electronic sensors, noise can be generally approximated as Gaussian-distributed. There are other forms of noise, e.g. *salt-pepper noise* (sparse pixels that assume the extreme values in the range and so appear as either pure black or pure white), *impulse noise* (like salt-pepper, but only involves white pixels).

2.3.2 Raw Images

A raw image consists of uncompressed, minimally processed data acquired by the sensor of a camera system.

Raw images are used because they retain all the information from the sensor and allow to exploit the best quality achievable with a specific system. This is an important feature because if the camera produced an image in a format such as JPG or TIFF, any compression would be performed by the camera itself but cameras do not normally do a good job at compressing images compared to a computer and even if they did, it would

be an additional computation requirement to the system. Another important feature is that raw images can record a higher number of intensity levels and so achieve better details.

The drawbacks of using raw data are the lack of a standardized format, every manufacturer uses one or more different formats (e.g. *.CR2, .CR3, or CRW* for *Canon*, *.NEF, or .NWR* for *Nikon*, *.ARW, .SR2, .SRW* for *Sony*). Moreover, there is no commercial software tool that is able to directly use raw images so one should use either proprietary or professional tools, or work directly with binary files.

2.3.3 Histograms

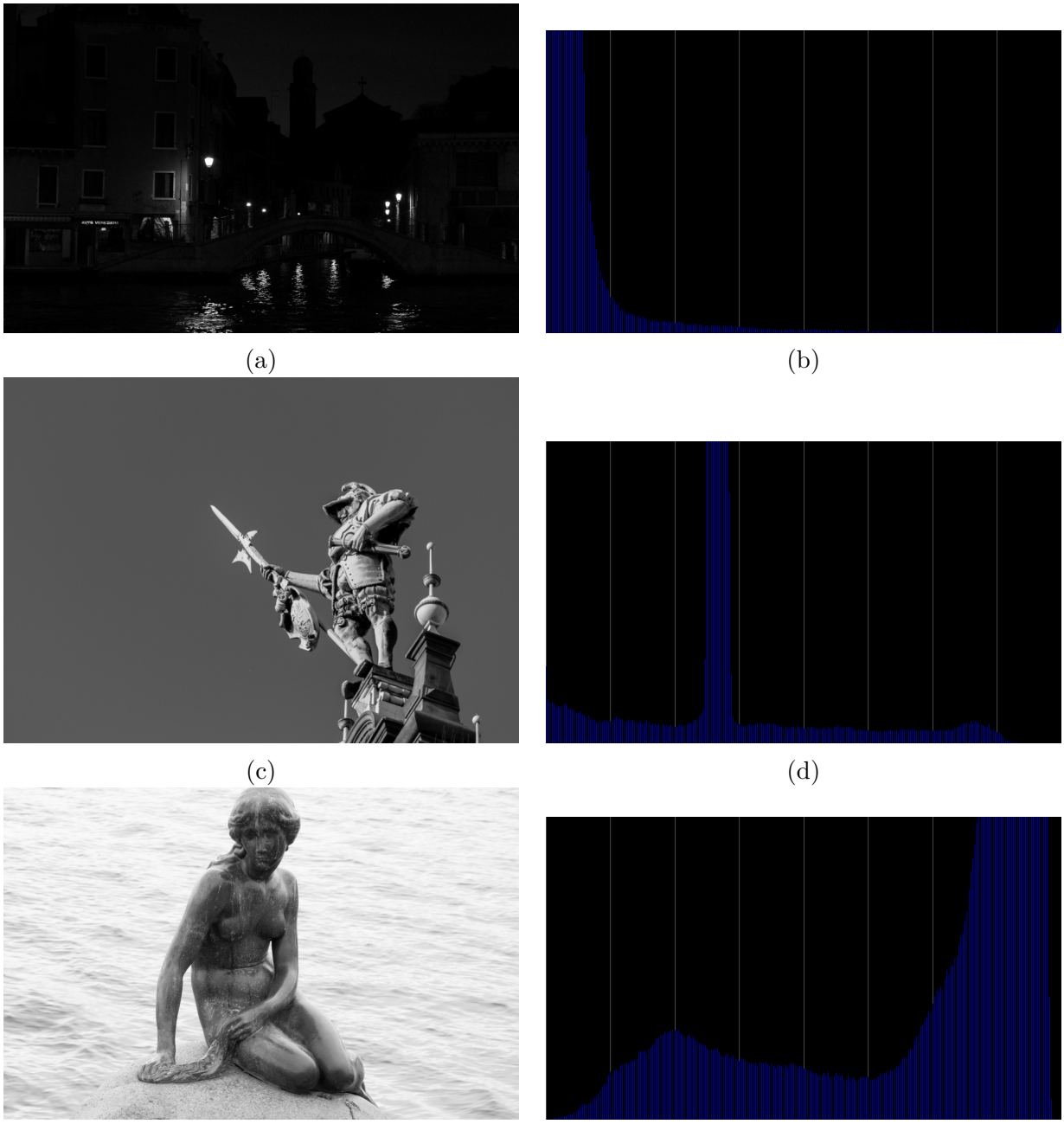
A histogram is a graphical representation of the distribution of brightness levels. It illustrates for each level the number of pixels in the image that assume that specific level. The left part of the histogram representing the darker tones is known as *shadows* whereas the right end is known as *highlights* (i.e. the brighter tones), the central part represents the *mid-tones*. When some pixels exceed the limits of the histogram, those pixels will be saturated either to pure black or to pure white and it is said that there is *clipping* to the blacks or to the whites respectively and it represents a loss of information.

A histograms is a very important tool in image processing and image analysis. It allows one to assess the *exposure* (amount of light per unit area) of an image as well as the *contrast* (difference in brightness that makes different parts of the image distinguishable), the possible presence of clipping, and it is fundamental for automatic processing. Even a quick look at an histogram allow a person to qualitatively assess important aspects of a picture.

In **fig. 2.10** three examples of three different scenes are illustrated. **Fig. 2.10a** and **fig. 2.10b** represent an underexposed scene: the overall image is very dark and the histogram is concentrated in the shadow part and there are no highlights. In **fig. 2.10c** and **fig. 2.10d** mid-tones dominates and it is visible that most of the pixels are located somewhere close to the middle of the histogram since the image is predominantly gray. Finally, **fig. 2.10e** and **fig. 2.10f** depicts an overexposed scene where the histogram is compressed to the right end and there is comparatively almost no shadow.

Also, in **fig. 2.10**, it can be seen that (a) has a low contrast since the histogram is concentrated in a narrow region compared to (c) and (e) for which the histogram is more stretched over the range and in fact the scene in (a) is not clearly visible whereas in the other two images can be easily distinguished from the background.

Finally, in **fig. 2.10** all three images exhibit an histogram that is mostly concentrated around a single region. In this case the histogram is said to be *unimodal*.



(a) (b)

(c) (d)

(e) (f)

Figure 2.10: Three different scenes and relative histograms: Underexposed scene (a and b) (*Venice, Italy*); Mid-tones dominating scene (c and d) (*Hamburg, Germany*); Overexposed scene (e and f) (*'Den lille havfrue', Copenhagen, Denmark*). Photos taken by the authors.

2.4 Image processing

In this section a description of the processing steps taken in this project is given.

Before starting, it is useful to give some definitions. We talk about *image analysis* when we make a transformation from an image to some form of information that is not an image, e.g. a decision, or results of measurements of some kind. *Image processing* instead is a transformation from an image to another image where relevant features have been enhanced. In the perspective of machine vision image processing aims at making subsequent analysis more simple and reliable. Within image processing we distinguish between *point operations* and *local operations*. In the first case a single pixel is replaced basing on the value of the single correspondent pixel in the input image. In the second case, the resulting pixel value depends on the correspondent input pixel and on its neighbouring pixels. We also distinguish between *linear* operations, in which the action of the operation is the same over the entire image irrespective of the local properties and *non-linear* operation where the action is adapted to the position in the input image.

2.4.1 Image Filtering

Image filtering is one of the most fundamental low-level processing operation whose purpose is to reduce noise in the image. The idea is that signal intensity levels for pixels that are close will be more correlated than noise levels, therefore by mixing the values of a certain number of neighbouring pixels, variations in intensity due to noise will be dumped. In this regard, filtering is a local operation.

A filtering operation is described by a discrete convolution between an image and a *mask*, or *kernel*. If f is the source image, h is the kernel, the resulting image g is given by eq. 2.2 [10].

$$g(i,j) = f * h = \sum_m \sum_n f(i-m, j-n)h(m,n) \tag{2.2}$$

The form of eq. 2.2 is found in different image analysis books and requires the kernel to be horizontally flipped by 180°. This equation is a formal mathematical expression that can result obscure. In practice, a filter consists almost always of a square kernel, say n x n, with n a odd number. The kernel is scanned over the source image in every possible position, that means that it is placed so that the central element corresponds to every and each pixel in the image. For a given position (i, j), the pixel g(i, j) is computed as a weighted sum over a square n x n area around f(i, j) with the elements of the kernel being the weights. Therefore, if the origin of the kernel is the central element, the resulting pixel g(i, j) is given by

$$g_{ij} = \sum_p \sum_q f(i+p, j+q)h(p,q) \tag{2.3}$$

With p and q varying from $-\frac{n-1}{2}$ to $\frac{n-1}{2}$.

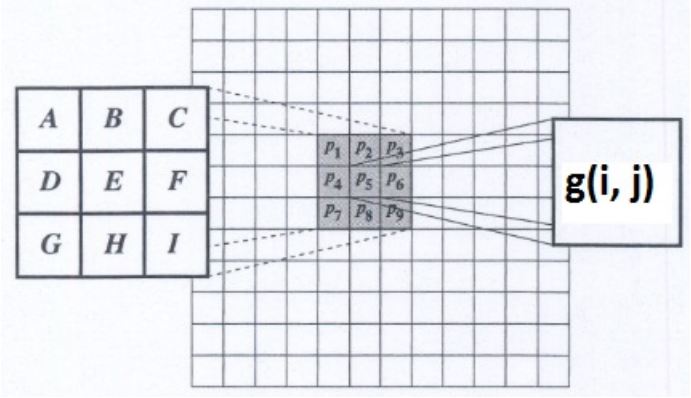


Figure 2.11: Filtering.

In **fig. 2.11** this is illustrated and with reference to it eq. 2.4 becomes

$$g(i,j) = AP_1 + BP_2 + CP_3 + DP_4 + EP_5 + FP_6 + GP_7 + HP_8 + IP_9 \tag{2.4}$$

The most simple kernel is the *mean filter* which is a n x n kernel where each element is equal to $\frac{1}{n^2}$. With this kernel, filtering is a simple weighted averaging and it is known as *blurring* or *smoothing*. It is also said that his type of filtering is a *low-pass* filtering. Since near pixels noise is less correlated than signal values, noise is averaged out. It is fairly effective in most cases but as a drawback the resulting image will be smeared because the filter doesn't discriminate between large variations due to noise and those due to inherent characteristics of the image, so sharp variations become more gradual and the ability to localize changes in intensity in later processing is sacrificed. The larger n, the more effective is the suppression of noise but the more severe will be the blurring of sharp details.

A better blurring is achieved when a filter has a main lobe (a higher value, normally as the central element) and lower values symmetrically distributed around it.

An important class of linear smoothing filters are the *Gaussian filters* (the name does not have anything to do with the fact that noise is often modelled as Gaussian, it is merely a coincidence). In a Gaussian filter, weights are chosen accordingly to a discrete approximation of a Gaussian function. For image filtering, the two-dimensional zero-mean Gaussian function is [10]

$$h(i,j) = e^{-\frac{i^2+j^2}{2\sigma^2}} \tag{2.5}$$

A Gaussian filter is a low-pass filter with some interesting properties. They are *rotationally symmetric*, they perform the same amount of filtering in every direction so avoiding bias in the image (a smoothing oriented in some direction if the orientation of edges is known, which is rarely the case); they introduce less distortion to the edges. The degree of smoothing is controlled by a parameter, σ. They can be implemented efficiently

(a 2D Gaussian filtering can be done by using a 1D filter and applying this twice with a 90° shift, this results in a linear growth of computation instead of quadratic). An arbitrarily big filter can be made efficiently by convolving the same image in succession (convolving with a n x n filter first and with a m x m filter after is the same as convolving with a (n + m - 1) x (n + m - 1) filter) [10].

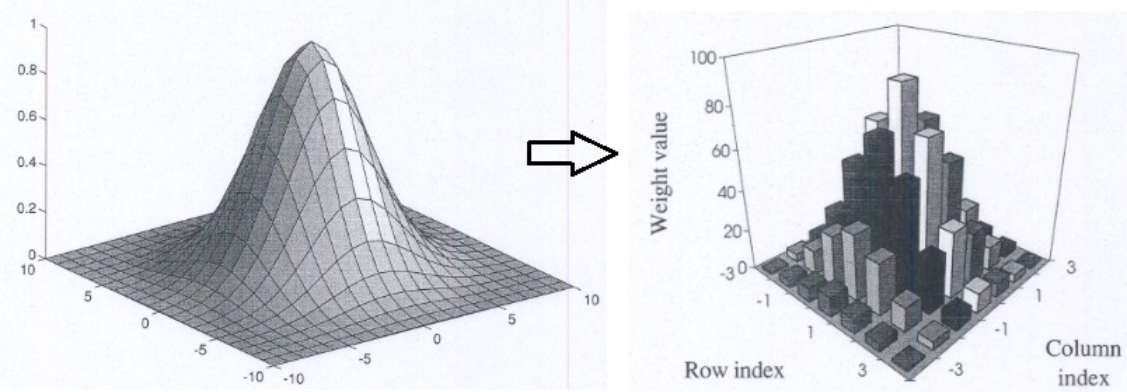


Figure 2.12: 2D Gaussian function and its discrete approximation [10].

As mentioned, the parameter σ controls how "large" the filter is. The higher its value is, the higher the degree of filtering (**fig. 2.13**) and the closer to the continuous is the discrete approximation but a higher value also mean more computation [10].

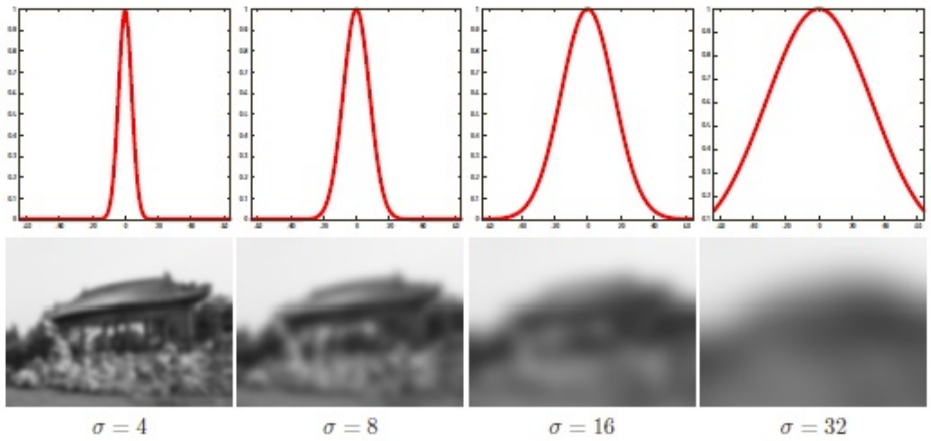


Figure 2.13: Effect of σ on filtering [11].

The smoothing filters described so far are linear.

Gaussian smoothing is still not optimum from the point of view of edges preservation. An good alternative is *bilateral filtering* [12], which is a non-linear technique that consists in the combination of filtering in spatial domain (geometrical closeness) and in range (photometrical closeness). Averaging in range alone only distorts the colour map and averages pixels that are similar in colour but may be far from each other so it is not

useful. By combining it with a traditional spacial filtering (**fig. 2.14**) however allows to achieve an averaging process with weights that decay with dissimilarity so a pixel is replaced in the resulting image by the average of pixel values that are close and similar in value. This situation is of course a local characteristic of an image, thus the technique is non-linear.

In a uniform area the filter behaves exactly like a spatial filter but considering the situation of a sharp edge, there will be a step in the pixel values as depicted in **fig. 2.14a**.

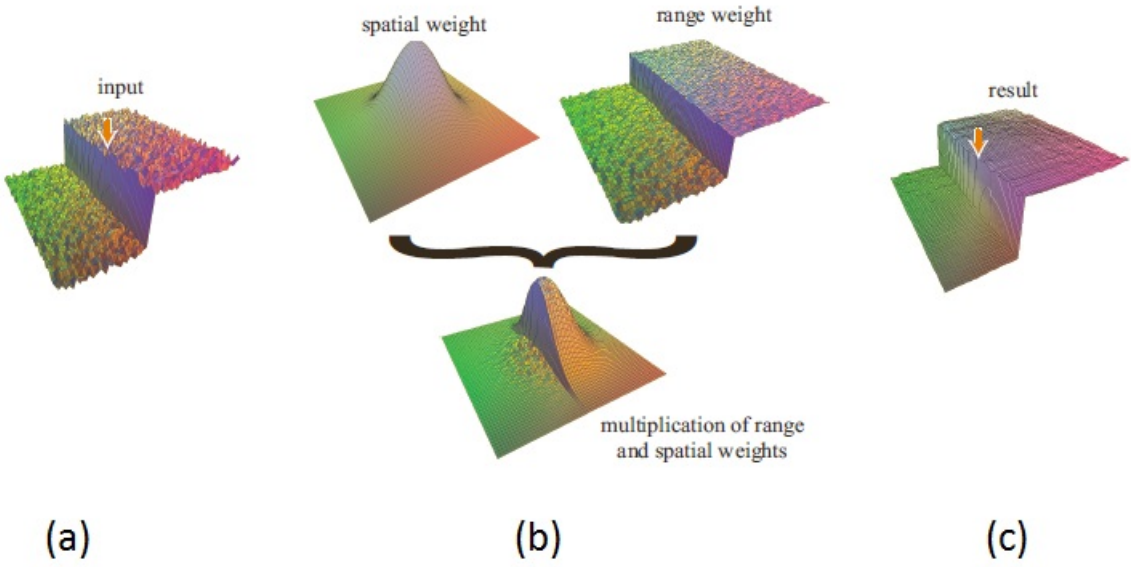


Figure 2.14: Bilateral filtering: a) visual representation of an edge in a image (a step of intensity values); b) the kernel of the filter; c) the result of smoothing. [11]

A pixel on one side of the step will be close in range to neighbouring pixels on the same side so they will be averaged together whereas pixels on the other side, being far in range will be suppressed totally by the weighting (**fig. 2.14b**). The situation will repeat analogously for the other side of the edge and the result will be a correct smoothing with edge preservation (**fig. 2.14c**). There will be no loss of overall shape, but textures are removed (**fig. 2.15**).

An example of bilateral filter is the *shift-invariant Gaussian Filter*. For this filter both closeness and similarity functions (the functions determining the weights in the two domains) are Gaussian. The parameters to control the degree of filtering in the two domains are σ_s (space) and σ_r (range) (**fig. 2.16**).

The mathematics describing this behaviour can be read in [12] and [11].

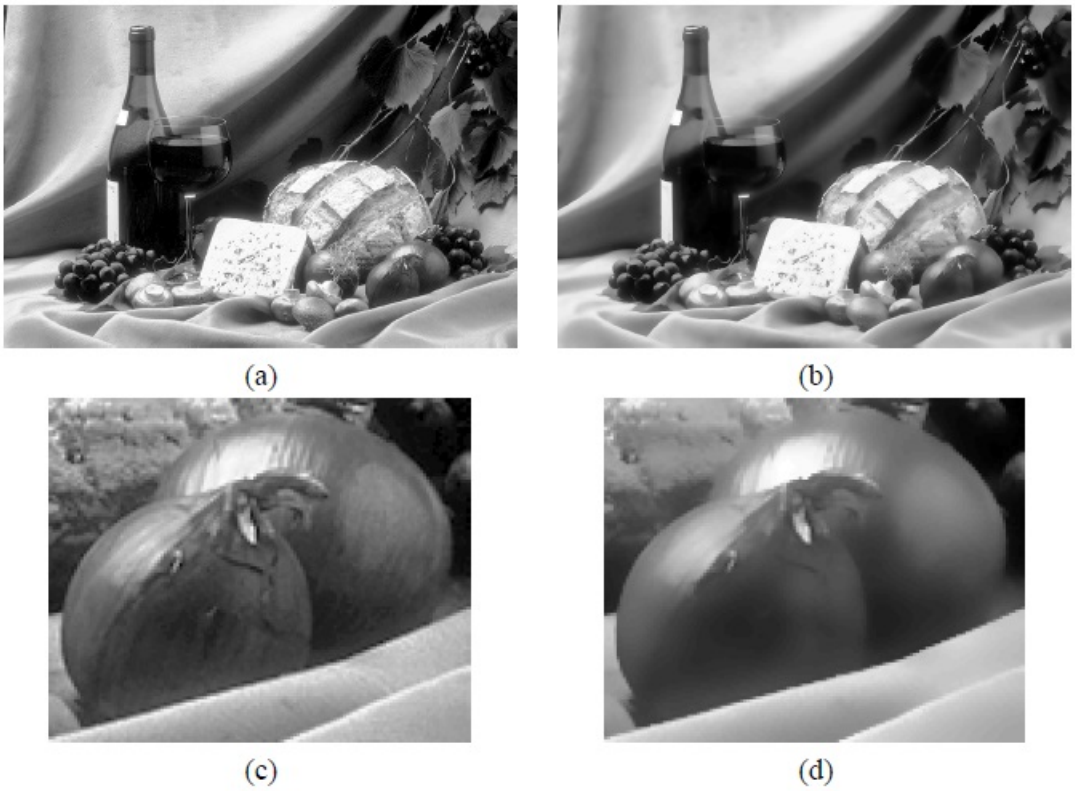


Figure 2.15: Bilateral filter effect: a) input image; b) output image; c) a detail of the input image; d) same detail after filtering. [12].

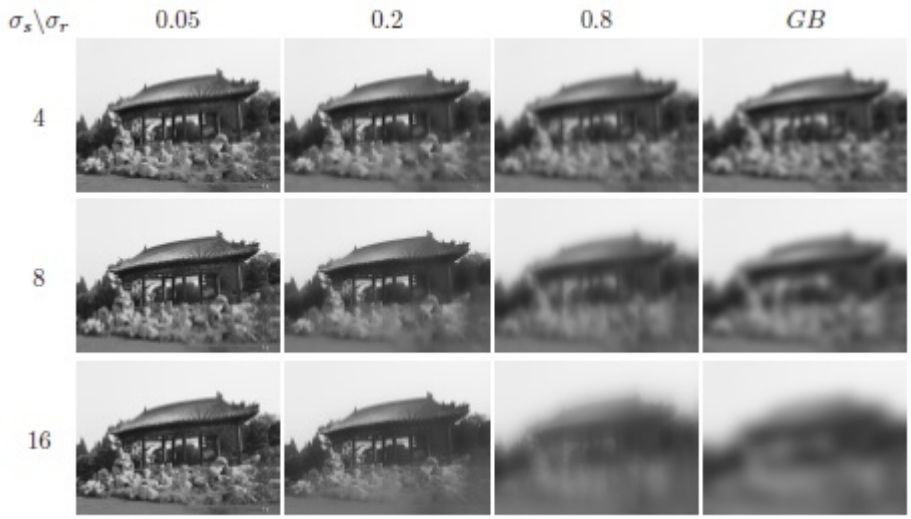


Figure 2.16: Results of shift-invariant Gaussian filter compared to a classic Gaussian blurring (rightmost column) [11].

The meaning of σ_s is the same as a normal Gaussian filter. Regarding σ_r, pixels with

values much closer than this parameter are mixed together. If σ_r is much higher than the overall range of values in the image and σ_s assumes small values, the range component of the filter has no effect. If both values are high enough, the spacial component has no effect and the result is just a remapping and a compression of the histogram. This causes the image to be sharper than the case in which σ_r is the same but σ_s is lower, which may seem a paradox but it can be explained by the fact that the exclusive action of the range filter does not affect the details [11] (**fig. 2.17**).

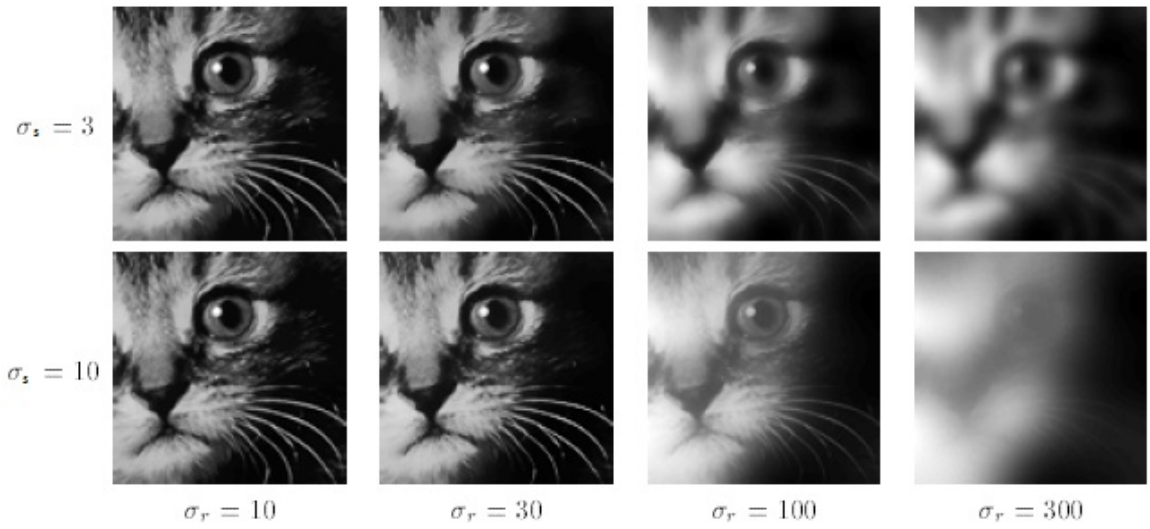


Figure 2.17: Sharpness paradox in bilateral filtering: comparison between $\sigma_s = 3$ and $\sigma_s = 10$ with $\sigma_r = 100$ [11].

Some things to note are: if an image is scaled, σ_s must be changed accordingly; σ_r must be reset in case of amplification or attenuation instead; the filter is shift invariant, meaning that if $f(i,j) * h(i,j) = g(i,j)$, then $[f(i,j) + a] * h(i,j) = g(i,j) + a$.

2.4.2 Morphological Transformations

Morphology transformations are a class of local, non-linear operations. They operate by scanning a kernel, that in this context is called *structuring element*, on an input image, similarly to filtering. A test against the structuring element is then performed on the underlying region and the pixel in the image corresponding to an *anchor point* (usually the centre of the structuring element, but in general it can even be outside of it) is replaced with a new value dependent on the specific operation and of course on the result of the test. A structuring element is a small *binary image*, i.e. an image where pixels can assume only two possible values, that act as a mask. Different patterns described in the structuring element allow one to enhance or suppress certain shapes in the input image. Note that using larger structuring elements is equivalent to performing successive iteration with one small element.

The most important thing in these operations is the position of the pixel rather than their values, so these operations are usually carried out on binary images. Nonetheless, these operations can be extended to gray-scale images.

For binary images, the aim of such operations is to improve the shape of a segment in the image by removing imperfections caused by textures and noise in the original image used to form the binary image (this operation will be described later). In gray-scale images they are used to reduce noise (even though filtering gives better results), brighten or darken areas of the image to improve the isolation of segments, i.e. regions of similar values in the image.

For the reasons described, morphological transformations are very important in supporting other operations such as segmentation, pattern recognition, objects and edge detection.

The formal theory of morphological transformation is vast, there is a great variety of operations and applications, and several algorithms are studied. Here only a brief description of a few operations employed in this work is given.

Two basic morphological transformations are *erosion* and *dilation*. The result of erosion on a gray scale image is that of a local minimum: the pixel under the anchor is replaced by the minimum value among the pixels covered by the structuring element pattern. Dilation is the converse operation that computes the local maximum. The effect of erosion is to isolate and shrink bright regions (making the overall image darker) whereas dilation will make bright regions to grow (**figg. 2.18** and **2.19**). Dilation is often used to reduce speckle noise [13]. Dilation is usually performed to restore components that may have been broken by noise, shadows, and so on.

Figure 2.18: Erosion (3 iterations, 3x3 square structuring element) [13].

Figure 2.19: Dilation (3 iterations, 3x3 square structuring element) [13].

For binary images, erosion replace a pixel with 0 if the structuring element pattern *"hits"* (if at least one pixel covered by the pattern is white) a segment or set it to white if the pattern *"fits"* (all pixels correspondent to the structuring element are white) the segment. Dilation is complementary. Therefore the effect of erosion will be to remove a layer of pixels from any boundary, whereas dilation will create a layer of pixels. The effect on the binary image is basically the same as in a gray-scale image, but visually more evident (**fig. 2.20**).

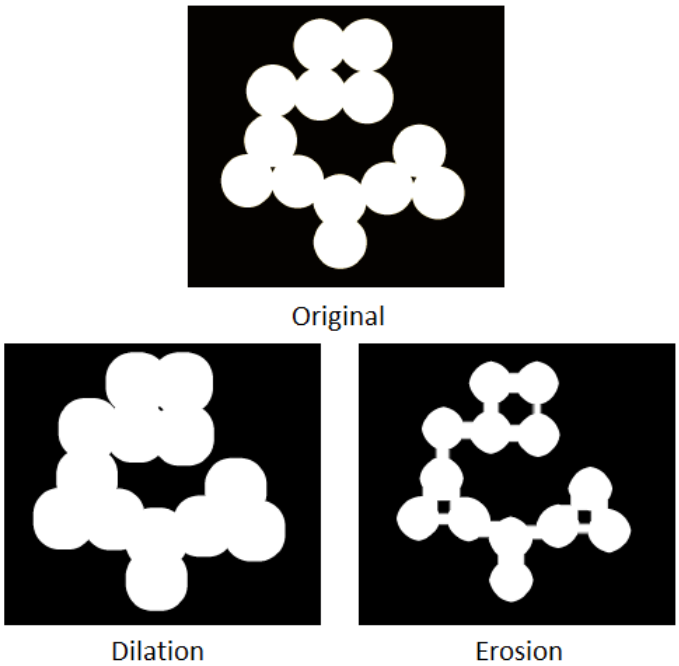


Figure 2.20: Erosion and dilation on a binary image (3 iterations, 3x3 square structuring element).

In general, dilation will tend to smooth concavities whereas erosion will smooth protrusions but the degree of this effect depends on the pattern of the structuring element.

In general the result will depend on the pattern of the structuring element as shown in **fig. 2.21**.

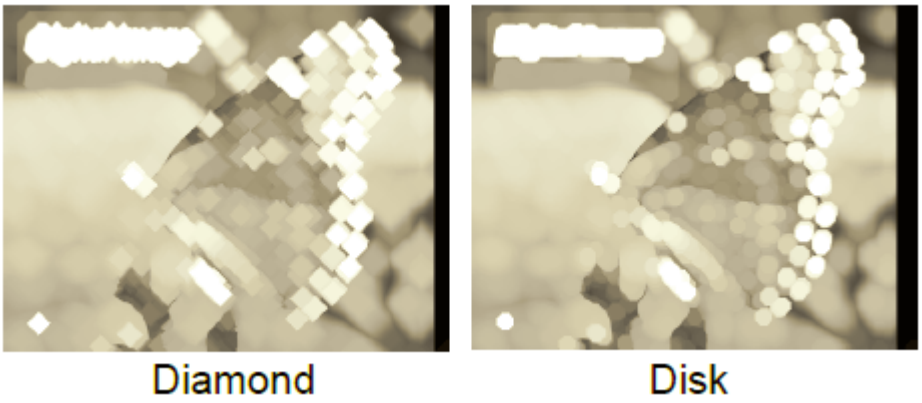


Figure 2.21: Dilation performed with two different structuring elements.

Starting from the basic morphological transformations, others are obtained.

One of these is called *opening* and it consists of an erosion followed by a dilation with the same structuring element.

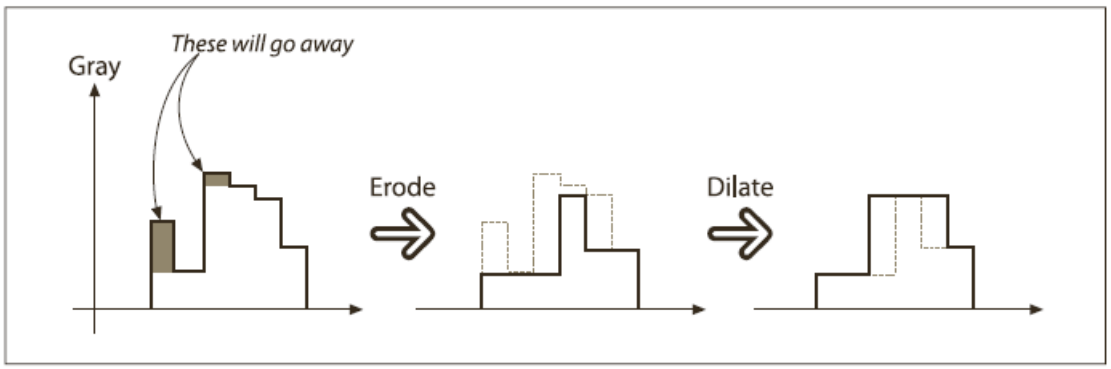


Figure 2.22: Opening operation [13].

The most noticeable effect of an opening operation is the separation of segments near each other by eliminating lone outliers with an higher value than their neighbours. In practice, thin "bridges" of pixel connecting segments will be broken by the erosion and then the dilation will restore the size of the segments that have been eroded as well in the previous process. Opening is often used in binary images to count regions.

Note that the operation is *idempotent*, i.e. after a proper separation is achieved an opening transformation has no longer effect.

2.4.3 Thresholding

Thresholding is a point operation through which a *boolean*, or binary image is produced from a gray-scale input image. In the previous section it has been mentioned what binary images are. Now they will be fully described.

A binary image is an image where pixels assume only two values, normally correspondent to black and white. To obtain this result, a threshold, T, is selected from within the range of possible values (e.g. 0-255, for 8 bits) and the resulting image g is produced from a 8-bits, gray-scale image f as:

$$g(i,j) = \begin{cases} 0, & \text{if } f(i,j) < T \\ 255, & \text{otherwise} \end{cases} \tag{2.6}$$

In the most simple case, the threshold is set to a constant value. This can be possible if there is a priori knowledge about a scene, otherwise a proper threshold would have to be selected on a case to case basis, maybe interactively, after an observation of the result or of the histogram. The latter is of course a better method because in the former case a binary image is produced every time and then possibly discarded whereas the histogram is drawn only once. However examining the histogram and manually selecting a reasonable threshold requires some experience. Either case is not very practical in the perspective of an autonomous system, or when working with considerable amount of data.

Automatic methods to select an appropriate threshold would be advisable. Many methods have been proposed but none of them is optimal for all possible conditions. The histogram itself strongly influences the efficiency of different methods since all of them are based on some assumptions on its distribution. For example, a popular method known as *Otsu's method* [14] is a method based on the minimization of the ratio between statistical measures and it is efficient and optimal for *binomial histograms*, i.e. histograms that exhibit two peaks around which the values are concentrated. There are methods based on methods to maximize some measure of entropy [15, 16]. An interesting method based exclusively of the shape of the histogram and for histograms that present a single dominant peak in the lower end has shown superior performance in certain analysis tasks such as change detection, segmentation, and change detection [17].

Another possibility is *adaptive thresholding* in which a threshold is determined locally depending on the neighbourhood of the pixel in the input image and applied to the same area. In this case the thresholding operation is no longer a point operation of course. These techniques are thought for cases where illumination conditions may vary across the scene. Again, a variety of algorithms are available [18, 19] but since they focus on the local conditions the final binary image is not always optimal.

2.4.4 Contour Search

The approach used in this project relies on edge detection. In this section techniques for edge detection will be introduced.

2.4.4.1 Edge detection

In image processing edge detection is a widely used technique for detecting objects and features in images. Edge is a sharp change, a step, in data. it can be found by simply checking pixel to its neighbours and comparing their values. It is simple when the edges are very sharp, like on binary image, but if edge is not a step but rather a ramp, detection becomes a lot harder because it becomes difficult to discern where exactly edge began. This is most apparent when finding edges between similar colour or brightens change.

2.4.4.2 Canny edge detection

One of the most popular and used methods for edge detection is **Canny edge detection**. It was created in the 90s and it is still considered one of the best due to couple of reasons. It has methods to deal with edge thinning and linking. And it is able to discern real edges from noise artifacts by employing **double thresholding** and **hysteresis**.

Canny edge detection is a multistep process [20].

- Removal of noise.
- Intensity gradient.
- Non-maximum suppression.
- Double thresholding.
- Hysteresis edge tracking.

The first step is the application of a Gaussian filter to reduce noise, as explained in-depth in a different section of this report. Filtering of image not only reduce noise but also has demerit of smoothing out edges slightly which is a special unwanted result in edge detection. An important point to take in consideration is also the size of the kernel. Kernel that is too large will result in reduced sensitivity to noise.

After image is smoothed it is necessary to find image a intensity gradient. This can be done by using a *Sobel kernel* that emphasizes edges. By applying it in both vertical and horizontal directions we can get first derivative in horizontal direction G_x and in vertical direction G_y. After forming images with sharpened edges it is possible to find edge gradient and direction for each pixel [21].

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.7}$$

$$\Theta = tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{2.8}$$

The following step removes any pixel that is not part of an edge. This is done by comparing pixel with other pixels that are in its neighbourhood. Pixels are usually

checked with an 8-connectivity to ensure that all nearby pixels are taken into consideration. Pixel intensity is checked in its direction of gradient and if it is a maximum value in relation to its neighbours it is considered to be an edge and other pixels are suppressed. This step provides a binary image that is formed of thin edges.

To ensure that all edges are accounted for and that they are true edges a double thresholding is performed. First two threshold values are chosen, this is usually done manually. But since it is not feasible for program that has to process multiple images, an automatic thresholding is desired. One of the approaches used is to apply Otsu's method, which provides a threshold value and then from it a second threshold value is determined, which is usually a half of the maximum value. Threshold values are used to classify edges. Edges are considered strong if they are above the maximum threshold value. This means that edge is true edge and can be used as reference for next step. Edges that are in between maximum value and minimum value are considered to be weak edges. They might be a result of noise or it is an edge that forks out from strong edge. Any edge that is bellow minimum value is discarded since it is most likely a product of noise.

Last step is to evaluate connectivity of *weak edges*. If there is at least one connection to a *strong edge*, it can be considered to be real edge and should be displayed. If weak edge is separate from any strong edges, then it will be discarded.

2.4.5 Image Moments and Centroiding

Image analysis has many uses for image moments. They help to recognize patterns, estimate noise sensitivity, and also to evaluate object in an image. For this project only zero'th and first order moments were necessary therefore only the method for finding them and their usage will be outlined.

2.4.5.1 Finding moments

A general, two-variable function to represent moments can be given as[22]:

$$M_{ij} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^i y^i f(x,y) dx dy \tag{2.9}$$

For image processing this can be expressed as:

$$M_{ij} = \sum_{x=1}^{W} \sum_{y=1}^{H} x^i y^i f(x,y) \tag{2.10}$$

This is possible since images have fixed, known dimensions and we simply consider $f(x,y)$ to be our image with specific height and width [23]. It is possible to further reduce this formula to receive zero'th order moment which indicates the mass of our image. If image is binary then it will give area of pixels that are one.

$$M_{00} = \sum \sum f(x,y) \tag{2.11}$$

If we know the total area of the object, it is possible to find the centre of mass simply by calculating M_{10} and M_{01} and getting mean values of $\bar{x}$ and $\bar{y}$.

$$(\bar{x}, \bar{y}) = \left(\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}}\right) \tag{2.12}$$

Equation 2.12 gives us centroid of an object. The ratios in the equation are called normalized first order central moments.

2.4.6 Interpolation

Image interpolation (also known as *image scaling* or *image resampling*) is a technique used in resizing images. The term includes technique for enlarging or shrinking an image. In this context enlarging is more interesting.

The amount of algorithms is massive but in every case one must keep in mind that it is not possible to find more information than already exists and the final quality of the image will always be worse, each algorithm introduces artifacts such as aliasing, moiré, blurring, ringing that may cause clipping and so forth.

The most popular techniques are: *nearest neighbour*, *bilinear*, *cubic* or *bicubic*. Among more advanced algorithms *Lanczos* is worth mentioning. More advanced methods based on auto-regression, fuzzy scaling and artificial neural network produce even better results but they are computationally very complex.

Interpolation is in essence a convolution operation where many elementary computation, e.g. multiplications, additions are performed. Hence, to determine if a method is suited or not, it is necessary to establish a trade-off between computation complexity and quality of the results.

Nearest-neighbour and bilinear methods techniques are computationally quite simple (and fast) but results are very poor. Cubic techniques (cubic is actually a class of different techniques) are usually the ideal solution considering results and complexity, produce noticeably sharper images and they are a standard in editing programs but they create a slight halo effect at the edges. Lanczos algorithm produces superior results in terms of detail preservation and aliasing, but the computational cost is very high (it uses kernels that are sinc window functions scaled and translated that are summed together to evaluate a single pixel causing a lot of computation, not to mention that the sinc itself requires a large number of operations to compute a Taylor series approximation).

2.5 Principal Component Analysis

2.5.1 What is PCA

Principal component analysis (PCA) is a statistical technique. When used in image processing, it can help to extract various features from the data, most often it is used for compressions, dimension reduction, and decorrelation [24]. It also can be used to find

object orientation in image[25]. PCA can be used to identify patterns in data. This is helpful when there is the need to compare data that is of a higher dimensions. By finding patters in data we can see how it is related to other dimensions.

2.5.2 Performing PCA

In order to perform PCA it is necessary to format data so that it takes up two rows.

$$D = \begin{bmatrix} x_1 & x_2 & ... & x_n \\ y_1 & y_2 & ... & y_n \end{bmatrix} \tag{2.13}$$

Matrix D has 2 elements that each have x_n and y_n values. These values can represent pixel coordinates of the object that is being evaluated.

2.5.3 Mean value

To minimize square error of approximating data, it is important to subtract the mean value, μ, from each dimension being evaluated.

Mean values can be simply found by:

$$\mu_x = \frac{1}{n}\sum_{n=1}^{n} x_i \tag{2.14}$$

$$\mu_y = \frac{1}{n}\sum_{n=1}^{n} y_i \tag{2.15}$$

After finding mean values, they are simply subtracted from each value of the dataset to form a new matrix:

$$\boldsymbol{M} = \begin{bmatrix} (x_1 - \mu_x) & (x_2 - \mu_x) & ... & (x_n - \mu_x) \\ (y_1 - \mu_y) & (y_2 - \mu_y) & ... & (y_n - \mu_y) \end{bmatrix} \tag{2.16}$$

2.5.4 Covariance matrix

Simple variance only is useful when working with one dimensional data. If there is need to see how multidimensional data varies in respect to each other, we need to find their covariance. To find covariance two dimensions are used. In our case we got x and y thus we need to find covariance between them. Calculating covariance between same dimension would simply give variance of it.

Covariance is found by using the formula:

$$cov(x, y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)} \tag{2.17}$$

Here x and y are our independent variables, n denotes number of samples in data, $\bar{x}$ and $\bar{y}$ is mean value of each variable.

Covariance value sign lets us determine how data is related. If sign is positive then both dimensions will increase together. If covariance is 0 then this shows us that each dimensions is independent. When covariance value sign is negative then one dimensions will increase while the other one will decrease.

Covariance matrix can be defined with a formula:

$$\boldsymbol{C} = \frac{1}{n}\boldsymbol{X}_c'\boldsymbol{X}_c \tag{2.18}$$

Here $X_c = X - 1_n\bar{x}' = CX$ with $\bar{x}'$ is mean variable vector and $C = I_n - n^{-1}1_n1_n'$ is a matrix which is in equation 2.16.

2.5.5 Eigendecomposition

A vector with dimensions of $N \times N$ can be considered an eigenvector if it meets linear equation [26]:

$$\boldsymbol{Av} = \lambda\boldsymbol{v} \tag{2.19}$$

here v represents eigenvector of linear transformation A which has eigenvalue λ. If all of our eigenvectors are represented by Matrix V and Eigenvalues by diagonal matrix L we can express covariance matrix C as:

$$\boldsymbol{C} = \boldsymbol{VLV}^{-1} \tag{2.20}$$

Eigenvectors represents the direction of the largest variance in data and the corresponding eigenvalue is the corresponding magnitude of this variance.

Basis vector of transformation matrix can be visualized by displaying a eigenvector $\boldsymbol{v}$ that is multiplied by $\sigma = \sqrt{\lambda}$. Due to *3 sigma rule*, which states that almost all values are situated within three standards deviations of mean [27].

2.5.6 PCA to determine object orientation

Object orientation can be determined by using PCA [25]. In this case PCA is applied to the edge of an object which requires preprocessing of image: thresholding, thresholding, and edge detection.

Object orientation is used on aerial images where it is difficult to determine object orientation due to its shape and low resolution [28].

CHAPTER 3

Implementation

3.1 General Structure

The data set consists of two series of images taken with different lenses.

Eleven images are available for the first lens and twelve for the second.

A text file for each camera containing the list of uncompressed file names is used to provide the input to the program.

The main function of the program opens the text file and a second file stream to direct the output which is a text file (details about its content will be discussed soon). C++ `fstream` functions allows one not to worry about specifying the mode in which a file is opened (reading, writing, or both).

The input file is read iteratively, line by line. At each iteration, the name of one of the uncompressed images is taken. The correspondent uncompressed file is read and a gray-scale image produced for internal use. This image is processed to prepare it for the next steps and a new gray-scale image is created. Then, from this last image a binary image is created. Finally a function is called that complete the rest of the necessary tasks.

The final result of the program is a text file containing the relevant information about single stars found in all the images, so the file will contain a long list organized into four columns. With reference to a single line, the five fields are, from left to right: an identification string for the star, the distance of this star from the center of the image, the eigenvalue representing the magnitude of the highest aberration, and the centroid coordinates of the star. The format of the identification string is:

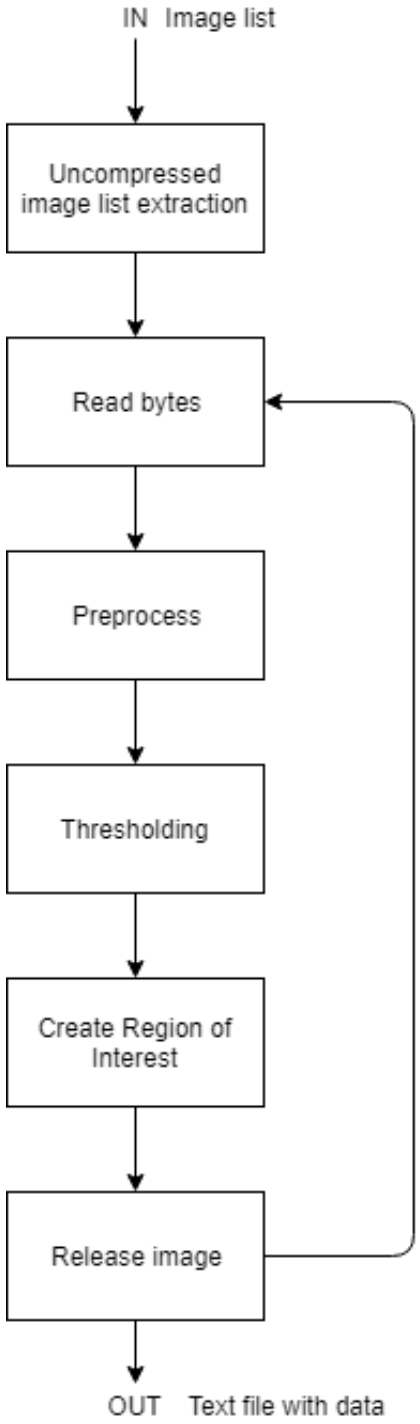


Figure 3.1: `main` function.

`ROI_i_s` , with i being the number of the image in the set and s being the number of the stars that has been identified.

Note that we have assumed that the center of the image corresponds to the optical axis. This assumption is not true in general, but it was not possible to assess what part of the image precisely was the actual zone correspondent to the center of the lens.

The program also produces a file called data.txt containing minimally formatted information about results of the PCA processing performed on the last set of ROI. It stores data mainly as an output log to refer and see if there were any anomalies.

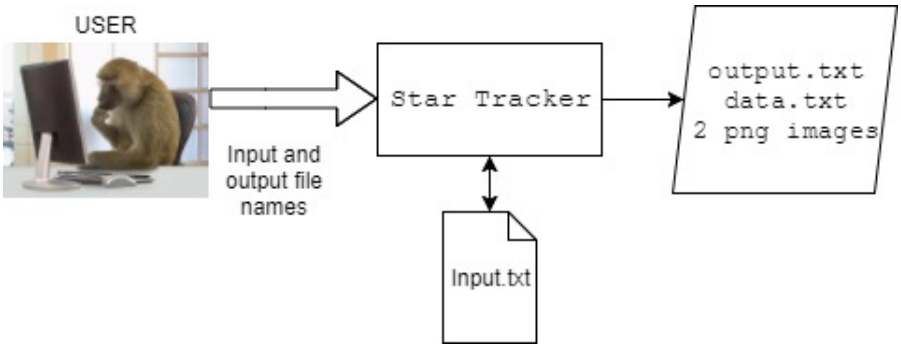


Figure 3.2: `main` function summarized.

3.2 Reading Images

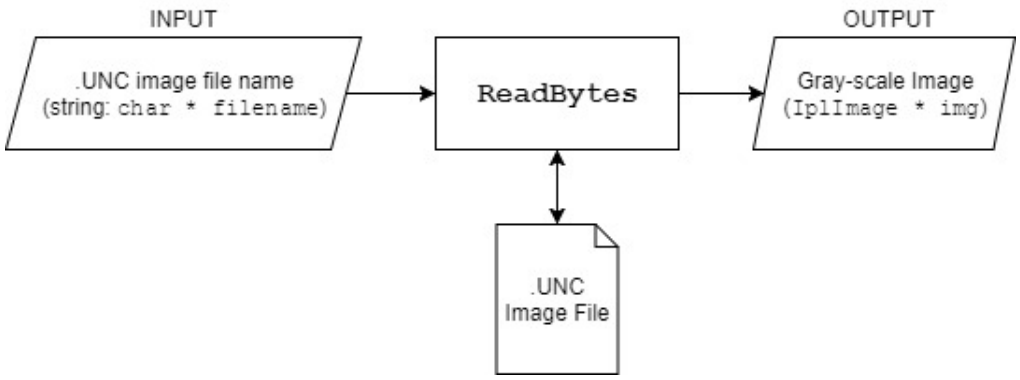


Figure 3.3: `ReadBytes` function.

Images have been downloaded from the μASC system as uncompressed (binary files with extension .unc). The header of each file is composed by the first 34 bytes in the file and consists of 17 different fields. The fields of interest for reading the images are listed in table 3.1.

Field	Description	Dimension	Offset
H	Height of the image	2 Bytes	28
W	Width of the image	2 Bytes	30
IMOD	Interlacing Mode	1 Byte	32

Table 3.1: Fields of interest in the uncompressed file header.

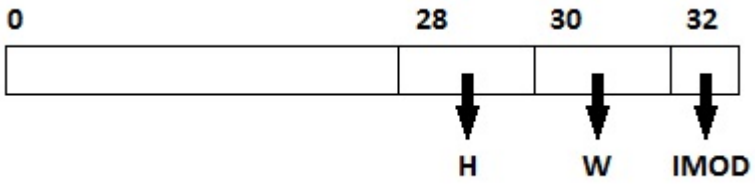


Figure 3.4: Visual representation of the .unc file header.

IMOD is a single byte representing a value of 1 if the image is interlaced of 0 otherwise. In the present case, images are interlaced. This influences the height of the image that is then calculated as H·(IMOD + 1).

The function designed to read the data is

```
IplImage * ReadBytes(const char * filename);
```

`ReadBytes` opens a binary file, whose name is specified by `filename`, in reading mode, extracts the first 34 bytes, and reads out the values of H, W, and IMOD. Then, the rest of the file is read byte by byte and each byte assigned to an element in a $W \times [H \times (IMOD + 1)]$ matrix which will then converted to an object of the OpenCV class *Mat*, representing the final image. This is finally converted to a OpenCV IplImage structure and a pointer to this structure is returned. The reason why the image is initially converted to a Mat object is because it is simpler to save data on a matrix and then convert a matrix to a Mat object, but in the end a pointer to IplImage is returned because the rest of the program mainly works with IplImage structures.

A last note is that images present some artifacts on the left side. The affected area extends for 10 pixels in width and for the entire effective height of the image. Therefore this area is cut out.

As an example, the first image from the first set is considered. If one wanted to print out an image, the result would be the one shown in **fig. 3.5**.

Figure 3.5: First image taken from the first camera; gray-scale, 752 x 580, IMOD = 1.

3.3 Preprocessing

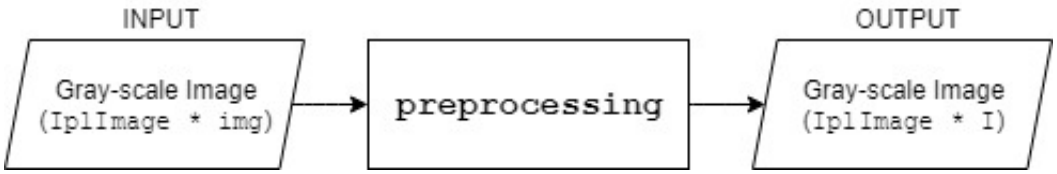


Figure 3.6: `preprocessing` function.

The very basic processing aiming at making a suitable image to be thresholded is carried out by the function

```
IplImage * preprocessing(IplImage * img);
```

This receives a gray-scale image, performs bilateral filtering on the input and stores the result in a new gray-scale image, the result of the filtering is then dilated, and finally a filtered and dilated gray-scale image is returned. Images are treated using pointers to `IplImage` structures.

Bilateral filtering has been chosen to remove some noise while keeping the original shape of the features as intact as possible. Bilateral filtering could remove textures, but this is not a concern in this case. Therefore a strong spacial filtering can be done as long as range filtering is kept low. To perform the filtering the OpenCV function `cvSmooth` has been used. This function employs a Gaussian bilateral filter but not *in place* (source and destination image must be different), so a new image has to be created. Note that with `IplImage` structures, `cvSmooth` does not allow to control the size of the kernel directly but gives control over σ_s and σ_r and the kernel is created internally basing on these values. The chosen values are $\sigma_s = 20$ and $\sigma_r = 0.05$.

A dilation transformation is performed because the images contain many small features that can be suppressed by the subsequent thresholding operation and may be impossible to detect in later steps. By dilating, these features are enlarged and easier to detect. The risk in that is that some segments can merge together and give a false detection of a single larger object. For this reason the dilation is limited to a single iteration. A standard square structuring element has been used.

The result at this point for the first image of the first set would be

Figure 3.7: Image of fig. 3.5 after a bilateral filtering with $\sigma_s = 20$ and $\sigma_r = 0.05$ and a single iteration of dilation.

3.4 Thresholding

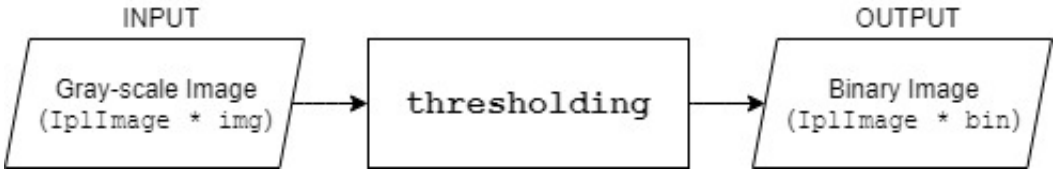


Figure 3.8: `thresholding` function.

In order to form binary images the function `thresholding` is called.

```
IplImage * thresholding(IplImage * img);
```

This function calculates the histogram of the input image by calling the function `calcHistogram`. Then, the background level, B, of the image is determined by calling the function `findBackground`, which calculates B as the 45% percentile of the histogram.

The threshold, t, is set to a fixed value of 10. The effective threshold used to form the binary image is then calculated as T = B + t.

Finally, the function
`simpleThresholding` is called. This is where binary images are built using the value of T.

The functions `calcHistogram` `findBackground`, and `simpleThresholding` have been written by the authors of this report. There is nothing special about the first two. `findBackground` takes a histogram of an image and the total number of pixel of the same image and returns the intensity value correspondent to the percentile that is the closest to 45% because it is not always guaranteed that an histogram has exactly the 45% percentile.

In this section of the program two choices have been taken arbitrarily: the value of t and the percentile to calculate the background B. After several tests these values seemed to be reasonably good at preventing the loss of small features and at avoiding the inclusion of noise-driven segments.

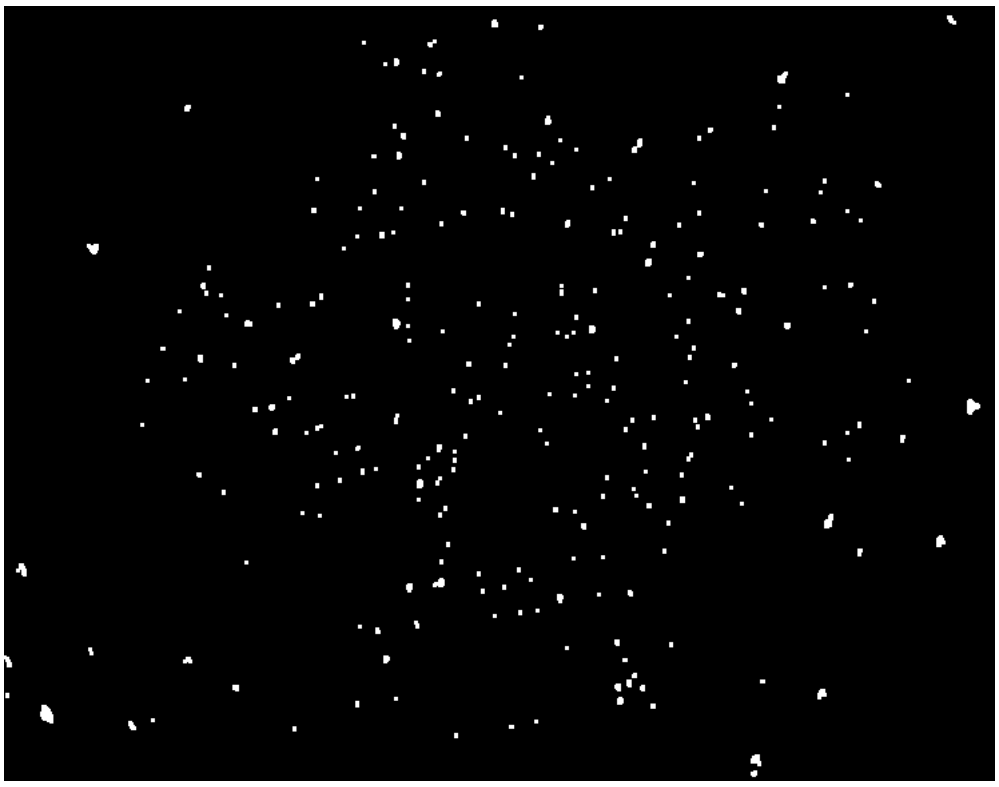


Figure 3.9: This is how the first image of the first set would look like after thresholding. The most prominent features are clearly visible as chunky blocks.

3.5 Regions of Interest

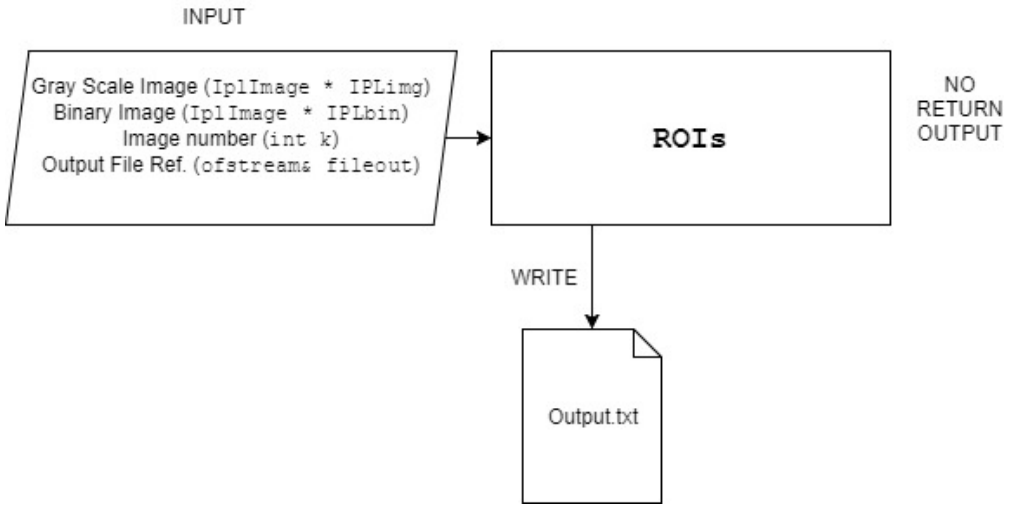


Figure 3.10: `ROIs` function.

This is the core of the entire program functionality.

```
void ROIs(IplImage *, IplImage *, int);
```

The function `ROIs` takes as input the original, unprocessed gray-scale image, the binary image produced by the thresholding phase, the image number, and a reference to the output file opened by `main`.

The binary image is used to pinpoint the individual features. This is accomplished by finding the contours of separated segments in it. In this step some features are filtered out because they are too small. Specifically, segments with less than 5 contour points are discarded. The reason for that is that 5 contour points is the minimum to complete the next step.

Once all the separated segments of the image are identified and the contour points saved, for each segment the normalized first order central moments are computed and used as centroid coordinates.

Then the function scans through all the centroid coordinates to analyze an area of 20×20 pixels around the centroid after up-scaling it to 400×400 pixels. These sizes have been chosen arbitrarily. The analysis is carried out by the function

```
bool check(Mat img);
```

It should be noted that when considering a 20×20, if a region does not remain within the image boundaries, it is not even taken into consideration because some important details could be clipped at the borders.

This function performs 2 tests on the up-scaled 400×400 pixels area of interest in order to exclude regions of interest containing features that are too faint to be treated and regions with multiple segments in them, i.e. features that are too close to each other to be treated precisely in the final steps of the image analysis. If one of the tests is not passed, the next region of interest is analyzed.

While the regions are scanned, for each image, 2 copies of the original gray-scale image are used to highlight all the pinpointed features and the those that have been extracted in the end as a visual comparison.

In implementing the function `ROIs` we relied completely on some OpenCV functions and classes outlined below.

To find the contours, we used the function `cv::findContours`. This method takes as input a binary image, an object of class `vector` of `vector` objects of objects of class `cv::Point` to save the contour points, for each segments. This may seem a little confusing, so fig. 3.11 illustrates the result.

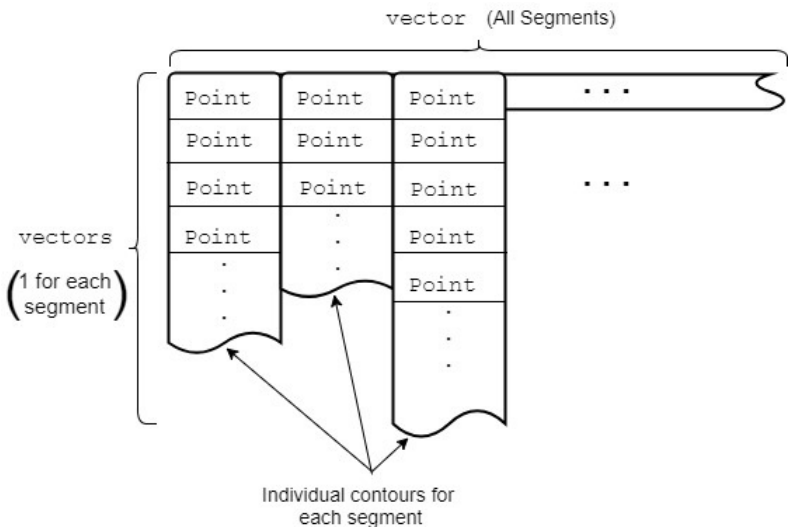


Figure 3.11: `Vector` of vectors of `Point` objects.

`cv::findContours` also constructs a hierarchy of contours basing on the mode selected, but this is not important in this application so it will not discussed any further. Moreover, the function has options to approximate a contour rather than saving all the points that it can find. We did not choose to approximate the contours (by specifying the value of the argument method as `CV_CHAIN_APPROX_NONE`) because the features of interest in the image are very small and also as many points as possible are needed to have a more accurate calculations of moments. The latter is also the reason why in the preprocessing phase the image has been dilated: to maximize the number of contour points.

To calculate the normalized first order central moments, we relied on the function `cv::moments` which calculates all of the moments up to the third order of a polygon described as an array ($1 \times N$ or $N \times 1$) of 2D points (objects of `Point` or `Point2f`), so each contour found previously can be used as input for `cv::moments`. The result is an object of class `Moments` whose attributes are the various moments so before calling `cv::moments` we create a `vector` of `Moments` objects. Unfortunately, the function does not calculate the normalized first order central moments, so they must be explicitly computed using the first order spatial moments and the 0th order spatial moment and the results stored in a `vector` of `Point` objects. Note that the function is more general and can work also with images directly (for details, OpenCV documentation is available) although we did not do that.

The function `ROIs` is also responsible to form the identification string of each individual star. It is reminded that the format of such string is ROI_i_s, with i and s being numbers of image and star respectively. The function receives from `main` the image number and it has the number of the star available as it scans through all the contours found previously. This construction is done in the iteration.

In each iteration cycle, also the distance of each star (of the point defined by the centroid coordinates) from the center of the image is calculated simply as

$$R_n = \sqrt{(x_n - x_o + 10)^2 + (y_n - y_o)^2} \tag{3.1}$$

for the n-th star, with $(x_o, y_i o)$ being the centre of the image. Note that since at the beginning the left border of the image has been cut out - and that region was 10 bits wide - the centre of the image we are working is not the real center of the image acquired by the camera. The latter is 10 bits shifted to the left with respect to the centre of the image used in the program.

The eigenvalue to be printed out in the end is also computed in the same iteration cycle by calling the function

```
double analysis (Mat gray);
```

Finally, also within an iteration, the output quantities are printed in the output file.

The function also print directly on screen the number of total stars that it manages to pinpoint and the number of "useful" stars - that have enough contour points.

As a side note, in `ROIs` we mostly used C++ tools (functions and classes) whereas in the previous section we mostly used C. The reasons are basically two. First, in the previous sections there is a wide re-use of functions from previous exercises and C had been used mostly. Second, some of the operations performed within `ROIs` are made easier by the C++ implementations of OpenCV functions with the support of classes. For example saving contour points is a lot easier to manage in C++. Fortunately, OpenCV makes it easy to convert images from `IplImage` structures (C style) and `Mat` objects (C++ style) by calling `cv::cvarrToMat`) and the inverse conversion is possible too.

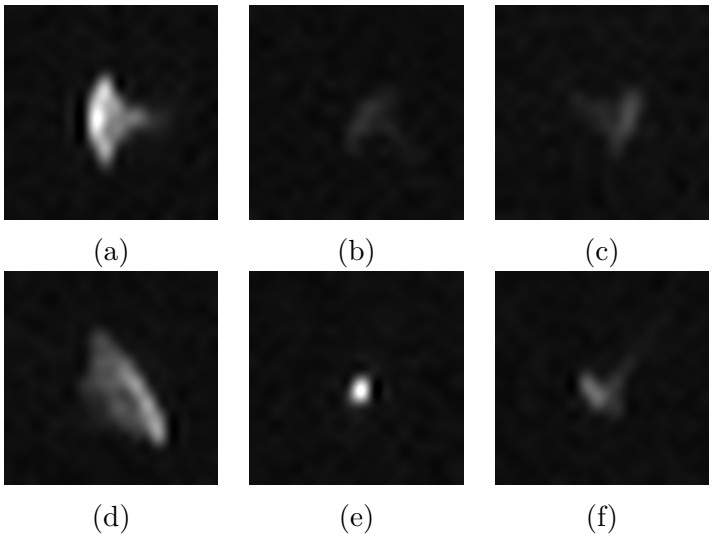


Figure 3.12: Some ROI images for the first camera. In a, b, c, d, and f the aberration is evident but as seen in d and e other types of shape are also extracted.

3.5.1 ROI Images Rejection

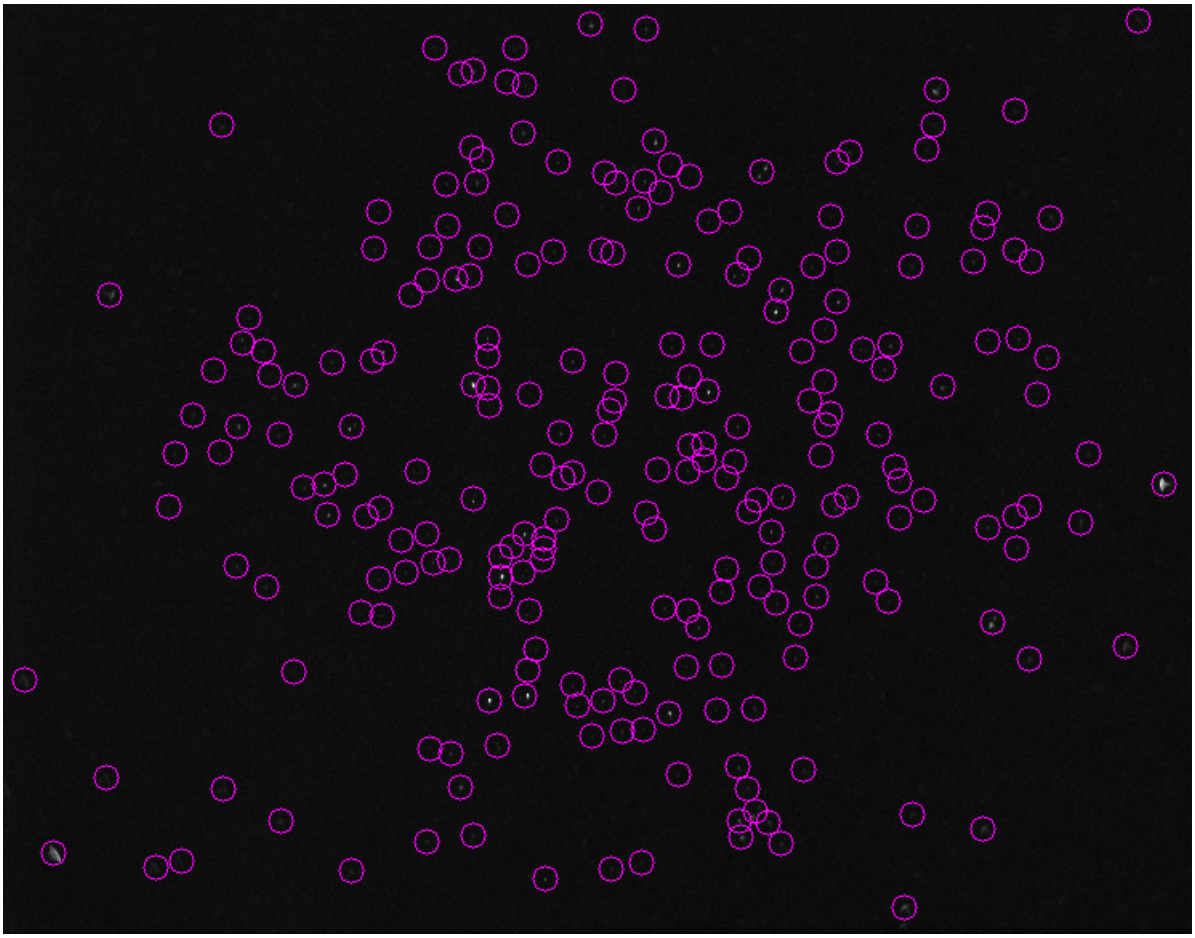


Figure 3.13: All ROI found, before rejection for the same image considered in previous figures.

It has been mentioned above that regions of interest are discarded according to how faint features are or depending on how many features are found within them. Specifically, the function `check` calls first another function to compute the contrast in the region of interest image (ROI) and if this contrast is lower than 10, the ROI is considered too faint and `false` is returned, otherwise the next check is done.

In the next check the multiplicity is analyzed. This requires some processing in the ROI to enhance some features. This include: 12 iterations of opening, 10 iterations of erosion, 3 iterations of dilation. Note that since ROI is a gray-scale image converse morphological transformation do not cancel each other exactly so, for example, 10 erosion followed by 3 dilation do not produce the same result as 7 erosion. Finally, thresholding is performed and segments are counted using `cv::findContours` in the binary image (in this case contour points are not the main interest so an approximation method can be applied to compress the data.

Unfortunately, a rather faint ROI can pass the previous check so it can occur to have situations of multiple faint objects or multiple bright objects in the same ROI. For this reason we employed multiple thresholds. As the threshold is raised, binary images for each ROI are created sequentially. If 2 or more objects are found in at least one of the binary images the check returns `false`. The 2 extreme thresholds chosen are: 10 and 20 (these values are superimposed to the background level of the ROI).

The fixed values chosen for the minimum contrast and for the thresholds have been found through a calibration process described below.

A final note: we decided to perform the check for faintness before that for multiplicity because faintness is the most prominent situation and by checking it first, the second check is performed on a considerable smaller number of ROI images, thus the overall computation is reduced.

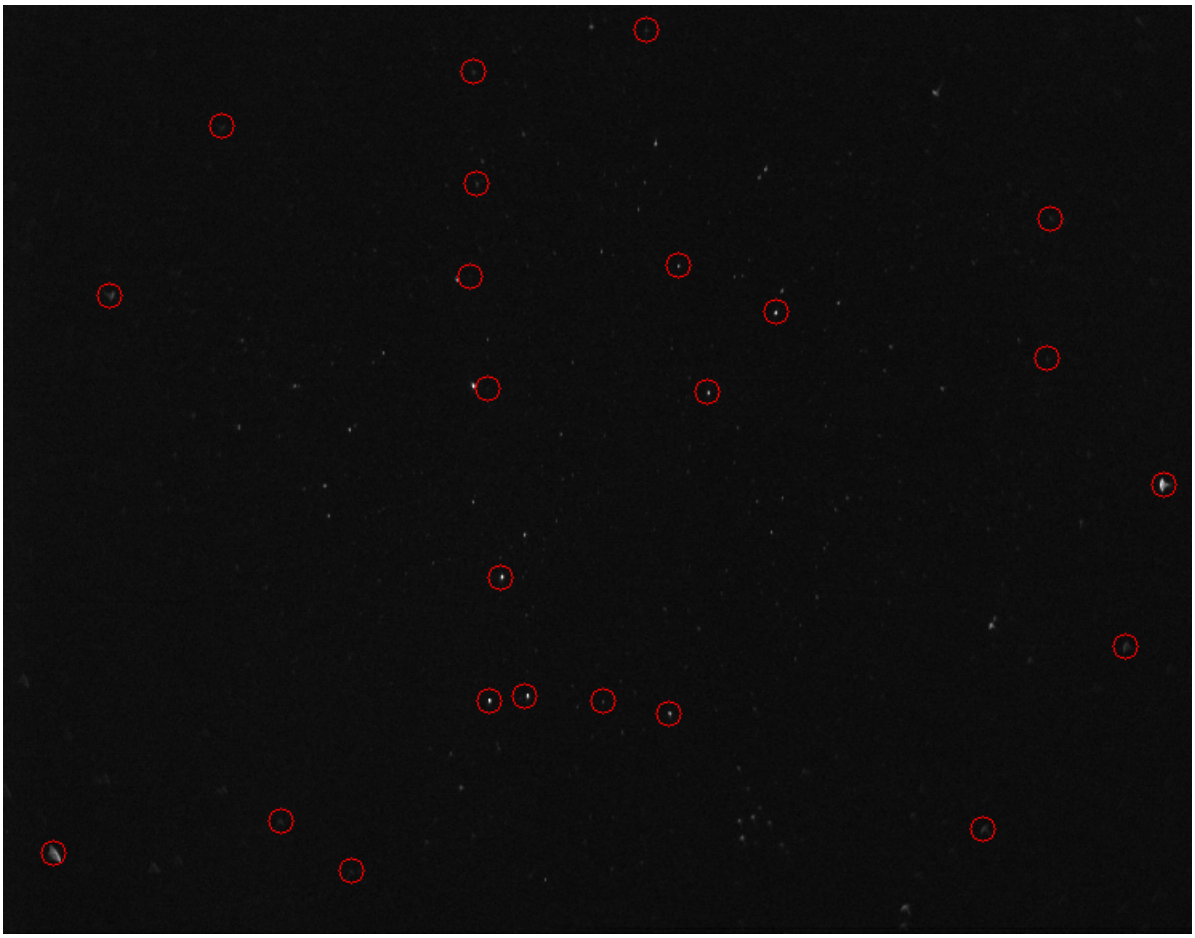


Figure 3.14: These are the ROI after the rejection.

3.5.2 Calibration

As seen above, in the rejection process fixed values have been used to perform checks against and make a decision. In order to find these fixed values the following tests have been conducted.

First, an image has been chosen from either set and all the possible ROI images have been produced for both.

To find a reasonable value to check the *faintness* of the ROI images, those ROI images that were visually very faint have been selected for a total of 163 images. For each and every single image the contrast has been determined. Here contrast is considered as the difference between the 95% and the 5% percentile of the histogram. Results have been collected and the mean value together with the standard deviation have been considered and are equal to 7 and 0,981423 respectively. Thus, the minimum accepted contrast is it has been chosen equal to 10 ($mean + 2\sigma + 1$). This value is effective for most cases, but sometimes a few (1 to 3 for each entire image) somewhat faint ROI images are not suppressed.

In regard of *finding multiple segments*, the same algorithm employed in the function `check` based on thresholding and already described was the baseline for the calibration. One image from either set has been taken as samples producing a total or 99 ROI images with multiple features visible in them as well as images with noticeably elongated and thin segments that may have been likely to be broken into multiple objects in a thresholding operation. The rationale was to find a compromise between separation of segments in close proximity and the avoidance of breaking a segment in multiple parts. By running the algorithm on all images it has been determined a suitable sequence of specific morphological transformations and the number of iteration for each operation. Then a suitable threshold had to be found but it turned out to be impossible to use a single value with the present algorithm because of multiple segments with different brightness. For example a certain threshold suited for bright segments would mask those that are less bright whereas a value that works well for relatively faint objects could cause brighter regions to merge. Hence, more threshold values have been used and the ability to find multiple segments for at least one value without separating objects with all thresholds has been observed by inspection of the original image. It resulted reasonable to use thresholds between 10 and 20 with a step equal to 4. Below 10 false segments sometimes appear; above 20 some segments start to break and some object may be suppressed.

This method is not perfect of course but worked reasonably well for most images. Specifically, out of 99 images only 5 gave false results.

3.6 Principal components analysis

PCA is a key part of program which as an input takes a ROI and gives Eigenvectors, Eigenvalues, and Orientation as output. The eigenvalue used for the quality assessment belongs to the largest eigenvector which indicates the largest aberration.

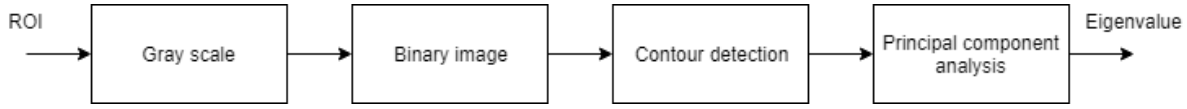


Figure 3.15: Processsing of ROI to get values that define star in it

3.6.1 Input

The input for this process is a ROI image extracted from the star tracker image. This ROI is a `cv::Mat` object that represents an image of 400 by 400 pixels. The image is passed by `ROIs` to the function `analysis` as already mentioned.

This function form a binary image employing the Otsu's method through the Opencv function `cv::threshold` and combining `THRESH_BINARY` flag with `THRESH_OTSU` flags to create a binary image. Otsu method automatically determines the threshold value. However, this method is not the best choice for images because it is thought for images with a bimodal histogram as mentioned before. Nonetheless, it provides a threshold value that is usable at this stage. A well defined star is not compromised after being converted to binary image. But for a ROI where star is dim and that has a complex shape it is giving results that are less than optimal because of the introduction of segments that do not correspond to usefl shapes. In some cases a star that at naked eye is visible becomes a blob that only retain its shape extension. This gives an approximation of star shape which can still be used but result is not as accurate.

On this binary image a contour search is performed to provide data to the last function called by `analysis` in the processing that computes the eigenvalue.

3.6.2 Contour finding

Object orientation detection is done by using the shape of it and here is where the contour plays a role.

Data on which PCA is performed is found by `cv::findcountours` function already encountered in `ROIs`.

A possible alternative method is Canny edge detection. However, `cv::findcountours` was chosen since it provides us with data set which was found to be sufficient enough for a prototype version of software and does not require filtering in preprocessing. Moreover, `cv::findcountours` allows one to count the segments in an image whereas Canny edge detection only provides a visual representation of edges and would require further processing to extract segments.

Each contour found by this function is further sorted in an ascending order and the largest contour is used in PCA. This is done since there are cases where more than one contour is found and in most cases multiple contours are a result of using the Otsu's method. No processing is performed on ROI images since the star in ROI is already affected by a non suitable automatic thresholding filtering would remove even more information about a star, making its aberration indistinguishable.

3.6.3 Finding Eigenvectors and Eigenvalues

For this project PCA was implemented using function pca_analysis provided by OpenCV image processing library. This way was chosen because it provides all of the steps required for PCA in a fast and compact manner. It internally computes covariance matrix, mean values, mean value subtraction as well as eigenvectors and eigenvalues. For this function however data input format is not as in typical PCA, where data is reshaped into two rows where each column represents data point, but as two columns where each row is representing data point. This is due to the way PCA functionality is written in OpenCV. This does not change procedure since for PCA first step is reshaping of data. Function to perform PCA analysis is called by function:

```
double performPCA(contours[contours.size() - 1], src)
```

Since contours are sorted in ascending order we need to pass the last element in the vector to access it. We also pass the original ROI as means for illustrating the procedure on individual ROI. This is removed later in code to reduce processing time since one of requirements is speed and visualisation, as well as saving of results, take up extra time, processing power, and memory. However, it was left in in call since a final version of PCA should take into consideration intensity of pixels of each data point. This would help to characterize star not only in terms of direction and strength of aberration but in intensity of aberration as well.

Return value of this function is the largest eigenvector eigenvalue. This eigenvalue is the magnitude of aberration that is most dominant. Meaning that one component is larger than the other. Stars in the centre of the image - well in focus - have lower eigenvalues which directly corresponds to smaller aberration extension (**fig. 3.16**), meaning that they are close to a small circular object. If one of the components is stronger, then eigenvalues is higher (**fig. 3.17**) allowing us to provide a quality indication of specific regions of the images and in turn to make a conclusion on the quality of the lens that produced those images.

Figure 3.16: Star with a high eigenvalue

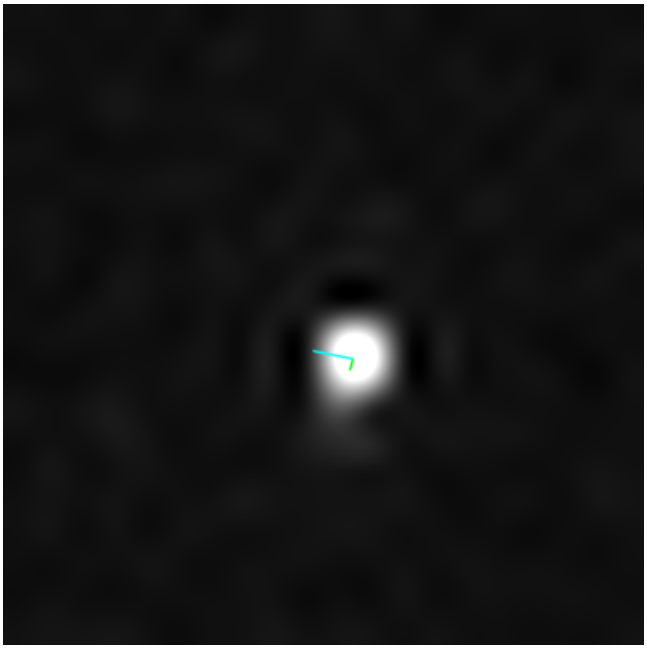


Figure 3.17: Star with a low eigenvalue

3.7 Improvements

Because of the limitations discussed, improvements are evidently required in multiple parts of the project.

In general, the efficiency of the code has not been evaluated and in some parts may not be optimal. Moreover there is a mixing of C and C++ which is not very elegant.

In several steps, parameters have been set arbitrarily, e.g. thresholding, bilateral filtering. This may have compromised results and affected intermediate steps in the processing. Statistical methods allowing to select these parameters more adequately and optimally, and automatically are instrumental.

In morphological transformations a squared structuring element has been used, which is not an optimal choice for shape preservation. A more prudent choice of structuring elements is certainly a step to take towards improvement.

Centroiding is only important to pinpoint stars in an assessment task. This aspect has not showed many faults as only a few stars have been excluded because of it but a better method would be a plus since dilating segments to ease their research can cause merging of them and lead to false results.

A more precise selection of region of interest, possibly based on the actual shape of features is a great improvement because considering a square area around a centroid caused many small features to fall within the same area and be rejected.

Rescaling caused clipping. Now this problem was never present close to edges so it has been accepted, but avoiding it altogether would be better.

A discrepancy in the way data is treated emerged due to miscommunication between members of the team, but not corrected in time. In fact, ROI images have been checked to avoid multiple segments in the same area and should have been treated using simple thresholding with one of the thresholds employed in the calibration process. This would have ensured to have ROI binary images with a single segment instead an unsuited automatic method for thresholding has been used and this will likely cause a lot of mistakes.

In the PCA section, it has been assumed more or less implicitly that the sagittal component is dominant. This statement is actually weak as the aberrations are a complex phenomenon.

CHAPTER 4

Results

The value of the eigenvalue correspondent to the sagittal component of aberration has been plotted against the distance of the centroid of a star from the centre of the image.

From the theoretical knowledge of aberrations, one would expect to notice an increasing trend as aberrations are more pronounced at the edges of the frame.

We can see on plot one (**fig. 4.1**) and two (**fig. 4.2**) that as the distance increases - receding from the centre of the image - the higher magnitude of aberration becomes. The trend is consistent with our expectation. However, the plots also present a certain amount of outliers. These outliers can be the result of many sources. These probably include non-optimal data pre-processing, errors introduced by some operations like interpolation, non-optimal thresholding at different phases of the processing, and failures at rejecting unsuited features for measurements.

The magnitude of aberration is arbitrarily found by:

$$MoA = \frac{eigenvalue}{heightoftheimageinpixels} \tag{4.1}$$

This measure of quality is proposed in article by Ian Norman [4] as a test to qualitatively assess aberrations in lenses used for photography. In this article he uses size of the largest aberration present in the image of white spots on a black background, the extension is measured in pixels and compares it with total height of the image.

We assume that there is a direct relation between the eigenvalue and the size of aberration in pixels since the more pixels aberration take up, the larger the correspondent eigenvalue will be as seen in (**fig. 3.16**) and (**fig. 3.17**). Therefore we use the eigenvalue instead of pixels to indicate the magnitude of stretching of the point image.

Therefore, the equation 4.1 provides us with an indicative measure of the aberration for any given star.

A comparison of the 2 cameras can be attempted for distances between 150 and 400 (**4.3**) since outside of this range the number of samples is low. It can be seen that the first camera has a higher dispersion of results whereas the second camera has more consistent values but in both case the increase seems to fit a similar curve. The magnitude measured in both cases vary in approximately the same range, specifically for the first camera ranges from around 0.471 to around 5.708 whereas for the second camera slightly higher values are seen in the near range, around 0.607, but are somewhat lower as one proceeds towards the edges of the image to approximately 5.22.

We are also provided with a small set of images that we know they are unfocused (**fig. 4.4**) to compare the results with those of the 2 cameras. In this case higher values of magnitudes have been observed, going from approximately 1.3 to around 5.73, which would be what we expect, but the set is considerably under-represented so the comparison may not be meaningful.

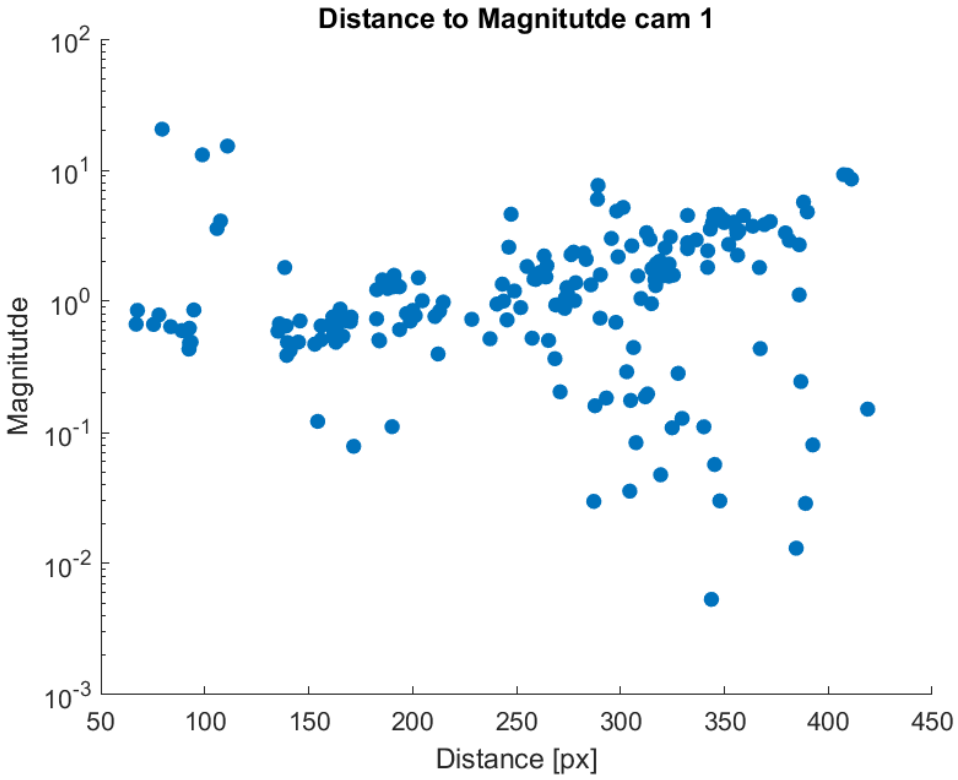


Figure 4.1: Distance to Magnitude of camera 1

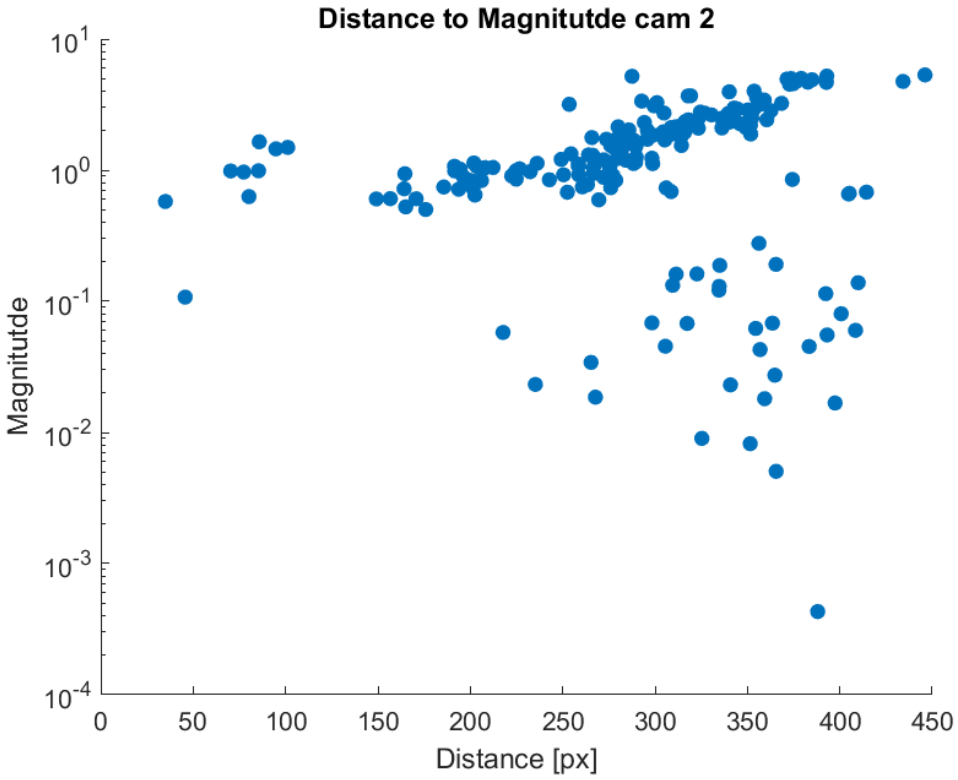


Figure 4.2: Distance to Magnitude of camera 2

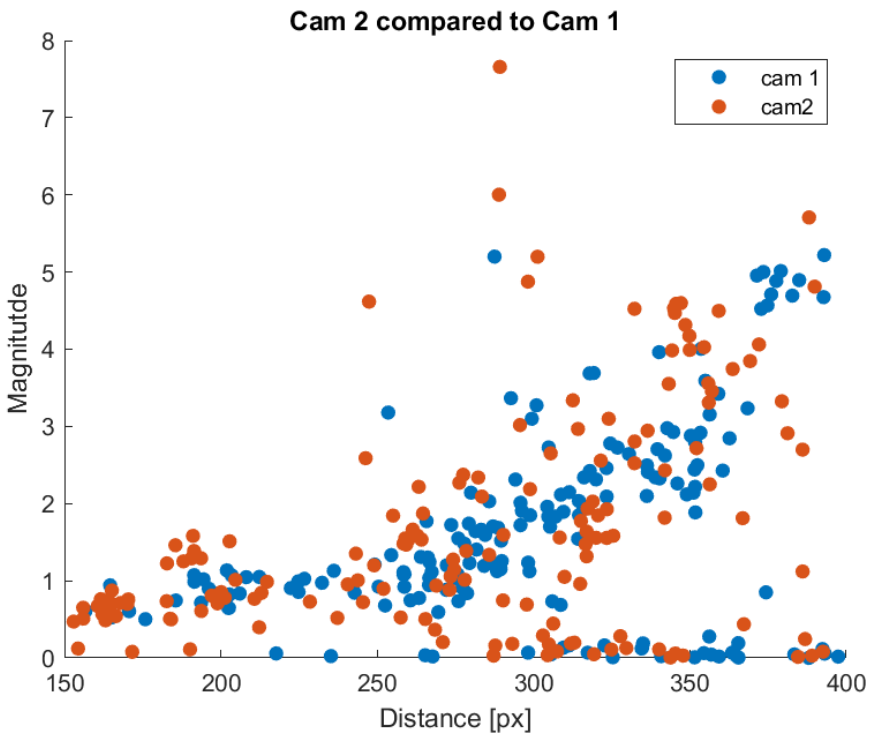


Figure 4.3: Comparison of the results for the two camera for distances between 150 and 400.

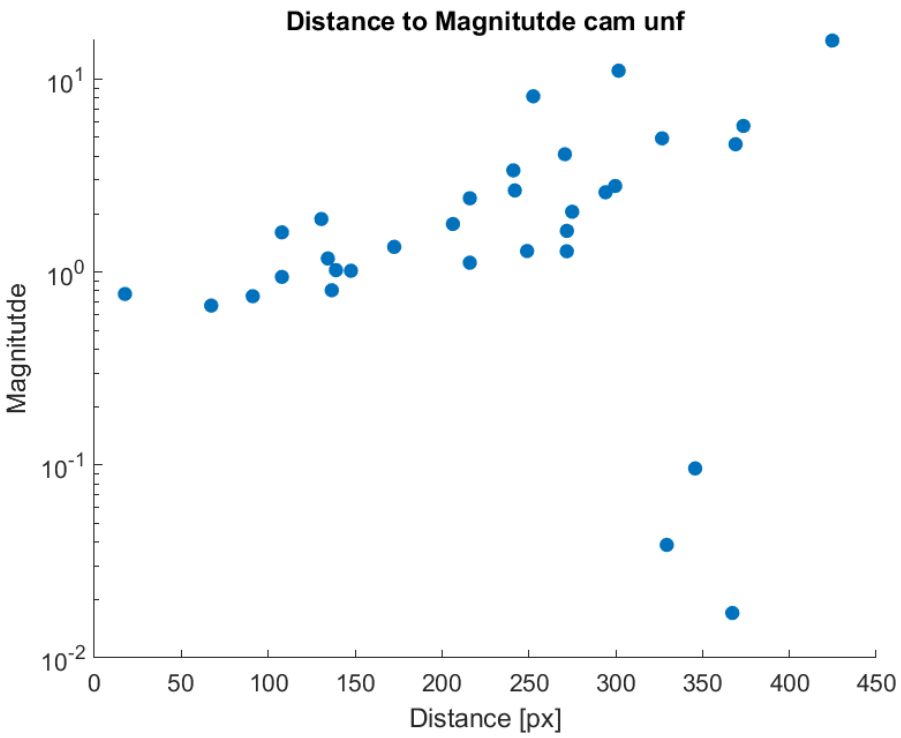


Figure 4.4: Magnitude vs Distance for the unfocused set.

CHAPTER 5

Conclusion

Principal component analysis was investigated to see the potential to identify the degree of aberration in lenses.

The analysis has been conducted on every star detected in 2 sets of 11 and 12 images taken by 2 different lenses.

The results are consistent with the theory of aberrations and can describe at least the behaviour of lenses. However, since there is not a theoretical model that describes the relation between eigenvalues and the extension of aberration, the accuracy of the method cannot be assessed since it is not known exactly how this value is supposed to vary with the distance from the center. A smaller set of unfocused images is also been investigated and results meet the expectations but the same conclusions as above apply for this case which is also under-represented.

We don't have a knowledge of the two lenses but it would seem like that the first lens experiences slightly more aberration. This conclusion may not be reliable though considering the uncertainty of the results and the fact that the program produced more samples for the second camera, in the observed interval.

Preprocessing involved calculating centroid of stars via moments, isolation of individual star and generating region of interest out of it, scaling region of interest up to produce more data points for analysis, calibrating regions of interest for rejection, binarization of region of interest. These steps may be the cause of uncertainties since many simplifications have been made.

A quality figure is derived from eigenvalue which belongs to the highest aberration through using equation 4.1 but the method should be backed up by a solid theoretical background.

Data acquired through this method was visualized via Matlab. Graphs showing distance to magnitude are given where a clear trend of increase of magnitude as it approaches edge of image is visible.

CHAPTER 6

Future work

An automatic thresholding method for unimodal histogram concentrated in the shadows has been mentioned but not implemented, it would be probably more performing considering that it matches the situation encountered in star tracker images but would require additional testing to ensure that it is actually suitable.

In general, whenever a parameter has been set arbitrarily, an automatic, more reliable method has to be found, or the calibration procedure should be refined.

Various improvements in the implementation are necessary as discussed in the relative section.

The PCA should be performed over a three dimensional array holding position of pixel and intensity instead of a simple two dimensional array holding only pixel position. This would help to compare not only magnitude of aberration based on size but as well as intensity of aberration.

Overall efficiency of the implementation should be examined and improved when necessary.

Lastly, investigations about the method for applications in lens quality studies should be conducted as in this report only the potentiality the it could be useful has been examined. Model to compare the results to are in fact necessary.

Bibliography

[1] Measurement The Space Instrumentation Group and Instrumentation Systems National Space Institute. *The micro-Advanced Stellar Compass*.

[2] `http://www.specinst.com/WhatIsACCD?`.

[3] G.G. Bennett. *The Calculation of Astronomical Refraction in Marine Navigation*. 1982.

[4] `https://www.lonelyspeck.com/a-practical-guide-to-lens-aberrations-and-the-lonely-speck-aberration-test/`.

[5] `https://www.handprint.com/ASTRO/ae4.html`.

[6] J. Sasián. *Introduction to Aberrations in Optical Imaging Systems*. Cambridge University Press, 2013. ISBN: 978-1-107-00633-1.

[7] M. J. Kidger. *Fundamental Optical Design*. SPIE, 2002. ISBN: 0-8194-3915-0.

[8] V. N. Mahajan. *Optical Imaging and Aberrations Part I – Ray Geometrical Optics*. SPIE, 1998. ISBN: 0-8194-2515-X.

[9] V. N. Mahajan. *Optical Imaging and Aberrations Part II – Wave Diffraction Optics*. SPIE, 2004. ISBN: 0-8194-2515-X.

[10] Ramesh Jain, Rangachar Kasturi, and Brian G. Schunck. *Machine Vision*. McGraw-Hill, 1995. ISBN: 0-07-032018-7.

[11] S. Paris et al. *A Gentle Introduction to Bilateral Filtering and Its Application*. `https://people.ssail.mit.edu/sparis/bf_course/course_notes.pdf`. Course Notes.

[12] C. Tomasi and R. Manduchi. *Bilateral Filtering for Gray and Color Images*. Proceedings of the 1998 IEEE International Conference on Computer Vision, Bombay, India. 1998.

[13] G. Bradski and A. Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008. ISBN: 978-0-596-51613-0.

[14] N. Otsu. "A threshold selection method from gray-level histograms". In: *IEEE Trans. SMC* 9 (1979), pp. 62–66.

[15] A.D. Brink and N.E. Pendock. "Minimum cross-entropy threshold selection". In: *Patt. Recog.* 29.1 (1996), pp. 179–188.

[16] J.N. Kapur, P.K. Sahoo, and A.K.C. Wong. “A new method for gray-level picture thresholding using the entropy of the histogram”. In: *CVGIP* 29.3 (1985), pp. 273–285.

[17] P. L. Rosin. “Unimodal Thresholding”. In: *Pattern Recogn.* 34 (2001), pp. 2083–2096.

[18] M. Sezgin and B. Sankur. “Survey Over Image Thresholding Techniques and Quantitative Performance Evaluation”. In: *Journal of Electronic Imaging* 13.1 (January 2004), pp. 146–165.

[19] Pavlos Stathis, Ergina Kavallieratou, and Nikos Papamarkos. “An Evaluation Technique for Binarization Algorithms”. In: *Journal of Universal Computer Science* 14.18 (2008), pp. 3011–3030.

[20] JOHN CANNY. *A Computational Approach to Edge Detection*. NOVEMBER 1986.

[21] `https://opencv-python-tutroals.readthedocs.io/en/latest/pytutorials/pyimgproc/pycanny/pycanny.html`.

[22] MING-KUEI HU. *Visual Pattern Recognition by Moment Invariants*.

[23] `https://breadboardgremlins.wordpress.com/image-moments/`.

[24] R. C. Gonzales and R. E. Woods. *Digital Image Processing*. 2002.

[25] `https://www.tandfonline.com/doi/pdf/10.1007/BF02826615`.

[26] `https://datascienceplus.com/understanding-the-covariance-matrix/`.

[27] `https://www.encyclopediaofmath.org/index.php/Three-sigma_rule`.

[28] Wei Yi S. Marshall. *Principal component analysis in application to object orientation*. 2000.

APPENDIX A

Functions

A.1 Main

A.1.1 Note for use

- Every single uncompressed file must be placed in the same directory as the executable file.
- the name of the input file from where the list of images is to be read is entered as input from the console.
- the name of the output file must be entered from the console.

The console command has the format - assuming one uses 'Star Tracker' as name for the executable file:

```
> Star Tracker InFile.txt OutFile.txt
```

A.1.2 Source Code

Note: The first inclusion depends on the way Opencv has been set with the IDE. In my case, I used the free IDE "*Code::Blocks*" and the first instruction is sufficient to include all Opencv content.

```
1
2  #include <opencv2/opencv.hpp>
3  #include <iostream>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string>
7  #include <fstream>
8  #include <vector>
9  #include <iomanip>
10 #include <algorithm>
11
12 using namespace std;
13 using namespace cv;
14
15 // FUNCTION PROTOTYPES
16 // These are the most important functions.
```

```
17 IplImage * ReadBytes(const char *);
18 IplImage * preprocessing (IplImage *);
19 IplImage * thresholding (IplImage *);
20 void ROIs(IplImage *, IplImage *, int);
21 double performPCA(const vector<Point> &pts, Mat &img);
22 bool forsort(const vector<Point>& c1, const vector<Point>& c2);
23 double analysis(Mat gray);
24
25 // These are support functions for ROIs
26 bool check(Mat);
27 bool multi(IplImage *, int *);
28 int findContrast(IplImage *, int *);
29
30 // These are recurrent support functions used by more other functions
31 int findBackground(int *, int);
32 void calcHistogram(IplImage *, int *);
33 void drawHistogram(const char *, int *);
34 IplImage * simpleThresholding(IplImage *, unsigned);
35
36
37 int main(int argc, char ** argv) {
38     // OPEN INPUT and OUTPUT FILES
39     ifstream out(argv[1]);
40     ofstream fileout(argv[2]);
41
42     string line;
43
44     // READ LINE BY LINE
45     int k = 1;                // Image number
46
47     while(getline(out, line)) {
48         const char * imgFileName = line.c_str();  // .UNC file name
49
50         // READ IMAGE
51         IplImage * img = ReadBytes(imgFileName);
52
53         // PREPROCESSING
54         IplImage * imgP = preprocessing(img);
55
56         // THRESHOLDING
57         IplImage * imgPBin = thresholding(imgP);
58
59         // FIND ROI
60         ROIs(img, imgPBin, k);
61
62         // RELEASE IMAGES
63         cvReleaseImage(&img);
64         cvReleaseImage(&imgP);
65         cvReleaseImage(&imgPBin);
66
67         k++;
68     }
```

```
69
70      // CLOSE FILES
71      out.close();
72
73      return 0;
74  }
```

A.2 Reading Images

A.2.1 Flow Chart

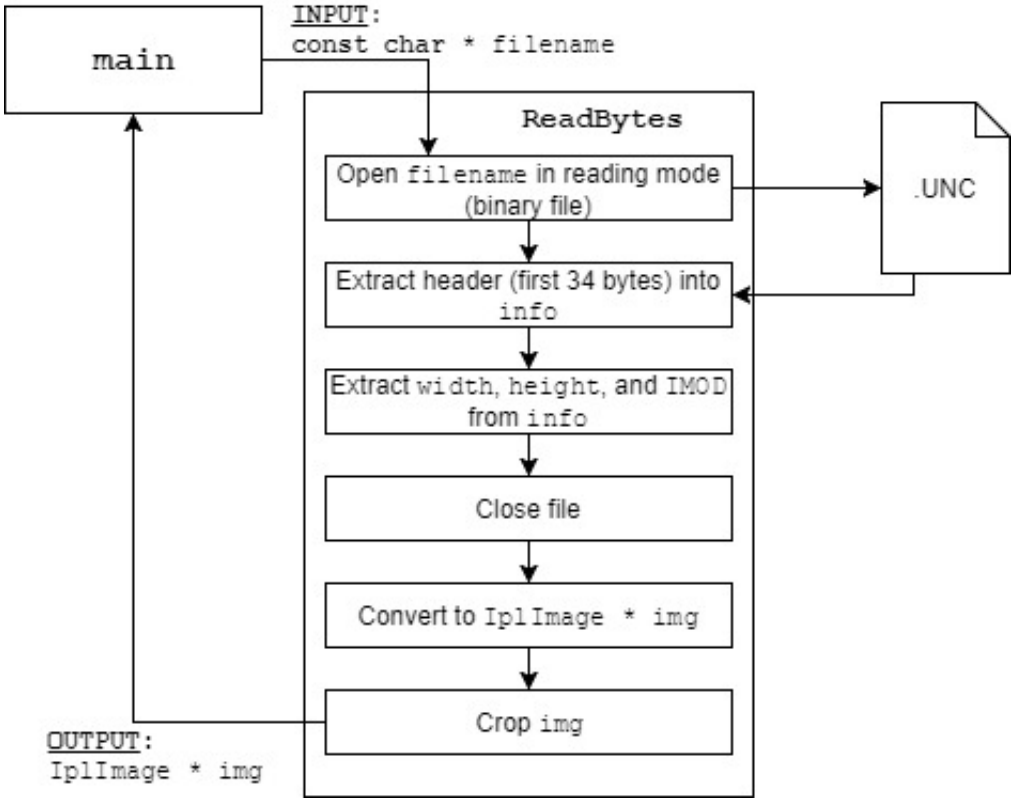


Figure A.1: `ReadBytes` function.

A.2.2 Source Code

```
1  IplImage * ReadBytes(const char * filename) {
2      FILE * f = fopen(filename, "rb");
3
4      if(f == NULL)
```

```
5           throw "Argument Exception";
6
7       // read the 34-byte header
8       unsigned char info[34];
9       fread(info, sizeof(unsigned char), 34, f);
10
11      // extract image height and width from header
12      int width = *(short*)&info[30];
13      int height = *(short*)&info[28];
14      int IMOD = *(short*)&info[32];
15
16      uchar data[height * (IMOD + 1)][width];
17      unsigned char temp;
18
19      for(int i = 0; i < height * (IMOD + 1); i++) {
20          for(int j=0; j < width; j++) {
21              fread(&temp, sizeof(unsigned char), 1, f);
22              data[i][j] = temp;
23          }
24      }
25
26      fclose(f);
27
28      Mat imageMat = Mat(height * (IMOD + 1), width, CV_8U, data);
29
30      IplImage * img = cvCreateImage(cvSize(imageMat.cols, imageMat.rows),
                                                    IPL_DEPTH_8U, 1);
31      IplImage ipltemp = imageMat;
32      cvCopy(&ipltemp, img);
33
34      cvSetImageROI(img, cvRect(0, 0, 10, height * (IMOD + 1)));
35      cvSet(img, 0);
36      cvResetImageROI(img);
37
38      imageMat.release();
39
40      return img;
41  }
```

A.3 Pre-processing

A.3.1 Flow Chart

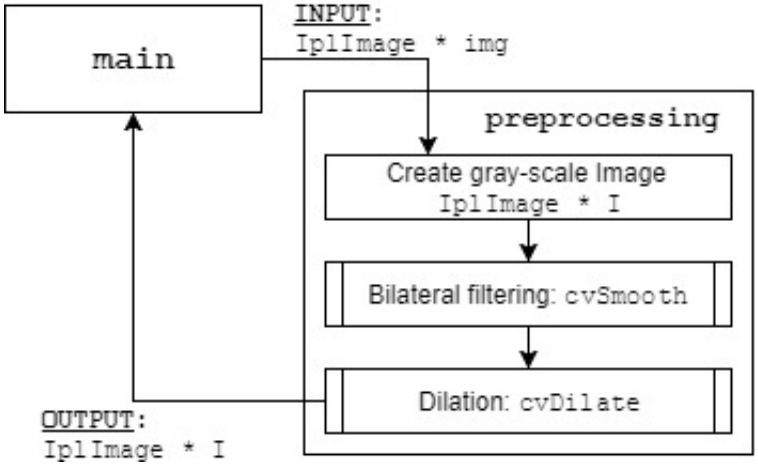


Figure A.2: `preprocessing` function.

A.3.2 Source Code

```
1
2  IplImage * preprocessing(IplImage * img) {
3      IplImage * I = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
4      cvSmooth(img, I, CV_BILATERAL, 0.05, 20);
5
6      int IT = 1;
7      cvDilate(I, I, NULL, IT);
8
9      return I;
10 }
```

A.4 Thresholding

A.4.1 Flow Chart

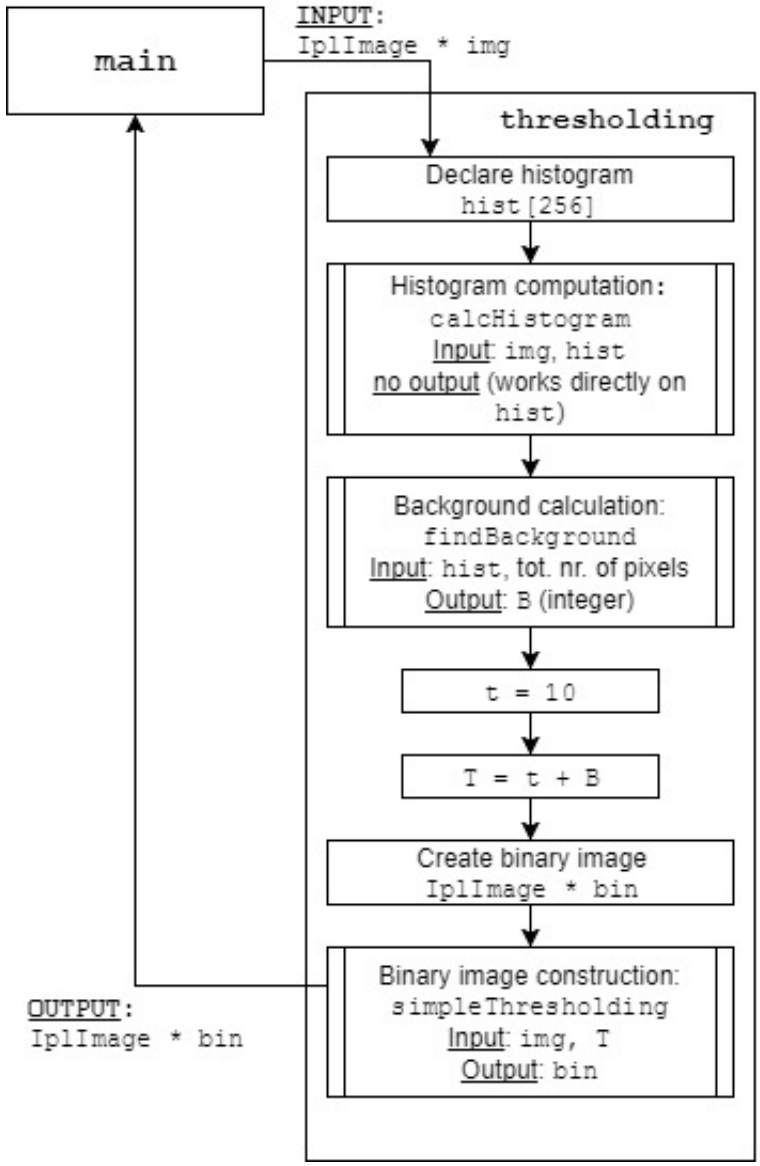


Figure A.3: `thresholding` function.

A.4.2 Source Code

```
1 IplImage * thresholding(IplImage * img) {
2     ///                          HISTOGRAM
3     int dep = 256;
4     int hist[dep];
```

```
5     calcHistogram(img, hist);
6
7
8     ///                 COMPUTE THE BACKGROUND = B
9     int B = findBackground(hist, img->height * img->width);
10
11    ///                  COMPUTE THE THRESHOLD = t
12    int t = 10;
13
14    ///                  FINAL THRESHOLD T = B + t
15    int T = t + B;
16
17    ///                     USE T TO THRESHOLD
18    IplImage * bin;
19    bin = simpleThresholding(img, T);
20
21    return bin;
22 }
```

A.5 ROI Selection

A.5.1 Source Code

```
1  void ROIs(IplImage * IPLimg, IplImage * IPLbin, int k, ofstream& fileout)
       {
2      Mat img, bin;
3      img = cvarrToMat(IPLimg);        // Binary Image
4      bin = cvarrToMat(IPLbin);        // Original Image
5
6      Mat bincpy = bin.clone();        // Copy used by 'findContour'
7
8      blur(bincpy, bincpy, CvSize(3, 3));
9
10     double yc = IPLimg->height / 2;
11     double xc = IPLimg->width / 2 - 10;  // Remember 10 pixels are
12                                          // cut out on the
13                                          // left side.
14
15     /// ////////////////////////////////
16     ///          FIND CONTOURS         ///
17     /// ////////////////////////////////
18
19     // vector of vectors of Point --> Table of x and y coordinates
20
21   vector<vector<Point> > contours;
22
```

```
23  findContours(bincpy, contours, CV_RETR_TREE, CV_CHAIN_APPROX_NONE, Point
        (0, 0));
24
25
26    /// ////////////////////////////////////////////////////////////////
27    ///   COMPUTE MOMENTS FOR EACH CONTOURS AND EXTRACT ONLY THE     ///
28    ///   NORMALIZED 1ST ORDER MOMENTS                               ///
29    /// ////////////////////////////////////////////////////////////////
30
31    int N = contours.size();  // Easier and less confusing to use N than
32                              // calling the method every time
33
34    /// ////////////////////////////////////////////////////////////////
35    ///           COUNT CONTOURS THAT HAVE ENOUGH POINTS             ///
36    /// ////////////////////////////////////////////////////////////////
37
38    // NOTE: Tests determined that 5 points is the minimum to be sure
39    // that moments can be computed correctly
40
41    int K = 0;
42    for(int i = 0; i < N; i++)
43        if (contours[i].size() >= 5)
44            K++;
45
46    /// ////////////////////////////////////////////////////////////////
47    ///     RECAP OF THE COUNT                                         ///
48    /// ////////////////////////////////////////////////////////////////
49
50    cout << endl << "Total number of objects: " << N << endl;
51    cout << endl << "Number of useful objects: " << K << endl;
52
53
54    /// ////////////////////////////////////////////////////////////////
55    ///          FOR EACH USEFUL CONTOUR DETERMINES ALL MOMENTS       ///
56    /// ////////////////////////////////////////////////////////////////
57
58    vector<Moments> mmt(K);
59
60    /// Keep the following 4 instructionS in mind and read the note below
61    int j = 0;
62    for(int i = 0; i < N; i++)
63        if (contours[i].size() >= 5)
64            mmt[j++] = moments(contours[i], false);
65
66
67    /// ////////////////////////////////////////////////////////////////
68    ///                     NORMALIZED FIRST MOMENTS                   ///
69    /// ////////////////////////////////////////////////////////////////
70
71    // NOTE: Necessary because the class 'Moments' doesn't have
72    // normalized first moments...
73  vector<Point2f> norm1stMoments(K);
```

```
74
75    for (int i = 0; i < K; i++)
76          norm1stMoments[i] = Point2f((mmt[i].m10 / mmt[i].m00), (mmt[i].m01
                / mmt[i].m00));
77
78
79    // NOTE: exactly the same for cycle is repeated twice:
80    // first, to compute K, and a second time to calculate
81    // the moments. In the present implementation this is
82    // necessary because the value K is used to declare
83    // 'mmt'. The most elegant (and probably correct) method
84    // is allocating dynamically 'mmnts'.
85
86
87      /// ////////////////////////////////////////////////////////////
88      ///          SCAN THROUGH COORDINATES OF EACH CONTOUR           ///
89      /// ////////////////////////////////////////////////////////////
90
91      // These are used to form the name of the ROI images.
92      string name = "ROI_";
93    char numIm[21];
94    sprintf(numIm, "%d", k);
95    char numROI[21];
96    string result = "test";
97    int n = 0;
98
99    int X = bin.cols, Y = bin.rows;     // Image sizes
100   int S = 10;                         // Half side of the ROI
101
102
103     // Show which objects have been "seen" from the original image
104     /*
105     IplImage * imgcpy1 = cvCloneImage(IPLimg);
106   IplImage * I1 = cvCreateImage(cvGetSize(imgcpy1), IPL_DEPTH_8U, 3);
107     cvCvtColor(imgcpy1, I1, COLOR_GRAY2RGB);
108     IplImage * imgcpy2 = cvCloneImage(IPLimg);
109     IplImage * I2 = cvCreateImage(cvGetSize(imgcpy2), IPL_DEPTH_8U, 3);
110     cvCvtColor(imgcpy2, I2, COLOR_GRAY2RGB);
111     unsigned short A = 8;
112     */
113     /// ////////////////
114
115   string Rname;
116
117   for (int i = 0; i < K; i++) {
118         // Discard ROIs that fall out of the image
119         if(norm1stMoments[i].x - S >= 0 && norm1stMoments[i].y - S >= 0 &&
                norm1stMoments[i].x + S < X && norm1stMoments[i].y + S < Y) {
120
121
122       //cvCircle(I1, cvPoint(norm1stMoments[i].x,norm1stMoments[i].y), A,
              cvScalar(255,0,255), 1, 4);
```

```
123         /// ////////////////////
124
125         // Definition of ROI
126               Rect rect(norm1stMoments[i].x - S, norm1stMoments[i].y - S, 2
                      * S, 2 * S);
127
128               // Creation of ROI image with reasonable size
129               Mat Roi, RoiCpy;
130               resize(img(rect), Roi, Size(400, 400), 1, 1, INTER_LANCZOS4);
131
132
133               /// - CHECKS: BRIGHTNESS, MULTIPLICITY
134               /// - CREATE IMAGES ONLY FOR THOSE ROIs THAT PASS THE CHECKS
135
136               if (!check(Roi)) {
137                   n++;
138                   Roi.release();
139                   continue;
140               }
141
142               //cvCircle(I2, cvPoint(norm1stMoments[i].x,norm1stMoments[i].y
                      ), A, cvScalar(0,0,255), 1, 4);
143
144         // Ensure a coherent numeration
145         // (i.e. Use 'n' instead of 'i' as part of the name)
146         sprintf(numROI, "%d", n);
147
148         double EV = analysis(Roi);
149
150         double R = sqrt(pow((norm1stMoments[i].x - xc), 2) + pow((
                  norm1stMoments[i].y - yc), 2));
151
152         Rname = name + numIm + "_" + numROI;
153
154         fileout << setfill(' ') << setw(16) << left << Rname;
155         fileout << " \t" << setw(7) << std::internal << R << "\t\t \t";
156         fileout << EV << "\t\t";
157         fileout << norm1stMoments[i] << "\n";
158
159         //imwrite(Rname, Roi);
160         Roi.release();
161
162         n++;
163           }
164           //else
165               //cout << "Image # " << k << ":   Discarded:  " <<
                      norm1stMoments[i] << endl;
166     }
167
168       /*string check1 = "Check";
169       string check2 = "Check_Result";
170       string checkName1 = check1 + numIm + sufix;
```

```
171   string checkName2 = check2 + numIm + sufix;
172
173   cvSaveImage(checkName1.c_str(), I1);
174   cvReleaseImage(&imgcpy1);
175   cvReleaseImage(&I1);
176
177   cvSaveImage(checkName2.c_str(), I2);
178   cvReleaseImage(&imgcpy2);
179   cvReleaseImage(&I2);
180   */
181     /// ////////////////////////
182
183   bin.release();
184   img.release();
185   bincpy.release();
186 }
```

```
1  bool check(Mat img) {
2      int hist[256];
3
4      // FAINTNESS
5      IplImage * I = cvCreateImage(cvSize(img.cols, img.rows), IPL_DEPTH_8U,
           1);
6      IplImage Itemp = img;
7      cvCopy(&Itemp, I);
8      int c = findContrast(I, hist);
9
10     if(c <= 10){
11         cvReleaseImage(&I);
12         return false;
13     }
14
15     // MULTIPLICITY
16
17     IplImage * cpy = cvCloneImage(I);
18
19     // Opening: 12 iterations
20     cvMorphologyEx(cpy, cpy, NULL, NULL, CV_MOP_OPEN, 12);
21     // Erosion: 10 iterations
22     cvErode(cpy, cpy, NULL, 10);
23     // Dilation: 3 iterations
24     cvDilate(cpy, cpy, NULL, 3);
25
26
27     calcHistogram(cpy, hist);
28     int B = findBackground(hist, cpy->height * cpy->width);
29
30     int t = 10;
31     int T;
32     IplImage * bin = NULL;
33
```

```
    while(t <= 20) {
        T = t + B;
        bin = simpleThresholding(cpy, T);
        if(multi(bin, hist)) {
            cvReleaseImage(&I);
            cvReleaseImage(&cpy);
            cvReleaseImage(&bin);
            return false;
        }
        t += 4;
    }
    return true;
}
```

```
bool multi(IplImage * img, int * hist) {
    vector<vector<Point> > segm;
    Mat imgM = cvarrToMat(img);
    findContours(imgM, segm, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point
       (0, 0));
    imgM.release();

    if(segm.size() > 1)
        return true;
    else
        return false;
}
```

```
int findContrast(IplImage * img, int hist []) {

    calcHistogram(img, hist);

    int cnt = 0, TOT = (img->height * img->width);
    int p5, p95;
    bool f1 = true, f2 = true;

    for(int i = 0; i < 256; i++) {
        cnt += hist[i];
        if(f1 && ((cnt * 100) / TOT) >= 5) {
            p5 = i;
            f1 = false;
        }

        if(f2 && ((cnt * 100) / TOT) >= 95) {
            p95 = i;
            f2 = false;
        }
    }

    return (p95 - p5);
```

```
}
```

A.6 PCA process

A.6.1 Flow Chart

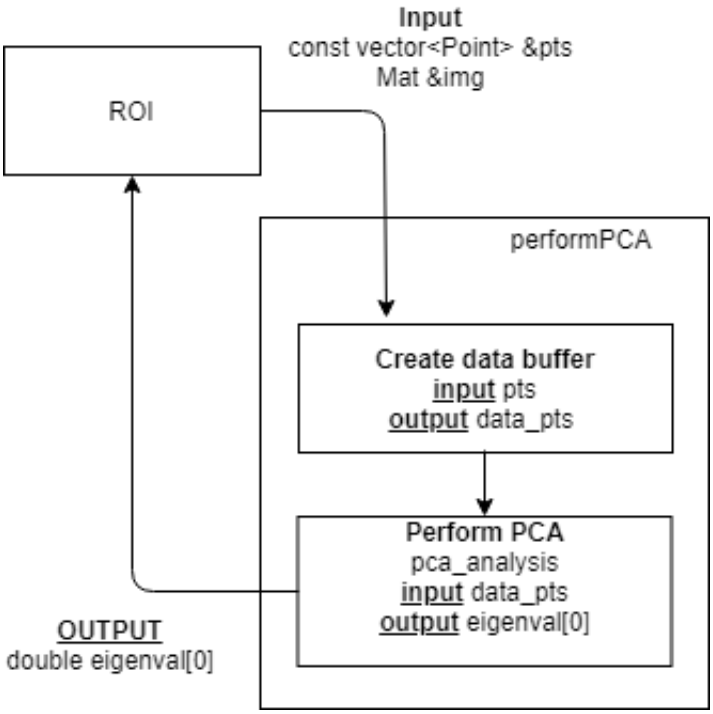


Figure A.4: `pca` function.

A.6.2 Source Code

```
double performPCA(const vector<Point> &pts, Mat &img) {
  //Construct matrix for data
  int size = static_cast<int>(pts.size());
  Mat data_pts = Mat(size, 2, CV_64F);

    //Reshape matrix
  for (int i = 0; i < data_pts.rows; i++) {
    data_pts.at<double>(i, 0) = pts[i].x;
    data_pts.at<double>(i, 1) = pts[i].y;
  }

  //Perform PCA analysis
```

```
  PCA pca_analysis(data_pts, Mat(), PCA::DATA_AS_ROW);

  ofstream myfile;
  myfile.open("data.txt", ios_base::app);
  myfile << "Eigenval " << pca_analysis.eigenvalues.at<double>(0) << endl;
  myfile << "Eigenval " << pca_analysis.eigenvalues.at<double>(1) << endl;
  myfile.close();

  return pca_analysis.eigenvalues.at<double>(0);
}
```

```
bool forsort(const vector<Point>& c1, const vector<Point>& c2) {
    return (contourArea(c1, false) < contourArea(c2, false));
}
```

```
double analysis(Mat gray) {
    Mat binary;
  threshold(gray, binary, 0, 255, THRESH_BINARY | THRESH_OTSU);

  //Contours in binary image
  vector<vector<Point> > contours;
  findContours(binary, contours, RETR_LIST, CHAIN_APPROX_NONE);

  //Sort contour in ascending order
  stable_sort(contours.begin(), contours.end(), forsort);

  //PCA part
  //If only 1 contour simple take the last (largest)
  double EV;
  if(contours.size() < 2)
    EV = performPCA(contours[contours.size() - 1], gray);
  //If 2 contours take 2 last ones
  else
    for (int i = 1; i < 3; i++)
      EV = performPCA(contours[contours.size() - i], gray);

    binary.release();
  return EV;
}
```

A.7 Other Functions

Here it follows a list of functions defined by the authors that have been used in different parts of the project.

```
// Support function used by calcHistogram
void func(uchar c, int * h) {
    unsigned x = c;
    (*(h + x))++;
}


void calcHistogram(IplImage * gray, int hist []) {
        /// Initialize histograms
    for (int i = 0; i < 256; i++)
        hist[i] = 0;

        /// Calculate histogram
    for(int i = 0; i < gray->height; i++) {
        char * ptr = gray->imageData + i*gray->widthStep;
        for(int j = 0; j < gray->width; j++) {
                func((uchar)(*ptr), hist);
                ptr++;
        }
    }
}
```

```
typedef struct THR {
    int B;
    int x;
} THR;

int findBackground(int hist[], int tot) {
    bool fmin = true, fmax = true;
    int m, M;

    for (int i = 0; fmin || fmax; i++) {
        if (fmin)
            if(hist[i]){
                m = i;
                fmin = false;
            }
        if (fmax)
            if(hist[255 - i]){
                M = 255 - i;
                fmax = false;
            }
    }

    THR res_t;
    res_t.B = m + (M - m) / 16;

    int frac = 45, cnt = 0;

    for(int i = m; i <= res_t.B; i++)
```

```
        cnt += hist[i];

    if( (res_t.x = ((cnt * 100) / tot)) == frac)
        return res_t.B;
    else if (res_t.x < frac) {
        while(res_t.x < frac) {
            res_t.B++;
            cnt += hist[res_t.B];
            res_t.x = (cnt * 100) / tot;
        }
        THR t1, t2;
        t1.B = res_t.B;
        t1.x = res_t.x;
        t2.B = res_t.B - 1;
        t2.x = ((cnt - hist[t2.B])* 100) / tot;

        if (abs(t2.x - frac) < abs(t1.x - frac))
            return t2.B;
        else
            return t1.B;
    }

    else { // x > frac
        while(res_t.x > frac) {
            res_t.B--;
            cnt -= hist[res_t.B];
            res_t.x = (cnt * 100) / tot;
        }
        THR t1, t2;
        t1.B = res_t.B;
        t1.x = res_t.x;
        t2.B = res_t.B + 1;
        t2.x = ((cnt + hist[t2.B])* 100) / tot;
        if (abs(t2.x - frac) < abs(t1.x - frac) )
            return t2.B;
        else
            return t1.B;
    }
}
```

```
IplImage * simpleThresholding(IplImage * img, unsigned th) {
    IplImage * bin = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
    for(int i = 0; i < img->height; i++) {
        char * ptr = img->imageData + i*img->widthStep;
        char * p = bin->imageData + i*bin->widthStep;
        for(int j = 0; j < img->width; j++) {
                if ((uchar)(*ptr) >= (uchar)th)
                    *p = 255;
                else
                    *p = 0;
            ptr++;
```

```
12                 p++;
13             }
14         }
15         return bin;
16 }
```

APPENDIX B

Calibration

The following source codes are shown for completeness but will not discussed in detail. Efficiency was not a priority, so functions are clumsy and often recycled versions of previous work.

B.0.1 Faintness

```
1
2  #include <iostream>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string>
6  #include <fstream>
7  #include <vector>
8  #include <opencv2/opencv.hpp>
9
10 using namespace std;
11 using namespace cv;
12
13 int findContrast(IplImage *, int *, int);
14
15 int main(int argc, char ** argv) {
16     int dep = 256;
17     vector <int> C;
18     string line;
19     ifstream out(argv[1]);
20
21     int tot = 0, cnt = 0;
22
23     while(getline(out, line)) {
24         const char * imgName = line.c_str();
25         IplImage * img = cvLoadImage(imgName, CV_LOAD_IMAGE_GRAYSCALE);
26         int hist[dep];
27         int c = findContrast(img, hist, cnt);
28         C.push_back(c);
29         tot += c;
30         cnt++;
31         cvReleaseImage(&img);
32         cout << cnt << endl;
33     }
34
35     cout << cnt << " images..." << endl;
```

```
36
37      int M = tot / cnt;
38      cout << "average contrast: " << M << endl;
39
40      double SD = 0;
41
42      for (int i = 0; i < C.size(); i++)
43          SD += pow(C[i] - M, 2);
44
45      SD = sqrt(SD / C.size());
46
47      cout << "Standard Deviation: " << SD << endl;
48
49      out.close();
50
51      return 0;
52  }
```

```
1
2   void func(uchar c, int * h) {
3       unsigned x = c;
4       (*(h + x))++;
5   }
6
7
8   int findContrast(IplImage * gray, int hist [], int l) {
9       int st = 4;
10      int dep = 256;
11
12      for (int i = 0; i < dep; i++)
13          hist[i] = 0;
14
15      for(int i = 0; i < gray->height; i++) {
16          char * ptr = gray->imageData + i*gray->widthStep;
17          for(int j = 0; j < gray->width; j++) {
18                  func((uchar)(*ptr), hist);
19                  ptr++;
20          }
21      }
22
23      int cnt = 0, TOT = (gray->height * gray->width);
24      int p5, p95;
25      bool f1 = true, f2 = true;
26      for(int i = 0; i < 256; i++) {
27          cnt += hist[i];
28          if(f1 && ((cnt * 100) / TOT) >= 5) {
29              p5 = i;
30              f1 = false;
31          }
32
33          if(f2 && ((cnt * 100) / TOT) >= 95) {
```

```
34              p95 = i;
35              f2 = false;
36          }
37
38      }
39
40      return (p95 - p5);
41  }
```

B.0.2 Multiple Segments

```

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;

void preprocessing (const char *);
void thresholding (const char *);
void multi(const char *);
int findBackground(int *, int);
void drawHistogram(IplImage *, int *);
IplImage * simpleThresholding(IplImage *, int);
IplImage * getBinImage(IplImage *, unsigned);

int main(int argc, char ** argv) {
    const char * imgName = argv[1];
    preprocessing(imgName);
    return 0;
}
```

```
void preprocessing(const char * imgName) {
    IplImage * img = cvLoadImage(imgName, CV_LOAD_IMAGE_GRAYSCALE);

    IplImage * CL = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1);
    cvSet(CL, 0);

    int IT = 12;

    cvMorphologyEx(img, CL, NULL, NULL, CV_MOP_OPEN, IT);
    cvErode(CL, CL, NULL, 10);
    cvDilate(CL, CL, NULL, 3);

```

```
    cvSaveImage("temp.png", CL);
    thresholding("temp.png");

    cvDestroyAllWindows();

    cvReleaseImage(&CL);
}
```

```
void thresholding(const char * imgName) {
    IplImage * img = cvLoadImage(imgName, CV_LOAD_IMAGE_GRAYSCALE);

    int dep = 256;
    int hist[dep];
    drawHistogram(img, hist);

    int B = findBackground(hist, img->height * img->width);

    int t = 10, T;
    IplImage * bin;
    char imNum[2];
    char c;
    int i = 0;

    while(t <= 20) {
        char imName[30] = "";
        T = t + B;
        bin = simpleThresholding(img, T);
        strcat(imName, "bin");
        imNum[0] = 49 + i;
        imNum[1] = '\0';
        strcat(imName, imNum);
        strcat(imName, ".png");
        cvSaveImage(imName, bin);
        multi(imName);
        i++;
        t += 4;
    }

    cvReleaseImage(&img);
    cvReleaseImage(&bin);

}
```

```
void multi(const char * imgName) {
    Mat img;
    // Already processed externally
    img = imread(imgName, CV_LOAD_IMAGE_GRAYSCALE);

    vector<vector<Point> > segm;
```

```
7     findContours(img, segm, CV_RETR_TREE, CV_CHAIN_APPROX_NONE, Point(0,
          0));
8     cout << endl << "objects: "<< segm.size();
9
10    if(segm.size() > 1) {
11        cout << " -> multiple objects in the same ROI!" << endl;
12    }
13    img.release();
14 }
```

APPENDIX C

Matlab code

C.1 Visualization

```
%%% Reads in text file that is produced by Startracker.exe .
%%% File has to be be prepared for reading in.
%%% Remove coma values ( ',' ) as well as brackets ( '[ ]' ) from it.
%%% This can easily be done by using notepad replace all option.
%%% ROI_name|Distance from center|Eigenvalue|X coordinate|Y coordinate

close all;
%% Cam 1
fileID = fopen('CAM1.txt');
C1 = textscan(fileID,'%s %f32 %f32 %f32 %f32');
fclose(fileID);
whos C1
Mc = mean(C1{1,3});

figure
scatter(C1{1,2},(C1{1,3})/580,'filled'),
set(gca, 'YScale', 'log')
xlabel('Distance [px]'), ylabel('Magnitutde')
title('Distance to Magnitutde cam 1')
saveas(gcf,'cam1.png')

figure
scatter3(C1{1,4},C1{1,5},C1{1,3}/580,'filled')
set(gca, 'ZScale', 'log')
xlabel('X coordinate'), ylabel('Y coordiante'), zlabel('Magnitutde')
title('Position of star to magnitutde cam 1')
saveas(gcf,'cam1_3d.png')
%% Cam 2
fileID = fopen('CAM2.txt');
C2 = textscan(fileID,'%s %f32 %f32 %f32 %f32 %f32');
fclose(fileID);
whos C2
Mc2 = mean(C2{1,3});

figure
scatter(C2{1,2},(C2{1,3})/580,'filled'),
set(gca, 'YScale', 'log')
xlabel('Distance [px]'), ylabel('Magnitutde')
title('Distance to Magnitutde cam 2')
saveas(gcf,'cam2.png')
```

```
41
42 figure
43 scatter3(C2{1,4},C2{1,5},C2{1,3}/580,'filled')
44 set(gca, 'ZScale', 'log')
45 xlabel('X coordinate'), ylabel('Y coordiante'), zlabel('Magnitutde')
46 title('Position of star to magnitutde cam 2')
47 saveas(gcf,'cam2_3d.png')
48 %% Cam 3
49 fileID = fopen('CAMunf.txt');
50 Cu = textscan(fileID,'%s %f32 %f32 %f32 %f32 %f32');
51 fclose(fileID);
52 whos C2
53 Mcu = mean(Cu{1,3});
54
55 figure
56 scatter(Cu{1,2},Cu{1,3}/580,'filled'),
57 set(gca, 'YScale', 'log')
58 xlabel('Distance [px]'), ylabel('Magnitutde')
59 title('Distance to Magnitutde cam unf')
60 saveas(gcf,'camunf.png')
61
62 figure
63 scatter3(Cu{1,4},Cu{1,5},Cu{1,3}/580,'filled')
64 set(gca, 'ZScale', 'log')
65 xlabel('X coordinate'), ylabel('Y coordiante'), zlabel('Magnitutde')
66 title('Position of star to magnitutde cam unf')
67 saveas(gcf,'camunf3d.png')
```

APPENDIX D

Graphs

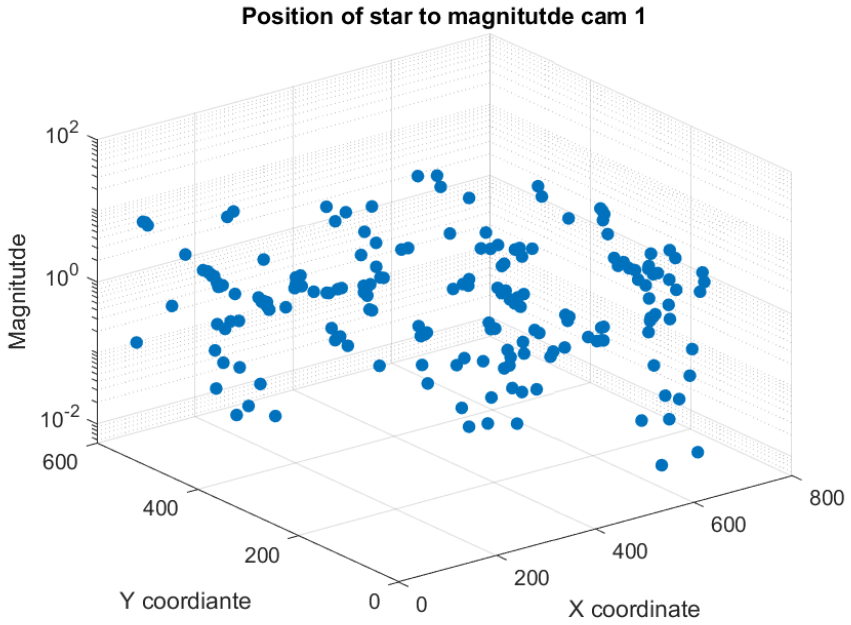


Figure D.1: 3D scatter plot, x coordinate, y coordinate, magnitude per pixel

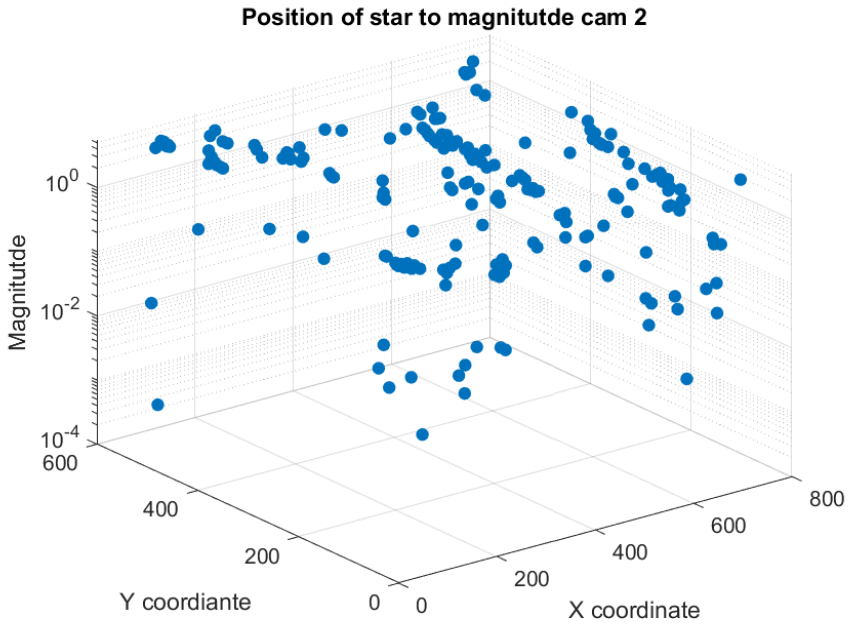


Figure D.2: 3D scatter plot, x coordinate, y coordinate, magnitude per pixel

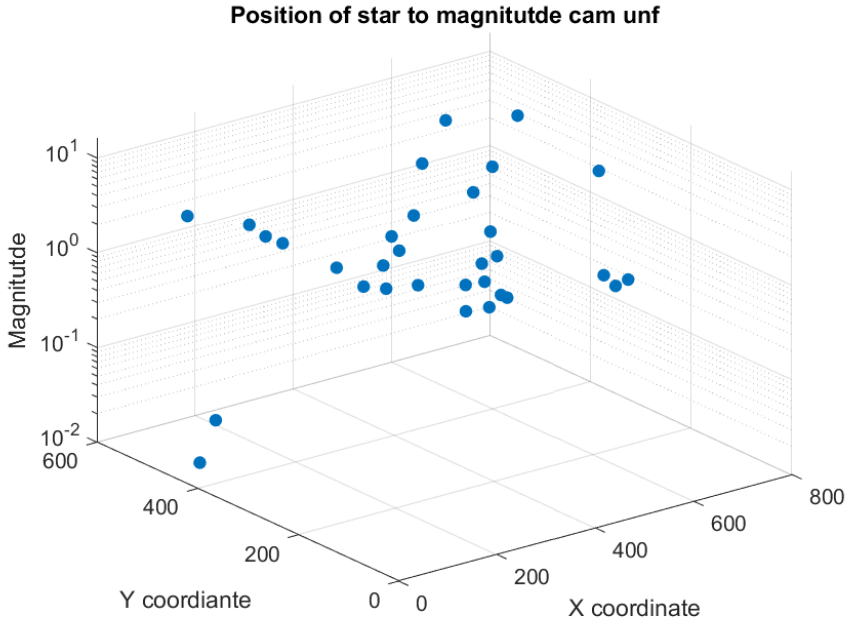


Figure D.3: 3D scatter plot, x coordinate, y coordinate, magnitude per pixel