NTNU – Trondheim
Norwegian University of
Science and Technology

A Hyperspectral Imaging System using an Acousto-Optic Tunable Filter

Constructing and evaluating the
hyperspectral imaging system

Alejandro Baranda Castrillo

Master of Science in Electronics
Submission date: July 2015
Supervisor: Lise Lyngsnes Randeberg, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

NTNU – Trondheim
Norwegian University of
Science and Technology

A Hyperspectral Imaging System using an Acousto-Optic Tunable Filter

Alejandro Baranda Castrillo

June 2015

MASTER THESIS

Department of Electronics and Telecommunications

Norwegian University of Science and Technology

Supervisor 1: Professor Lise Lyngsnes Randeberg

Supervisor 2: Ph. D. Matija Milanic

Preface

This document is the result of my Master Thesis work, carried out at the Norwegian University of Science and Technology (NTNU) during both the fall and spring semesters of the academic year 2014/2015. It is done according to the Learning Agreement signed between NTNU and the University of Valladolid (Spain) regarding my Erasmus+ exchange program.

The project was proposed by Lisa L. Randeberg, who is the Supervisor, as part of the work corresponding to the Department of Electronics and Telecommunications.

I wish to express my sincere thanks and gratitud to Matija Milanic, Ph. D., for providing excelent guidance, help and encouragement on every stage of this work. Also, I would like to thank Lise L. Randeberg for her leading work as Supervisor. Thanks to Professor Amund Skavhaug and Senior Engineer Terje Mathiesen for their advice, and to Professor Lars Lundheim for his tuition to choose one among the many projects available.

Trondheim, 2015-06-18

[illegible]

Alejandro Baranda

Summary

The aim of this work was to built a high performance hyperspectral imaging system, a state-of-the-art technology with applications among many fields, like medicine. Chapter 1 introduces this technology and its benefits. In Chapter 2, we have a look at the physical theory behind the technology, the elements needed to build such a system and the problematic introduced by them. Chapter 3 deals with the actual design of the system. Reasons supporting the election of one model of camera available in the market over another are presented (3.1). Also, the main features for all the final elements of the system are commented. A closer look to the system from an optical point of view is also included (3.3). Description of the software part of the system is the main topic in Chapter 4. First, a user interface developed within this work is presented (4.1). Then, the synchronization issue is treated (4.2). Two solutions are described, although none of them could be fully implemented, due to problems with the RF driver. Chapter 5 presents three tests that should be carried out with the final system, to have a better knowledge of its performance (5.1 and 5.3), and to improve it (5.2). Results of two additional tests performed with a borrowed hyperspectral imaging system are shown in Chapter 5 (5.4 and 5.5). Finally, Chapter 6 provides some pointers regarding future work and improvements of the system.

Contents

Preface i

Summary ii

1 Introduction **2**

2 Background Theory **4**

2.1 Hyperspectral imaging 4

2.1.1 Spectral reflectance 6

2.1.2 Image cube 7

2.2 Camera 8

2.2.1 sCMOS sensor 8

2.2.2 Global shuttering mode 9

2.3 Acousto-optical tunable filter (AOTF) 10

2.3.1 Polarization 13

2.3.2 Chromatic aberrations 14

2.4 Optical fundamentals of the system 15

2.4.1 Radial distortion 20

2.5 Matlab User Interface 21

2.5.1 Interfacing with other languages 21

2.5.2 Portability 22

3 Building AOTF system **23**

3.1 Andor Zyla sCMOS camera 23

3.1.1 Camera selection 23

3.1.2 sCMOS sensor 27
3.2 Acousto optic tunable filter (AOTF) 30
3.2.1 RF driver 31
3.3 Optical design of the system 33
3.3.1 Lenses 35
3.3.2 Polarizers 36

4 Software **38**
4.1 Graphical User Interface 39
4.2 Synchronization 50
4.2.1 Software triggering 50
4.2.2 Hardware triggering 52

5 System characterization and experiments **56**
5.1 Spectral analysis 56
5.2 Power optimization 57
5.3 Wavelength dependent resolution 59
5.4 Grid aberrations 59
5.5 Color chart spectra 62

6 Discussion **68**

7 Conclusion **71**

A Graphic User Interface: code **72**
A.1 Matlab code 72
A.1.1 userInterface 73
A.1.2 cooling 96
A.1.3 finDisplay 97
A.1.4 showImages 98
A.1.5 timeString 98
A.2 Mex files 100
A.2.1 mOpenCamera 100

A.2.2 mCloseCamera 103
A.2.3 mGetHSISoftware 104
A.2.4 mSetLiveView 113
A.2.5 mGetFrameLiveView 116
A.2.6 mStopLiveView 118
A.2.7 mModifiedCloseCamera 120
A.2.8 mCooling 121
A.2.9 mExposureTime 124

B MEX files and Andor SDK **126**
B.1 MEX-Files 126
B.2 Andor SDK 130

C Starting point: RF driver communication **134**
C.1 Matlab AOTF 134
C.2 C source files 141

D Matlab code for testing **146**
D.1 Grid 146
D.2 Color chart 151

Bibliography **155**

Curriculum Vitae **158**

Chapter 1

Introduction

Hyperspectral imaging has become a powerfull tool for scientific and industrial analysis in many different fields, its applications go all from agriculture or air surveillance to medical diagnosis. A hyperspectral image contains information about how light interacts with matter. Allowing that some properties of the target object, such as texture or composition, can be infered. Potential applications are classification, detection or analysis of objects.

Spectroscopy is the study of light that is emitted by or reflected from materials and its variation in energy with wavelength. A lot of types of radiated energy can be used for spectroscopic measurements, but electromagnetic spectrum is the most widely used among them. Imaging spectroscopy combines the spectral information with spatial information. The resulted data forms a stack of monochrome images taken at different wavelengths which is sometimes known as an 'image cube'. The number of image planes building the image cube varies and it is an important parameter. If many narrow contiguous spectral bands are captured, it can be assumed that fully spectrum information is had for every image point. Ideally, spectrum information is accurately sampled and it can be seen as a continuous signal. In such a case, the technique is called hyperspectral imaging, instead of multispectral imaging.

Several spectral imaging technologies have been developed for acquiring the three dimensional (x, y, λ) matrix of a hyperspectral cube. Spatial scanning systems use a filter or a monochromator and a so-called push broom system, where only one line of the image is scanned at a time. The spatial dimension is collected through platform movement or scanning, this requires accurate pointing information to build the image. Spectral scanning systems, however, acquire a full spatial image at each spectral band. To achieve this, a band-pass optical filter is placed before the sensor, selecting a narrow spectral band at a time. Spectral smearing can occur if there is movement within the scene, invalidating spectral detection. In such wavelength scanning systems, the optical filter must be tunable. This can be achieved in several ways: mechanically, by using filter wheels, or with electronically tunable filters. For stationary applications, spectral scanning imagers are a natural choice, while for systems moving in one direction relative to the target scene, like airborne surveillance, push broom imagers seem to be a better option.

The purpose of this study is to design a hyperspectral imaging system that implements spatial scanning by using an acousto-optic tunable filter (AOTF) to select the spectral narrow bands. A state-of-the-art sCMOS scientific digital camera is used to acquire the images. System parameters such like spatial and spectral resolution, framerate, optical characteristics will be evaluated. Synchronization between camera and AOTF and timing issues should be addressed in a way that optimizes the mentioned parameters. A fully working MATLAB® user interface will be provided.

Chapter 2

Background Theory

In this chapter, the theory behind hyperspectral imaging systems is presented and the functions that each component in the system must accomplish are described. First, some concepts about hyperspectral imaging and its use will be introduced, then we have a look to the camera properties and explain the fundaments behind the filter technology. Finally, the optical foundations of the system are described.

2.1 Hyperspectral imaging

Contrary to color photography, in which each pixel contains spectral information from three bands: red, green and blue, each pixel in a hyperspectral image is made up of light intensity data from many different narrow bands. Depending on the number and width of the spectral bands the spectrum is divided into, it may be refered to as multispectral or hyperspectral imaging. The latter requires at least around one hundred images, if full spectum data are taken; or a dense concentration of bands, for example, twenty 10 nm wide bands from 500 to 700 nm. A comparison between hyperspectral and multispectral imaging is shown in Figure 2.1. It can be seen that multispectral imaging does not provide continuous spectral information, since it uses fewer and wider spectral bands. The spectral data provides information about the material that has been imaged. This is possible because of the signature that matter prints on the light spectrum when light and matter interact.

Spectral *signature* is the specific combination of emitted, reflected and absorbed electro-

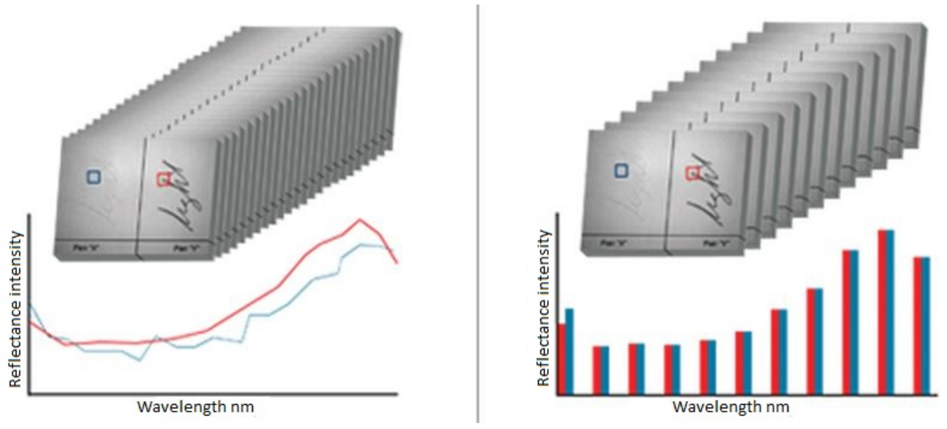


Figure 2.1: Image cube and spectrum for hyperspectral and multispectral images, respectively.

magnetic radiation at varying wavelengths which can uniquely identify an object. Every material can potentially be detected by spectroscopy. Actual detection depends on the spectral resolution, signal-to-noise ratio of the system, spectral coverage as well as other factors like the amount of the material present in the scene and the strength of the absorption/reflectance features for the material in the spectrum region analysed.

The HSI technology was initially developed for remote sensing and military applications. However, it also finds use within, e. g., geology (M. Govender and Bulcock, 2007), food quality measurements (Gowen, 2007) and medicine. Different medical applications have emerged during the last years as the technology has become more available, e.g. monitoring tumor hypoxia (B. S. Sorg and Cao, 2005), and cancer detection using fluorescent techniques (M. E. Martin and T. Vo-Dinh, 2006). Advanced hyperspectral microscopic techniques have also been developed (P. De Beule and French, 2007) (W. F. J. Vermaas and D. M. Haaland, 2008) (M. B. Sinclair and Jones, 2006).

An HSI is mainly composed of the light source, wavelength dispersion devices and area detectors. HSI systems fundamental classification is based on the acquisition mode, i.e., how spectral and spatial information is acquired (Sellar and Boreman, 2005). There are two main scanning methods: spatial scanning and spectral scanning. Spatial scanning methods genereate hyperspectral images by acquiring a complete spectrum for each pixel in the case of a whiskbroom (point-scanning) instruments, or line of pixels in pushbroom (line-scanning) instruments, and then spatially scanning throughout the scene. On the other hand, spectral scanning methods, also called staring or area-scanning imaging, are based on capturing the whole scene with 2-

D detector arrays in a single exposure and then stepping through wavelengths to complete the data cube.

The wavelength dispersion devices are the core element of a HSI system. There are many types of optical and electro-optical dispersive devices which can perform spectral selection or dispersion in HSI systems. The mainly dispersive devices used in the literature are: monochromators, which are the key component of pushbroom HSI systems, they may be prisms or diffraction gratings; optical bandpass filters, either tunable, like liquid crystal tunable filters or acousto-optic tunable filters, or fixed, such as filter wheels; and single-shot imagers, such as computer-generated holograms that enables to capture both spatial and spectral information in a single frame.

The environment the scene is placed in influences the spectral information. A good characterization of the light source emission spectrum is important. This allows its effect to be removed computationally, which increases the quality of the data. Otherwise, the information is corrupted. The solar emission spectrum is well known, however it includes some random effects mainly due to atmospheric variations, such as atmospheric scattering or absorption. These random effects are difficult to be characterized, making the sun a difficult light source to deal with. Its use is inevitable for outdoor applications, but more suitable light sources may be chosen for indoor applications. Even using light sources with a quite flat, steady and well known spectrum properties there are still local effects, for example those due to shadowing, corrupting the spectra in a way that can not be completely removed with image processing. Characterization of illumination systems is treated in Jaka Katrasnik and Likar (2013).

Halogen lights are one of the few available light sources to be used for near-infrared hyperspectral imaging as they emit light in the visible and in the shortwave infrared range and do not have any sharp spectral peaks. Mercury lamps show a less flat spectrum compared to that of LEDs, which are by far the most widely used light source in HSI systems.

2.1.1 Spectral reflectance

It has already been mentioned how materials print a spectral signature on the light they reflect. In order to get this signature, a parameter called *spectral reflectance* is of interest. It represents the ratio between reflected and incident light, as a function of wavelength. This dependence is due to the fact that light is scattered or absorbed to different degrees at certain wavelengths, and it exists for almost every material.

There are several physical processes involved that determine the nature of the reflected light, and thus, the spectral signature of the material. In the first place, almost every object shows some degree of specular reflection, which means that some of the light rebounds directly on the surface of the material, as on a mirror. In this case, the spectrum of the reflected light remains the same as that of the incident light. There is no signature printed. In the second place, part of the light diffuses into the material where some is absorbed and some is randomly scattered, which is known as diffuse reflection. Finally, fluorescence, which is the emission of light by a substance that has absorbed light or other electromagnetic radiation, may also take place. In this case, a photon at shorter wavelength is absorbed and a photon at longer wavelength is emitted consequentially.

A white material, for example, does not absorb any wavelength while a colored material will absorb some wavelengths and diffusely reflect others. These reflected wavelengths are responsible for the color of the material. In a way, they shape the nature of the incident light to create what we perceive as *color*.This effect can be thought from the continuous spectrum point of view: it explains why matter prints a signature on the incident light depending on how different spectral components of light are absorbed or reflected.

2.1.2 Image cube

A hyperspectral image is a three-dimensional matrix made up of many two-dimensional images taken at different wavelengths. Frequency or wavelength (they are related through the phase velocity: $\lambda = v/f$) distance between two adjacent frames should be small enough so that, globally, they provide virtually continuous information of the spectrum. Naturally, this implies that a lot of frames are needed to cover the spectrum, what leads to a large amount of data to be stored and processed. With abundant spatial and spectral information available, advanced image classification methods for hyperspectral datasets are required to extract, unmix and classify relevant spectral information.

Analysis of hyperspectral data is complex in many ways. In particular, it presents a high data redundancy due to high correlation in the adjacent bands. Dimensionality becomes an issue and power processors may be required to perform the required processing in time. Image preprocessing usually involves data normalization, from the camera observation to reflectance or transmittance and filters to smooth the spectral signatures and reduce the noise effect. But the actual processing is much more resources consuming. On a first stage, the most relevant information from the original data must be obtained and represented in a lower-dimensionality space. Then, resources greedy hyperspectral image classification methods are applied.

2.2 Camera

As we are dealing with an imaging system, naturally, the camera is a key element of the system, and its overall performance is tightly related with the camera features. The scientific camera used in this project is based on sCMOS technology. The reasons why this camera was selected and its specifications will be treated in the Section 3.1, while here the main mechanisims making the camera works are explained. Further information about how the camera works can be found at the hardware guide Andor® (2014a).

2.2.1 sCMOS sensor

The Zyla sCMOS camera, used in this project, is a high performance camera which uses sCMOS technology instead of the more established Electron Multiplying CCD (EMCCD). EMCCD is based on a charge-coupled device, widely used for digital imaging, with the addition of a solid state Electron Multiplying register which allows weak signals to be multiplied before any readout noise is added by the output amplifier, hence rendering the read noise negligible. Deeper information about EMCCD technology can be found on the web of the EMCCD forum (web site EMCCD). On the other hand, the sCMOS sensor is based on the CMOS technology, and it is an *active pixel sensor* (APS), whereby each pixel has its own integral amplifier, as it is shown on Figure 2.2. The sequence of operation is as follows (each number points to an element on the Figure 2.2):

1. Light photons hit the sensor and generate charge.

2. The photo-generated charge is converted to an analog voltage inside each pixel amplifier.

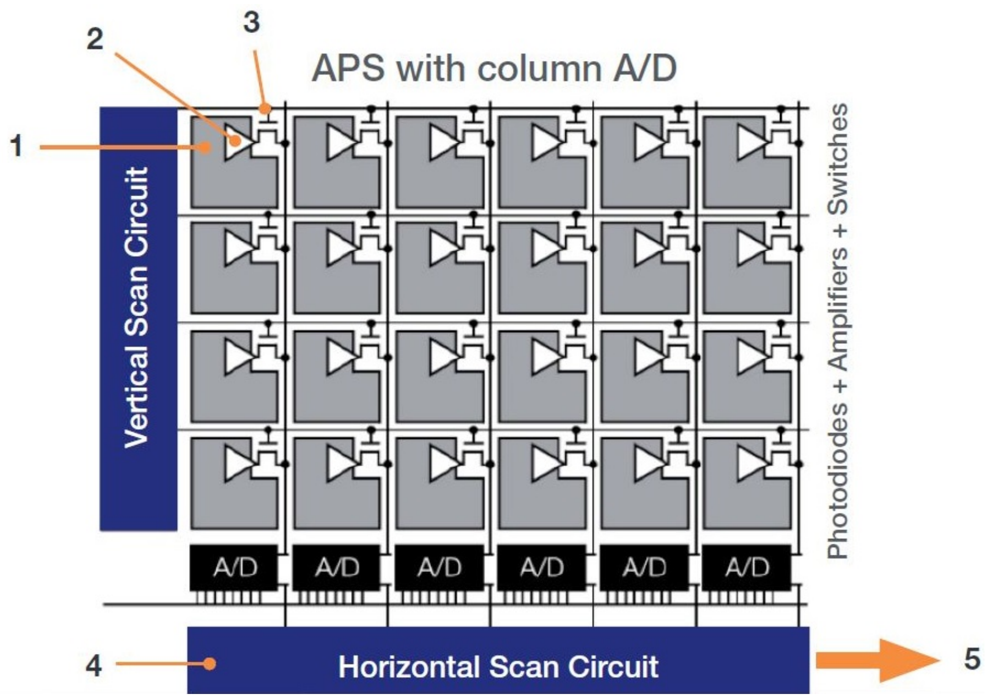


Figure 2.2: sCMOS Sensor Architecture

3. A row-select signal is used to transfer the pixel voltage to the column bus.
4. A set of analog to digital converters (A/D) sample the analog voltage signal for each column.
5. The final digitized signals are then read out sequentially at a pixel readout speed of up to 280 MHz.

2.2.2 Global shuttering mode

Zyla sCMOS camera offers two shuttering modes, Global and Rolling shutter, that can be selected. These modes are different in the way that the image is read off the sCMOS sensor. For rolling shutter mode, charge is transferred from each row sequentially during readout, while in global shutter mode every pixel in the sensor efectively ends the exposure simultaneously.

Lowest noise and fastest frame rates are achieved from rolling shutter mode. Unfortunately, spatial distortion may appear as each row will start and end its exposure slightly offset in time from its neighbour (around 10 μs). From the point of view of readout, the sensor is split in half horizontally and rows are read out from the centre outwards. This implies that rows at the top edge of the sensor start and end their exposure around 10 ms after rows at the centre of the sensor. This time difference makes the rolling shutter mode non-suitable for our application. When imaging human skin, although it seems to be a static target, blood pumped into the veins may introduce movement during the acquisition. This would affect the image if rolling shutter mode is on, in a way that can not be tolerated.

For that reason, Global shutter mode is our choice. It can be thought of as a ’snapshot’ exposure mode: all pixels of the array are exposed simultaneously. Before starting the exposure, charge is drained from all pixels in the array during what is called a *keep clean state*. When the exposure starts, every pixel simultaneously begins to collect charge and keeps doing it until the exposure time is over. Then each pixel transfers charge to its readout node. It must be remarked that the global shutter mode allows to be operated in a continuous *overlap mode*, whereby an exposure can proceed while the previous one is being read out from the readout nodes of each pixel, provinding a 100% sensor duty cycle. It requires higher complexity, but it results in optimal time resolution and photon collection efficiency. Further information is provided on the synchronization section (4.2), as this is the desirable mode to have the system working on.

Global shutter mode main drawback is that it requires that a reference readout is performed *behind the scenes*, in addition to the actual readout of charge from each pixel. Therefore, it halves the maximum frame rate that would be achieved in rolling shutter mode. RMS (root mean square) noise is also increased by a factor of 1.41.

2.3 Acousto-optical tunable filter (AOTF)

An optical filter is a device that only allows light of a certain band of the electromagnetic spectrum to pass through it. Its behaviour can be thought as that of a window which remains open for the wavelengths of interest and close for any other. Therefore, light outside the band of interest is strongly attenuated. Such a behaviour receives the name of *band-pass filter*. To be useful for hyperspectral imaging, the filter must have the ability to *quickly* change the spectral band for which light passes without being attenuated. For that purpose, a signal of some form, that allows to electronically control the spectral transmission characteristics of the device, is of interest.

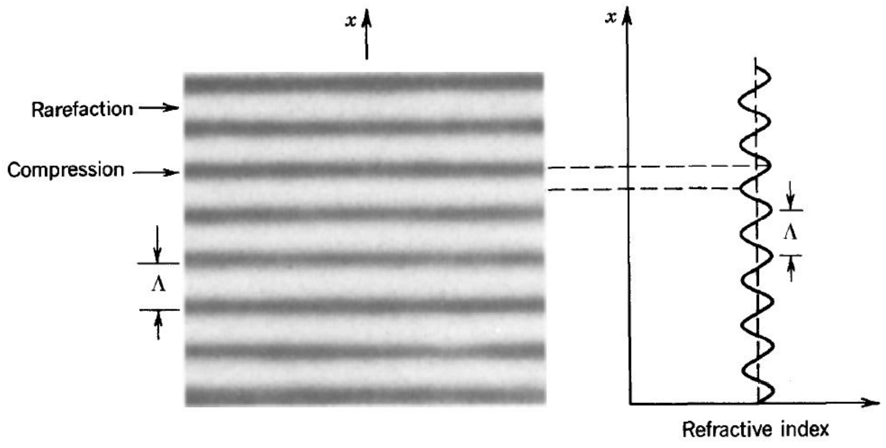


Figure 2.3: Variation of the refractive index accompanying a harmonic sound wave. The pattern has a period Λ, the wavelength of sound, and travels with the velocity of sound.

Different filter technologies could be used to satisfy these requirements, i.e. liquid crystal tunable filters (LCTFs) (Gat, 2000) or rotating wheels (Afromowitz, 1988). However, attention will be focused on acousto-optical tunable filters, since they offer the highest speed (although LCTFs provide better transmitivity). These devices are based on the acousto-optic effect, deeply explained in Bahaa E. A. Saleh (1991), chapter 20; the illustrative figures shown in this section were taken from the mentioned reference. Basically, the acousto-optic effect states that, for being sound a pressure wave, it affects the structure of a material it is travelling through. Specifically, some regions will be compressed inside the material while others remain relaxed. As density varies along the material, so does the optical reffractive index. Thus, the sound wave modifies the effect of the medium on light.

As shown in the Figure 2.3, the frequency variations of the sound wave result in different periodic distributions of relaxed and compressed regions within the crystal. This causes a phenomenon to happen called Bragg difraction, which is the partial reflection of an optical plane wave, due to the periodical parallel variations of the refractive index in a material created by a sound wave. An acoustic plane wave acts as a partial reflector of light (a beamsplitter) when the angle of incidence θ satisfies the Bragg condition,

$$\sin\theta = \frac{\lambda}{2\Lambda} \tag{2.1}$$

where λ is the wavelength of light, Λ is the wavelength of the sound wave and θ is the angle of

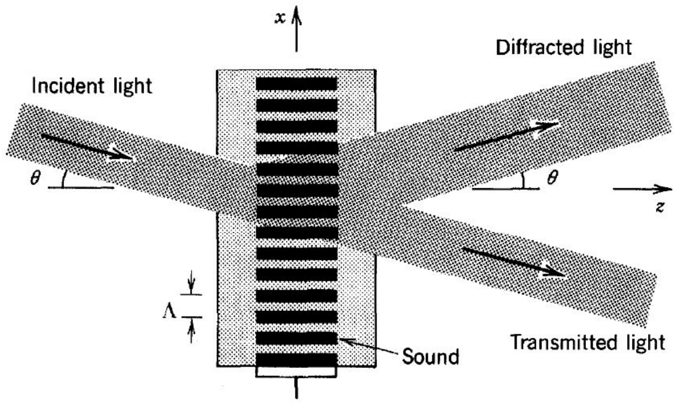


Figure 2.4: Bragg diffraction: an acoustic plane wave acts as a partial reflector of light (a beam-splitter) when the angle of incidence θ satisfies the Bragg condition.

incidence. The Bragg effect is graphically explained in Figure 2.4.

Soundwave creation is a well known process that can be seen in daily life, i.e. how speakers produce sound. A radio frequency (RF) signal is sent through a wire to a piezo-electric transducer which vibrates following the electrical wave and producing sound. This transducer is then attached to a suitable crystal medium in which the sound wave is excited. The presence of the sound wave in the crystal produces that light at certain wavelengths is diffracted when passing through the crystal. The range of wavelengths that are diffracted can be controlled by changing the frequency of the sound wave, which is achieved varying the frequency of the RF signal.

With all these elements in mind, an acousto-optic tunable filter may be built, as described in the scheme shown in Figure 2.5. The acoustic transducer produces a sound wave that interacts with the incoming light producing its diffraction. Incident light beams are diffracted into two orthogonally polarized *first order* beams, labeled as (+) and (-) beams. The undiffracted beam, labeled as *zero order* beam, must be blocked so that only the first order diffracted light reaches the camera sensor. *Higher order* diffracted beams may exist, but they are not of interest as their intensity is lower and their direction differ from that of the camera. The diffracted beams and the undiffracted beam are physically separated, therefore the latter can be blocked by a physical element. Thus, only the diffracted beam, i. e. the filtered light, reaches the camera sensor. For the diffraction process to happen in an optimal way, the angle of incidence of the incoming light beam should be as close to 90º as possible.

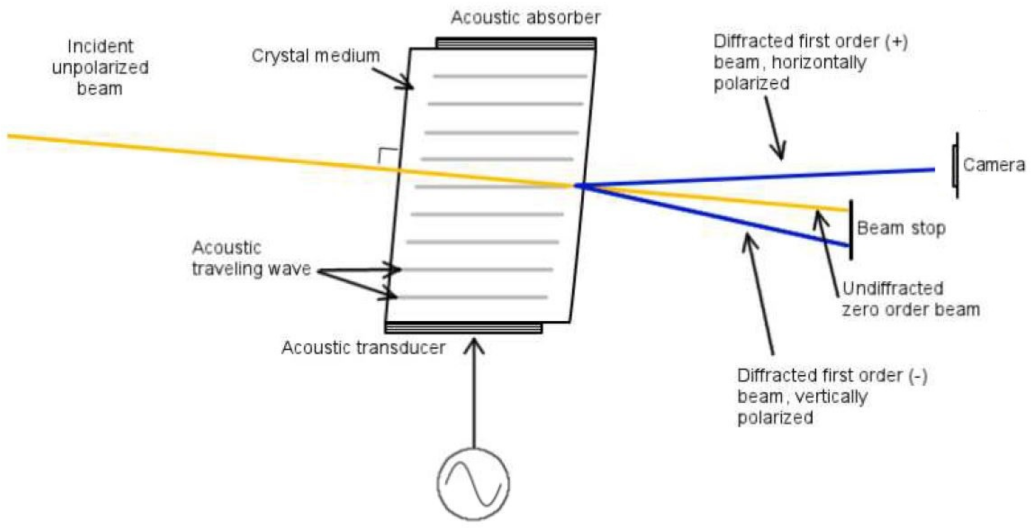

Acoustic absorber
Incident unpolarized beam
Crystal medium
Diffracted first order (+) beam, horizontally polarized
Camera
Beam stop
Acoustic traveling wave
Undiffracted zero order beam
Acoustic transducer
Diffracted first order (-) beam, vertically polarized

Figure 2.5: AOTF structure

2.3.1 Polarization

In the former explanation, polarization has been mentioned as it is a consequence of the Bragg diffraction. Now, its role in actual HSI systems is considered. A light wave that is vibrating in more than one plane is referred to as unpolarized light. Light emitted by common sources, like the sun or a lamp, is unpolarized light. Such light waves are created by electric charges that vibrate in a variety of directions, thus creating an electromagnetic wave that vibrates in a variety of directions. The concept of unpolarized light can be thought as a wave that has an average of half its vibrations in a horizontal plane and half of its vibrations in a vertical plane. On the ohter hand, in polarized light waves the vibrations occur in a single plane. Polarization is the process of transforming unpolarized light into polarized light. It can be achieved in several ways like by transmission, reflection, refraction or scattering of light.

A polarizer is an optical filter that passes light of a specific polarization and blocks waves of other polarizations, as it is shown in Figure 2.6. Such an element can be incorporated to the HSI system, in what is known as cross-polarized configuration, in order to take advantage of the polarizing effect of the AOTF. In such a configuration, a polarizer is placed in the input of the AOTF that only allows vertical polarization of light to pass through. At the output, the beam of interest (it has been refered to as first order (+), as shown in Figure 2.5), is horizontally polarized, while the other beams of light remain vertically polarized. A polarizer can be placed

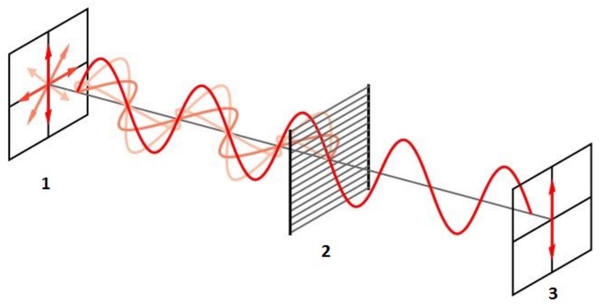


Figure 2.6: Scheme showing the effect a polarizer has in light: 1, unpolarized light is emited, the electromagnetic wave vibrates in many directions; 2, a polarizer removes all the components except for the vertical; 3, only vertically polarized light is observed by the camera sensor.

at the output that only allows horizontal polarization to pass through it, therefore eliminating all the beams except for the beam of interest. See that the polarizer at the input is vertical, while the one at the output is horizontal. For this reason, it is called *cross-polarized* configuration.

2.3.2 Chromatic aberrations

The use of an AOTF to filter light coming from an object before being imaged produces chromatic aberrations to appear in the processed image. Chromatic aberrations may be divided into longitudinal and transverse aberrations, according to V. B. Voloshinov and Yukhnevich (2012), in which novel optical schemes characterized by a low level of transverse and longitudinal aberrations are proposed. Figure 2.7 is an ilustrative scheme showing the effect of the inherent chromatic aberrations. Transverse aberrations are caused by the dependence of the diffraction angle on the light wavelength. They produce the image to suffer a lateral shift in the plane of diffraction at an angle of γ', as shown in Figure 2.7. On the other hand, longitudinal aberrations are caused by a change of the optical path length in the system due to refractive index dispersion of the AO crystal. As a result, the plane of the best image sharpness shifts along the optical axis of the system which causes defocusing of the image. It is represented in Figure 2.7 as a shift in the best image sharpness plane position of δb. Filters with ultrabroad operating range or those which work in a spectral range close to the transmission cutoff of the crystal are more affected by chromatic aberrations. Compensation of the aberrations is critical for those devices.

A frequently used method, for the reduction of transverse aberration, consists of tilting the exit face of the optical filter relative to its entrance face by a certain angle. Of course, as it requires

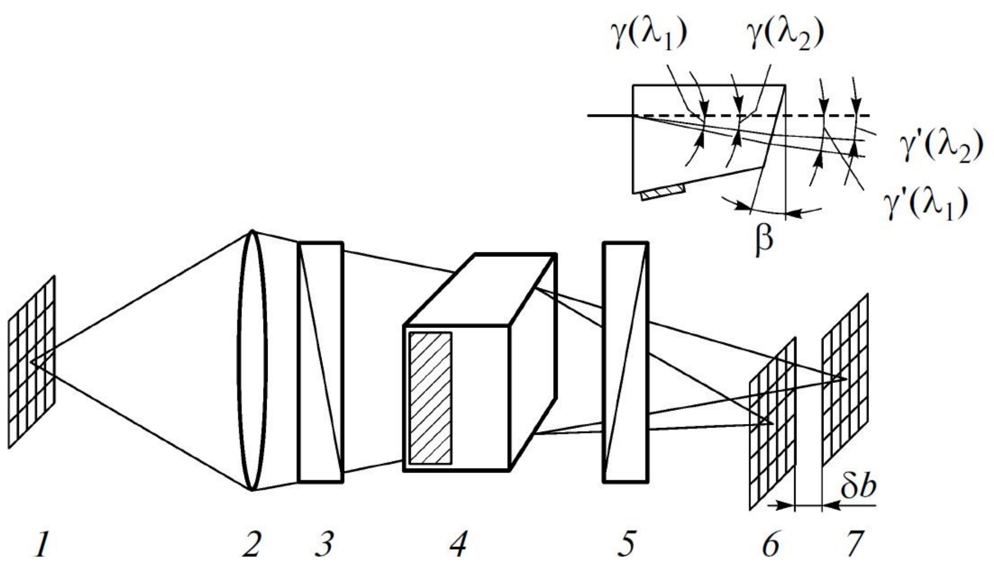


Figure 2.7: The principal optical scheme of an image-processing system with an AO filter: 1, an object with a broadband emission spectrum; 2, objective; 3, polarizer; 4, tunable AO filter; 5 crossed polarizer; 6, the image in the first diffraction order at the wavelength of λ_1; 7, the image in the first diffraction order at the wavelength of λ_2 ($\lambda_1 < \lambda_2$). The inset shows the ray paths in the acousto-optic filter

changes in the shape of the crystal, this method can only be applied by the manufacturer. The interested reader may find a deeper explanation and mathematical details in V. B. Voloshinov and Yukhnevich (2012). On the other hand, longitudinal optical aberration is caused by the dispersion of the crystal, and it strongly depends on the the optical scheme that is used for the formation of an image. For that reason, its correction must be studied in parallel to the optical scheme design. Both topics will be treated together in the next section.

2.4 Optical fundamentals of the system

Getting sharp macroscopic images is not straightforward when working with an AO filter. Not only the system has to deal with the intrinsic chromatic aberrations, commented in the former section (2.3.2), but it must also take into account other effects.

Firstly, the AOTF does not diffract every beam of light coming through the crystal. Instead, only light beams coming inside the cone centered on the optical axis, and defined by the *acceptance angle* will be diffracted. Secondly, light rays must be as parallel as possible to the optical axis before entering the AOTF. It is important for the Bragg diffraction to happen in an optimal

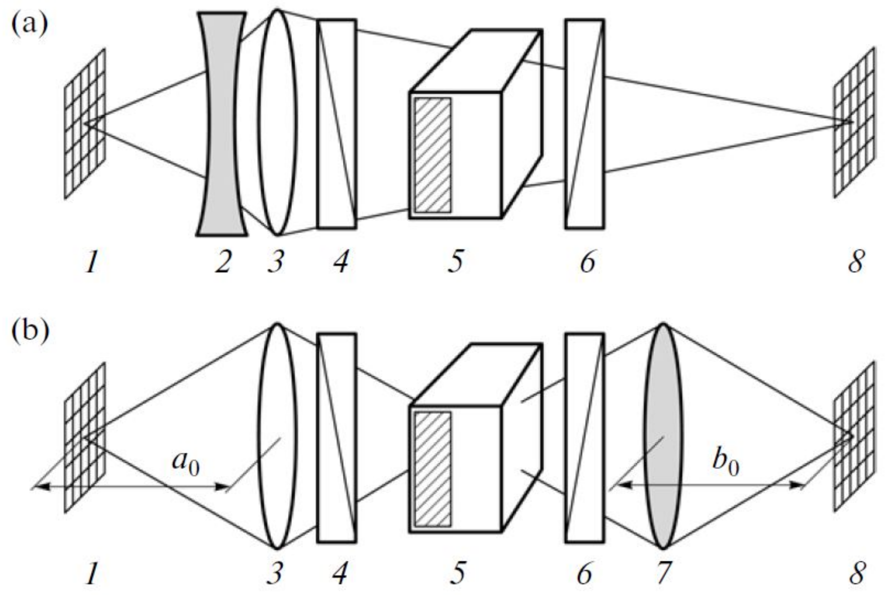


Figure 2.8: Improved optical schemes of the spectral image analysis systems with reduced longitudinal chromatic aberration: (a), with an additional negative lens, and (b), a two-lens confocal system. 1, Object; 2, negative lens; 3, objective; 4, polarizer; 5, tunable AO filter; 6, crossed polarizer; 7, additional lens; 8, image in the first diffraction order.

way.

Longitudinal aberration is caused by the difference in propagation time of light through the crystal for different wavelengths. The longitudinal shift or longitudinal aberration, for a given lambda, is defined by:

$$\delta b_c(\lambda) \approx \frac{\partial n_o}{\partial \lambda}\frac{1}{n_o^2}L_c(\lambda - \lambda_o) \tag{2.2}$$

according to V. B. Voloshinov and Yukhnevich (2012), where L_c is the crystal length, and n_o is the refractive index of the crystal.

The very article also proposes two different systems to correct longitudinal aberrations, both shown in Figure 2.8. The first one, represented in Figure 2.8(a), is the principal optical scheme of an image-processing system with an AO filter considered to study the chromatic aberrations in the former section (Figure 2.3.2). See that the system contains now an additional negative lens. It introduces a negative longitudinal chromatic aberration whose exact mathematical expression can be found in the literature. It is a function of two parameters (of course, it also depends on the wavelength) $\delta b_1(\lambda; a_1, l_1)$, where a_1 is the distance between the object and the negative lens ($a_1 > 0$), and l_1 is the distance between the negative lens and the objective ($l_1 > 0$). Its negative sign allow us to compensate for the positive shift defined by Equation 2.2. Therefore, the total longitudinal chromatic aberration in the system is defined by:

$$\delta b_c = \delta b_c + \delta b_1 \tag{2.3}$$

It should be noted, however, that the parameters a_1 and l_1 that may be varied to correct the chromatic aberrations are furtherly constrained by a condition that requires that the objective should form a real image at a finite distance:

$$l_1 - \frac{a_1 F_1}{a_1 - F_1} > F_0 \tag{2.4}$$

where F_0 is the focal length of the objective, and F_1 is the focal length of the additional negative lens.

As shown in the mentioned article, and also in Dennis R. Suhre and Gupta (2004), a *confocal* configuration based on two objectives offers certain features that are important for imaging systems. Remarkable characteristics are their versatility, which is the ability to change maginification factor of an image keeping focal lengths of lenses constant, and the abilityy to compensate longitudinal chromatic aberrations of AO filter to nearly any degree. The optical layout of such a system is represented in Figure 2.3.2(b). The difference comparing to the system shown in Figure 2.3.2(a) is that the negative lens (2) has been replaced by a collecting lens. Therefore, the object to be imaged is placed in the front focal plane of the first objective, while the image is formed in the back focal plane of the exit objective. Such a system is *free* from longitudinal chromatic aberration. While the afocal system provides fixed image magnification factor, the confocal system allows to vary it, what is important to match the field of view of the device with the size of the photomatrix.

The longitudinal aberration in the confocal system is studied now. The system is formed by two identical objectives with a focal length F_0 and the distance between their principal planes is L_0. The object is placed at a distance a_0 from the front principal plane of the first objective and the image is formed at a distance of b_0 from the back principal plane of the second objective, provided that the central wavelenght is λ_0. Figure 2.3.2(b) illustrates the system described here. The variation of the optical length between both lenses produced by dispersion of the refractive indices of the crystal, leads to the shift of the principal planes of the entire system and also to the change of the effective focal length. The displacement of the image plane can be calculated as:

$$\delta b_0 = \frac{aF_0^2 d\Delta l[3a(l-2F_0)+4F_0^2+ad\Delta l]}{[a(l-2F_0)-F_0^2]^2(l-2F+d\Delta l)} \tag{2.5}$$

where $l = l_0 + \Delta l$, $a = a_0 - lF_0/(l-2F_0)$ and the relative dispersion d is:

$$d(\lambda) = \frac{n_o(\lambda)-n_o(\lambda_0)}{n_o(\lambda_0)-1} \tag{2.6}$$

The parameters Δl and F_0 are defined by the choice of the lenses and the sizes of the acousto-optic cell during the design process. On the other hand, a_0, l_0, and b_0 can vary during the system tuning. According to V. B. Voloshinov and Yukhnevich (2012), the best compensation of the aberration is achieved under the condition $b_0(\lambda_{min}) = b_0(\lambda_{max})$. A zero image shift $\delta b_0 = 0$ then corresponds to the relative dispersion upon wavelength tuning from λ_{min} to λ_{max}. Equation 2.7 is obtained from this condition. It links the parameters of the optical scheme (a and l) to the relative dispersion d:

$$a = -\frac{4F_0^2}{3(l-2F_0)+d\Delta l} \tag{2.7}$$

The choice of a and l according to Equation 2.7 retains the possibility of changing the magnification coefficient of the system freely, which is one of the advantages of confocal systems, as it has been mentioned before.

According to Dennis R. Suhre and Gupta (2004), a *telecentric confocal* configuration can perform even better. Such a system should compensate for the chromatic aberrations, as the confocal design does. Besides, it should provide the same resolution and diffraction efficiency throughout the scene, leading to a uniform image field.

Telecentric optics are built by adding apertures (they may be sometimes referred to as pinholes here) located at the object and image focal planes. Such a system is telecentric for both the object and the image space. A scheme of the telecentric optics is shown in Figure 2.9 in which polarizers are not represented for simplicity. Telecentric optics suppose an improvement in the

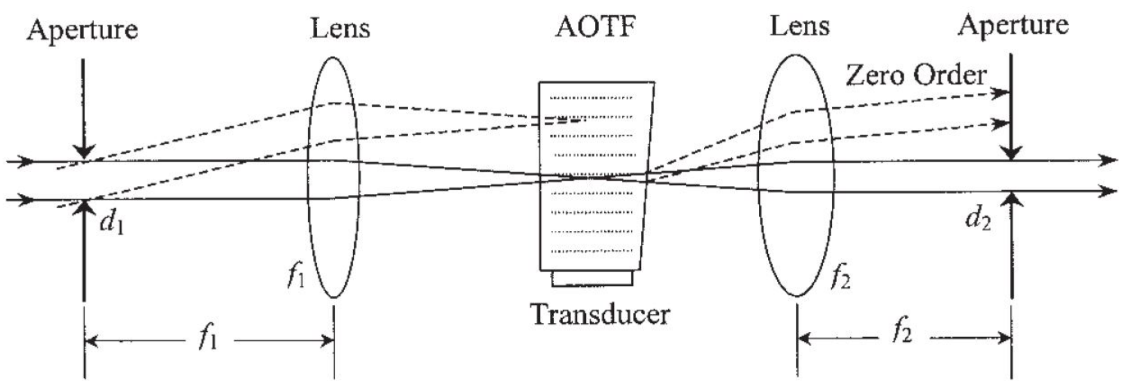


Figure 2.9: Schematic diagram of the telecentric confocal optics for the AOTF system.

way that they tend to reduce errors caused by defocusing. This is due to the fact that the principal ray of a pencil of rays passing through a telecentric aperture is parallel to the optical axis of the system after focusing and does not produce a lateral displacement of the image. Then the aim of having parallel beams in the AOTF can be achieved. This provides a constant input angular spread for the AOTF so that the resolution and efficiency of the AOTF, which are functions of the input angle, will be constant over the field of view of the input scene. The second aperture blocks the zero order beam, therefore, only the filtered beam of light reaches the camera sensor.

The size of the apertures is selected such that the acceptance angle of the AOTF matches the input f-number (the ratio of the lens's focal length to the diameter of the entrance pupil). The output aperture is also adjusted such that $f_2/d_2 = f_1/d_1$, where f_1 and f_2 are the input and output lens focal lengths and d_1 and d_2 are the input and output aperture diameters. This allows the entire input beam to be transmitted through the output aperture while the zero-order beam is blocked, provided that the diffraction angle is larger than the angular spread defined by the input aperture.

A system summarizing all what has been discussed in this and former sections is shown in Figure 2.9. It is telecentric, as the apertures are located at the focal planes; and it is confocal, as there is a common focal point between the input and the output lenses, at the middle of the crystal. From the optical point of view, the system consists of the front-end optics (FEO) and the back-end optics (BEO). A large f-number of the optics is desired to provide a large depth of focus at the detector focal plane position. The polarizers in the FEO and BEO are cross-oriented to achieve highest extinction ratios of the rejected light, as it was described in section 2.3.1. Their

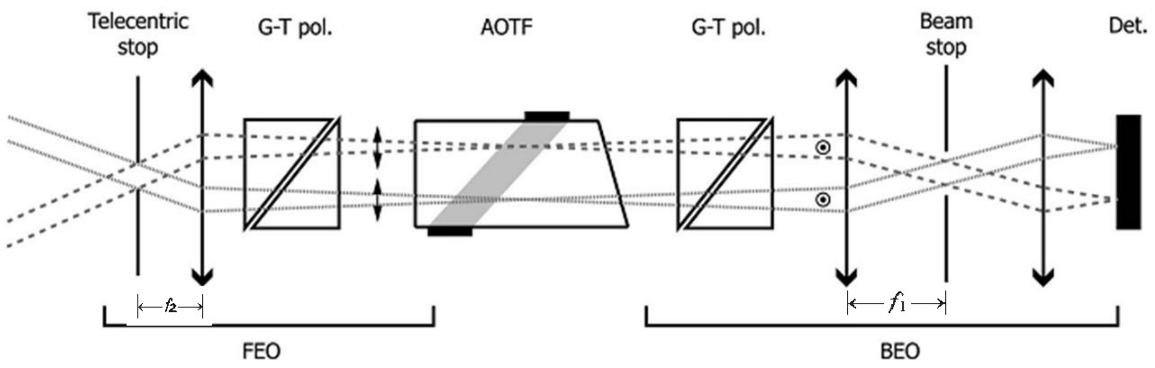


Figure 2.10: Optical scheme of the telecentric confocal configuration for the AOTF system. The system is composed of a frond-end optics (FEO), AOTF, a back-end optics (BEO) and a camera (Det.). G-T pol. stands for a polarizer.

use in combination with the BEO pupil ensures efficient stray light reduction.

2.4.1 Radial distortion

We have already mentioned the chromatic aberrations that the system may suffer, specially because of the use of the AOTF. But there are other optical distortions that the system may present. The most commonly encountered distortions within an optical system are radially symmetric due to the symmetry of the system around its optical axis. Usually, the radial distortions are classified as either barrel distortions or pincushion distortions. Barrel distortion is produced if the image magnification *decreases* with distance from the optical axis, its apparent effect is that of an image that has been mapped around a sphere. Pincushion distortion is the opposite effect, it appears if the image magnification *increases* with the distance from the optical axis. Its visible effect is that lines that do not go through the centre of the image are bowed inwards, towards the centre of the image. Examples of both types of radial distortions are shown in Figure 2.11. Their effect is quadratic with the distance from the center of the image. A mixture of both types might appear and its called mustache distortion. Radial distortion can be corrected using Brown-Conrady's distortion model (Conrady, 1919) and applying software processing, as long as low order radial components dominate.

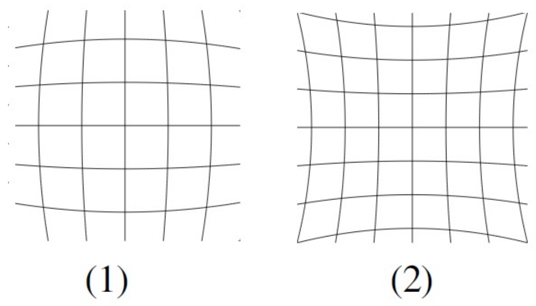


Figure 2.11: Result of imaging a grid when (1) barrel distortion is present (2) pincushion distortion appears.

2.5 Matlab User Interface

MATLAB®(**Mat**rix **Lab**oratory) is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis and numerical computation. It was developed by MathWorks. Among its many features, some specially relevant for this work might be found. It allows the creation of user interfaces in a user-friendly environment. It supports interfacing with programs written in other languages including C++, necessary to communicate with the camera selected. It provides huge capabilities regarding image processing and data analysis that should be exploded in further steps of the system development. Complex algorithms and new functionalities can be easily implemented as there is a vast library available, most of it supported with official documentation. Besides, a lot of online communities can be found that provide a really useful, non-official support.

2.5.1 Interfacing with other languages

Matlab can call functions and subroutines written in the C programming language. These programs, called binary *MEX-files*, are dynamically linked subroutines that the Matlab interpreter loads and executes. The MEX-file contains only one function or subroutine, and its name is the MEX-file name. This function can then be called from Matlab, as if it was a MATLAB function, using the name of the file.

The term mex stands for "Matlab executable" and has different meanings, source MEX-file refers to the C source code file while binary MEX-file is referring to the dynamically linked subroutine executed in the Matlab environment. The MEX function library, on the other hand, is the Matlab C API Reference library to perform operations in the Matlab environment.

MEX-files are often used to speed up code of complex algorithms. Although the development time of the algorithm is longer, it runs faster if it is written in C than if it is written in Matlab. However, optimizing processing times is not a priority yet. In fact, if optimizing the execution speed had been the goal, the user interface should have been migrated to C at the very first step. Instead, having access to Matlab's potential is thought to be more beneficial, in order to reduce development time and increase functionality. It is left as an idea for future optimization of the system: migrating resources greedy algorithms to C by using MEX-files.

MEX-files will still be used though. As the Application Programming Interface to control the camera is available in C, they are needed to interface with the camera from MATLAB.

2.5.2 Portability

Portability of the software is not among the goals to be achieved by this work. However, a brief comment is done here which is thought to be a starting point if it is required for future development of the system.

Once an application is built in Matlab, it can be distributed to other Matlab users, packaged as a Matlab app, which provides a single file for distribution. It may also be shared with others who do not have Matlab by using application deployment products. These add-on products automatically generate standalone applications, shared libraries, and software components for integration in C, C++ orJava environments. The executables and components can be distributed royalty-free. Anoher option to achieve portability is to use the Matlab Coder to generate standalone C code from Matlab code. It has the inconvenience that it only supports a subset ot the Matlab language.

Chapter 3

Building AOTF system

Once the processes involved in a hyperspectral imaging system have been presented, the actual construction of the system follows. For that purpose, in this chapter, we come back to the different elements of the system that have been treated in the Chapter 2 to present their specifications. When it is considered relevant, information will be provided about how or why a particular device was chosen to be part of the system, as for a given element of the system there are many options available in the market. By the end of this chapter, the reader should be able to picture an idea about what performance can be expected from the system.

3.1 Andor Zyla sCMOS camera

The scientific camera used in this project is manufactured by Andor® and it corresponds to the model Zyla sCMOS 5.5, which offers high speed, sensitivity and resolution. The thermoelectrically-cooled design ensures a low readout noise by lowering the dark current, and thus, reducing the effect of blemishes in the sensor.

3.1.1 Camera selection

The camera selection was a key aspect for the performance of the system. Two technologies able to provide the performance requiered were available. Scientific complementary metal-oxide-semiconductor (sCMOS) is a revolutionary technology based on CMOS image sensor design and fabrication techniques (Dr. Colin Coates, 2009). Conventional CMOS cameras offer very

EMCCD iXion Ultra	sCMOS Zyla
Dark noise ≈ 200 ± 20	Dark noise ≈ 100 ± 10
Dynamic range ≈ 12 – 18%	Dynamic range ≈ 93 – 97%
SNR_{real} low LS ≈ 26 - 368, with average 170	SNR_{real} low LS ≈ 20 - 600, with average 120
SNR_{real} high LS ≈ 35 - 801, with average 234	SNR_{real} high LS ≈ 2 - 6545, with average 370: *saturated*
Contrast ≈ 0.10	Contrast ≈ 0.04
Frame rate $\propto$ integration time (up to 50 fps)	Frame rate depends on other factors (max 14 fps)
512 x 512 pixels (0.25 Mega)	2560 x 2160 pixels (5.5 Mega)

Table 3.1: First test results summary: RTP on a white paper

fast frame rates but compromise dynamic range. sCMOS image sensors, on the other hand, offer extremely low noise, rapid frame rates, wide dynamic range, high quantum efficiency, high resolution, and a large field of view simultaneously in one image. This makes them particularly suitable for high fidelity and quantitative scientific measurement. However, Electron Multiplying CCD (EMCCD) is a more widely used technology for high performance applications (web site EMCCD). Although sCMOS readout noise is very low compared to CCDs, it does not hold the distinct advantage of being able to practically eliminate read noise, as EMCCD does.

To have a better knowledge about what performance could be achieved by using each sensor technology, two cameras were compared under experimental conditions: the sCMOS based Zyla 5.5 camera and the emCCD based iXion Ultra 897 camera. Both cameras were provided by Andor®. Three different tests were performed, whose results summary is presented below.

1. WHITE LIGHT REFLECTANCE TEST: The first test performed was regular imaging of a resolution test pattern (RTP) printed on a white paper. It was performed using a circular white light source at different levels (Illumination Associates, IT 2900) and a 25 mm lens was fixed to the camera. The Table 3.1 contains the results of this test. The advantages of one camera over the other are marked with red text color. It must be mentioned that the low contrast level achieved by the Zyla camera is due to the optics used for the test. The lenses used were not optimized for high resolution cameras. It affects more the Zyla camera because of its higher resolution: 5.5 Megapixels, 22 times more than the iXion.

2. FLUORESCENCE TEST: The second test was imaging the fluorescence of fluorescein irradiated by a 280 nm light source at different powers (SETi deep UV LED). The same 25 mm lens was fixed to the camera. The Table 3.2 contains the results of this test. Again, the ad-

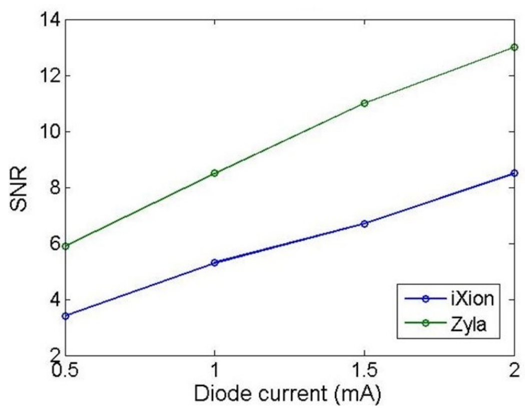
EMCCD iXion Ultra	sCMOS Zyla
Dark noise ≈ 200 ± 20	Dark noise ≈ 100 ± 10
Dynamic range ≈ 35 – 73%	Dynamic range ≈ 4 – 21%
Mean SNR_{real} ≈ 3.4 - 8.5 at currents 0.5 - 2.0 mA	Mean SNR_{real} ≈ 5.9 - 13 at currents 0.5 - 2.0 mA
Frame rate $\propto$ integration time (up to 55 fps)	Frame rate depends on other factors (max 4.4 fps)

Table 3.2: Second test results summary: fluorescein irradiated by a 280 nm light source

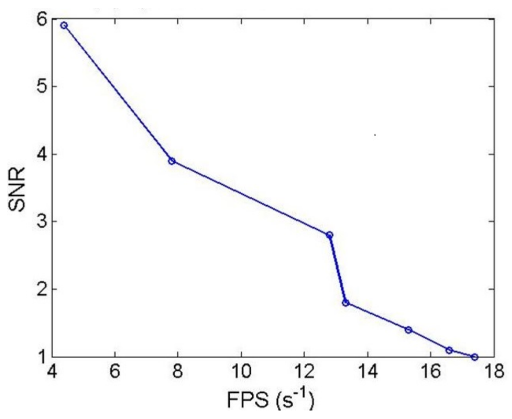
vantages of one camera over the other are marked with red text color. It is interesting that the dynamic range depends on the integration time used. The same integration time of 0.2 s was used for both cameras to provide similar testing conditions. However, this integration time for the Zyla camera was lower than the optimal one, therefore the maximum dynamic range was not reached.

Figure 3.1 is a good summary for this test. Using the same camera settings as in the former test, Zyla camera performed better, i.e. resulting in higher mean SNR values, as it is shown in 3.1(a). On the other hand, it can be seen on 3.1(b) that the SNR drops significantly by increasing frame rate. It was also noticed that in images obtained by Zyla camera, multiple hot-spots were present, i.e. pixels with different sensitivity. There were no hot-spots in the images taken by iXion Ultra. However, these aberrations can be corrected by the Spourious Noise Filter feature provided by the Zyla camera, which replaces them with the mean value of their neighbouring pixels.

3. AOTF TEST: The third test consisted of imaging the resolution test pattern irradiated by white light and filtering the light using the AOTF. It is the most relevant experiment, since it evaluates the performance of the camera placed in an environment which recreates the final system. A very basic setup was used since proper optics were not available at the time of the test. A confocal, but not telecentric, setup was used. Front optics and back optics consisted of 2x 25 mm lenses (25mm FL Compact Fixed Focal Lenght Lens, Edmund Optics) and the camera lens was 75mm. The broadband 1" wire-grid polarizers (EdmundOptics) were inserted in the cross-polarized configuration before and after the AOTF device. Imaging distance was approximately 20 cm. The manufacturer software was used to control the camera and the AOTF device, since this was a preliminary test done before the system software was available.



(a) SNR at $t_{int} = 0.2s$



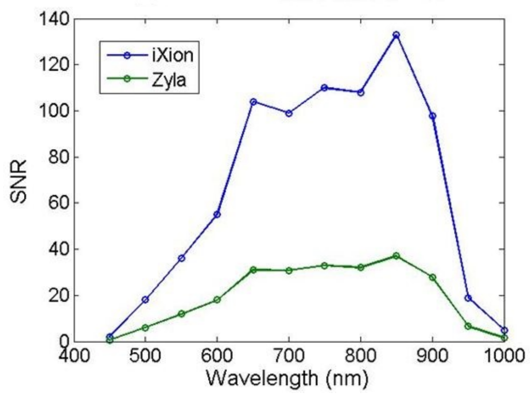
(b) Zyla: SNR at diode current = 0.5 mA

Figure 3.1: (a) Mean SNR values for both cameras (see legend), at integration time 0.2 s, for different diode currents. (b) SNR dependency on the frame rate (integration time) for Zyla camera, diode current fixed.

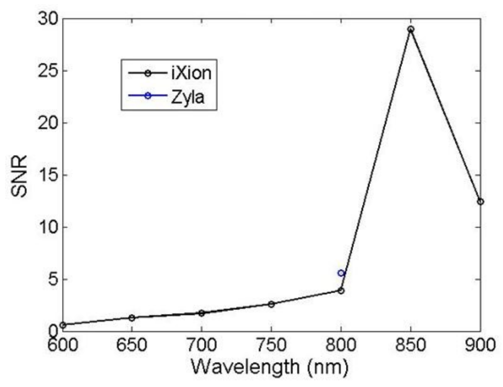
Figure 3.2(a) presents SNR of whole image obtained at 12 wavelengths in the spectral region from 450 nm to 1000 nm. Using the same integration time 0.2 s and good illumination at LS = 10, iXion Ultra results in images with on average 3-times larger SNR value as compared to the Zyla. When low illumination power is used, LS = 0, as shown in Figure 3.2(b) only iXion Ultra yields results in acceptable time. SNR levels were however on average 4-times lower than for the good illumination scenario. Zyla camera provided images with the similar SNR, but using 20-times higher integration time, resulting in extremely slow acquisition speed, 0.25 fps with Zyla in comparison to 55 fps with iXion Ultra.

Figure 3.3 shows a spectra comparison among: iXiom Ultra (black), Zyla (blue) and spectrometer (S2000, Ocean Optics) (red). (a) Presents the regions where spectra were collected. They are marked by the white and the black rectangles. They correspond to a black and a white region in the image respectively. For pictures (e) and (f) Zyla requires a really long integration time: 4s. A significant difference between the obtained spectra can be observed. The difference is most likely due to different detector size and consequently the field of view. Using iXion Ultra whole camera's field of view was filled with an image, while Zyla had larger field of view.

Based on the test results, it can be concluded that Zyla is the optimal choice if the system is used under normal-light conditions, as it will be, resulting in acceptable SNR levels, high res-



(a) SNR at $t_{int} = 0.2s$ and $LS = 10$



(b) SNR at $t_{int} = 0.2s$ and $LS = 0$

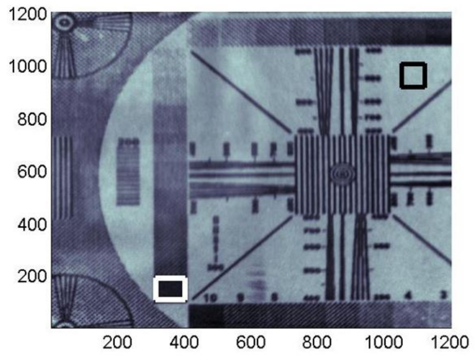
Figure 3.2: (a)Mean SNR values for both cameras (see legend) at integration time 0.2 s, $LS = 10$. (b) Mean SNR values for both cameras (see legend) at integration time 0.2 s, $LS = 0$.

olution and good acquisition speed. For low-light conditions iXion Ultra performs better, but offers significantly lower resolution for high speeds. These conclusions are consistent with the information provided by the manufacturer in Andor® (2014b) and shown in Figure 3.4: the performance advantage of EMCCD is important when working with low light intensities, but it is negligible for well illuminated environments.

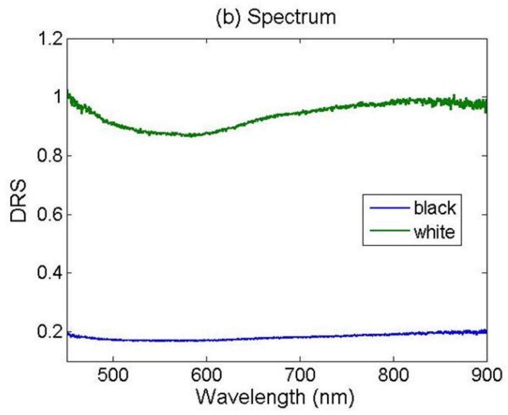
Price difference must also be taken into account, and it turns out that the iXion Ultra is 2 times more expensive than the Zyla. Thus, sCMOS offers an affordable alternative to EMCCD sensors and, for the purposes of this work, sCMOS technology widely fits the quality constraints. Therefore the Zyla sCMOS will be integrated in the system.

3.1.2 sCMOS sensor

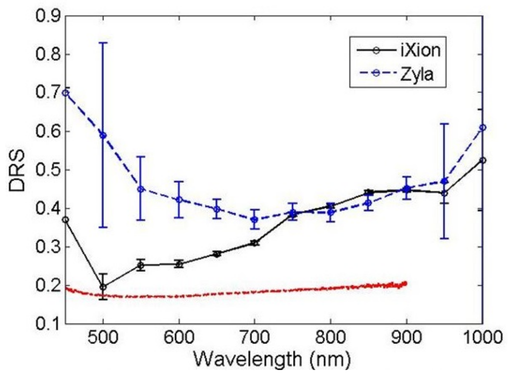
A parameter of interest for the sensor is its quantum efficiency which is represented in Figure 3.5 as supplied by the manufacturer in Andor® (2014b) and is a function of wavelength. Such a curve represents the percentage of photons hitting the sensor suface that will actually produce a charge. As it can be seen, quantum efficiency barely reaches 60% on its maximum, and it decreases below 30% for wavelenghts upon 800 nm.



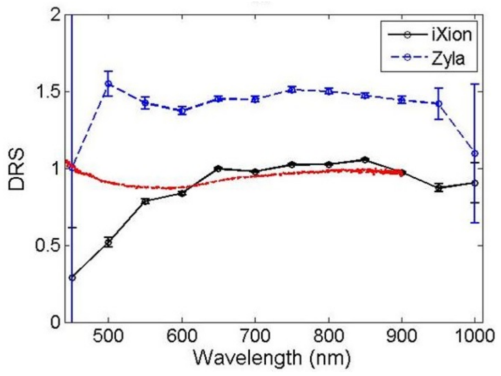
(a) Regions where spectra were collected



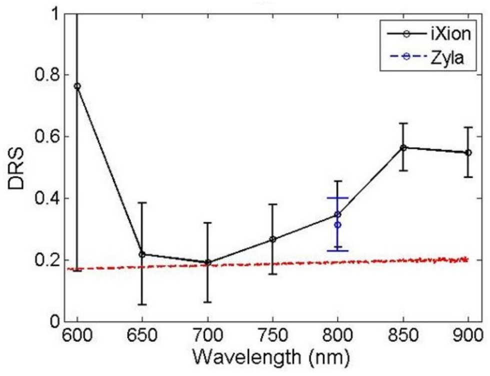
(b) Spectra measured by Ocean Optics spectrometer.



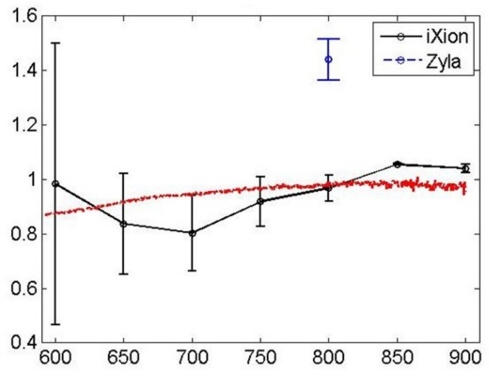
(c) Black field at LS = 10.



(d) White field at LS = 10.



(e) Black field at LS = 0.



(f) White field at LS = 0.

Figure 3.3: (a) shows the regions selected to obtain the spectral information – a black area and a white area. Figure (b) presents the corresponding spectra mesaured by Ocean Optics spectrometer. The corresponding DRS spectra are presented in (c),(d),(e) and (f)

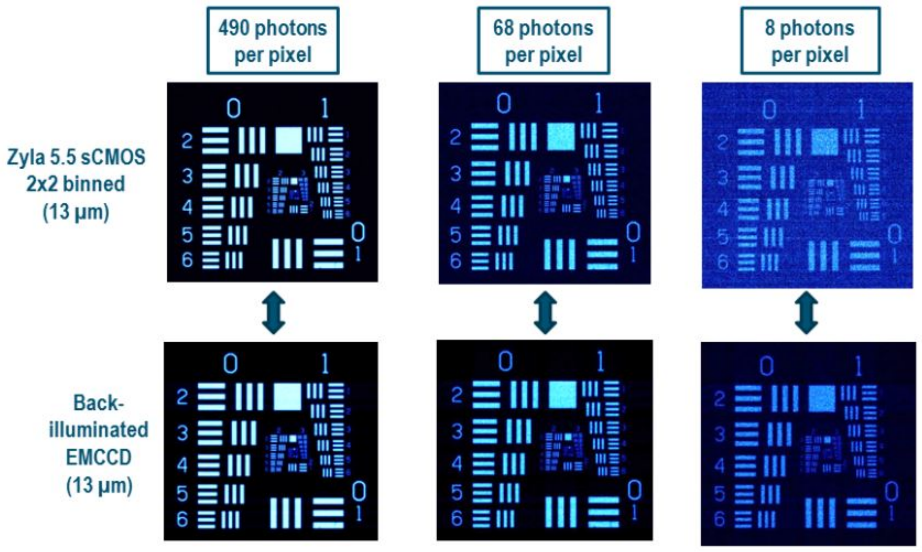

490 photons per pixel
68 photons per pixel
8 photons per pixel
Zyla 5.5 sCMOS 2x2 binned (13 µm)
Back-illuminated EMCCD (13 µm)

Figure 3.4: Images at a range of incident light intensities, acquired using back-illuminated EM-CCD iXon 888 and Zyla 5.5 sCMOS cameras

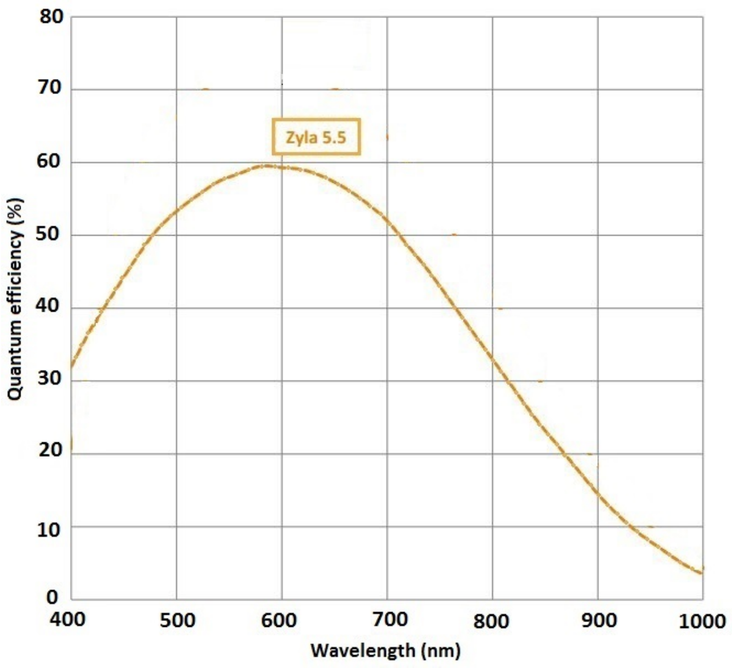


Figure 3.5: Quantum efficiency of the sensor at 20°C

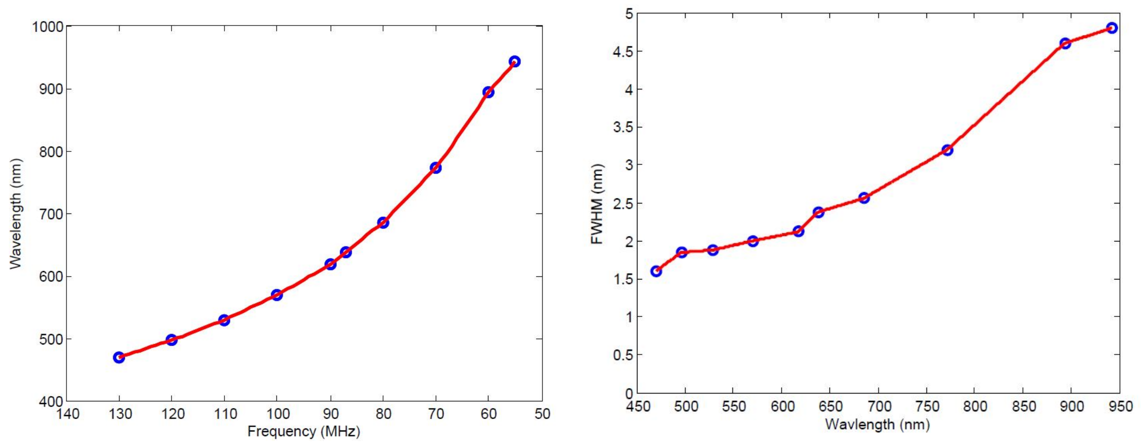
3.2 Acousto optic tunable filter (AOTF)

In our case, an AOTF manufactured by Gooch & Housego®is used (model number TF625-350-2-12-BR1A), which is made of a tellerium dioxide (TeO_2) crystal. This AOTF is based on a non-collinear design operating at the parallel tangents condition, therefore, it was optimized to have a relatively wide field of view: $\pm 2°$ according to its specifications sheet Gooch&Housego (2014a). The time required for the filter to provide and stable spectral output, after a frequency change command is sent, is called the *tunning time* and it is an important parameter. There are two terms contributing to the tunning time:

$$T_{tunning} = T_{RF} + T_{acoustic}. \tag{3.1}$$

Where T_{RF} is the time required by the driver to switch the RF signal, it is around 2 μs, as specified by the manufacturer, small compared to $T_{acoustic}$. Up to 64 profiles defining the filter shape can be stored in the RF driver, then switching only requires to process a new instruction (no a command to be read), and update the RF output. $T_{acoustic}$ is the time the acoustic wave needs to propagate through the crystal, (at a speed of 620 $\frac{m}{s}$), also called access time. C Stedham (2008) claims that the access time for our crystal is less than 25 μs. Thus, the tunning time for our crystal should be around $T_{tunning} = 2\ \mu s + 25\ \mu s = 27\ \mu s$. A more realistic assumption could be made about having a tunning time of around 50 μs ($T_{tunning} \approx 50 \mu s$).

The spectral range of the device specified by the manufacturer is between 450 nm and 800 nm with FWHM 0.65 nm at 457.9 nm increasing to 3.5 nm at 800 nm. Laboratory measurements were carried out to assess the performance of the device. A circular white light source at the maximum power (Illumination Associates, IT 2900) was used as a light source in front of AOTF. One polarizer in front of the AOTF and another behind AOTF were used in the cross-polarized configuration to reduce the unwanted spectrum of the light source. The light transmitted through AOTF was collected onto an optical fiber (50 μm VIS-NIR, Ocean Optics) and a spectrometer (S2000, Ocean Optics) was used for detection. The results, shown in Figure 3.6, are consistent with those provided by the manufacturer in Gooch&Housego (2014a). Therefore, the AOTF is characterized. For a given RF frequency input, it is known where the optical filter is placed, i.e.the peak wavelength of the filter, and its spectral width, i.e. the full width at half



(a) Peak wavelength vs. RF frequency (b) FWHM vs. Wavelength

Figure 3.6: Measured (a) peak wavelength vs. frequency and (b) FWHM vs. peak wavelength for AOTF and white light source.

maximum (FWHM).

3.2.1 RF driver

The spectral response of the AOTF is controlled, as it has already been mentioned, by changing the frequency of the applied RF signal. The RF driver MSD0XX-YYY-10UC-16x1 manufactured by Gooch&Hosego® also allows to vary the power of the RF signal, by doing this the amplitude of the diffracted light can be controlled. This may be useful, for example, to compensate the differences in quantum efficiency of the camera sensor for different wavelengths. With this device is also possible to apply several RF signals at the same time, up to 16, adding them. A more complex filter shape could be obtained, however for this application an ideal band-pass shape is of interest, so no attention will be paid to such feature.

A USB interface communicates with the host computer for control and setup. Four four-channel Direct Digital Synthesizer (DDS) chips controlled by a master processor are responsible for the signal synthesis. Up to 64 complete sets of channel data, called profiles can be stored in the RF driver's memory. This allows to predefine a behaviour, and then, the AOTF will switch from one instruction to the next when receiving an incoming TTL pulse. This procedure will be further developed in the synchronization section (4.2).

Cooling requirements were surprisingly high for this device, and unfortunately, they were

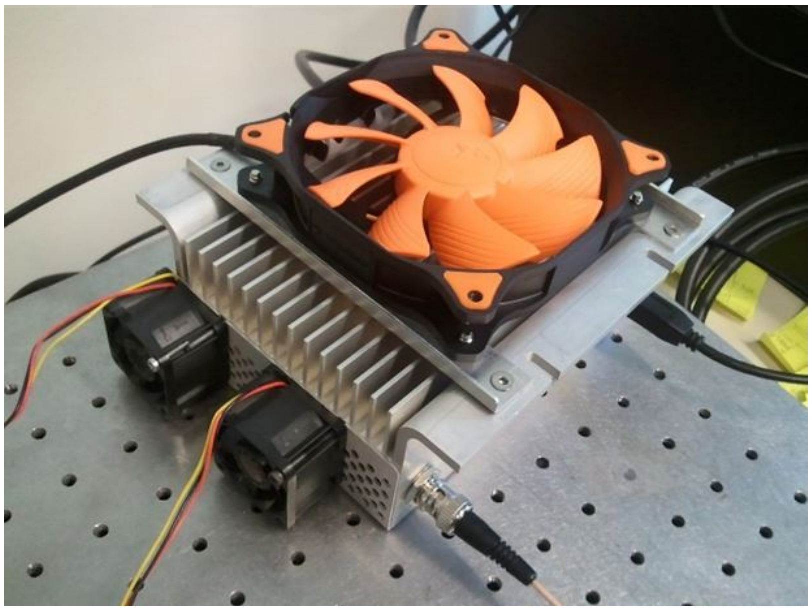


Figure 3.7: Three cooling fans installed on the RF driver to prevent overheating

not fully satisfied at the earliest steps of the system construction due to insufficient instructions in the manual (Gooch&Housego, 2014b). This led the driver to become unresponsive and/or unstable. In particular, channels from one to four and nine to sixteen did not function, while channels from five to eight required occasional restarting of the device to work, making the system unstable.

Feedback from the manufacturer revealed that the USB chip was partialy burnt, which was causing the device to fail. Then, the options that could have led this to happen were explored. The possibility of having supplied a voltage higher than the allowed by the device was considered unlikely: a standard power supply was used, giving 24 V to the RF port and 3.3 V to the USB port. Insufficient cooling seemed to be the most probable cause. The manufacturer was contacted again for further information about cooling requirements. The device was supposed to work properly cooled if an air flow higher than 17 litres per second was passed through it. Three cooling fans were installed on the device, as shown in Figure A.1.2. The fan on top of the cooling fins, the orange one, provides more than 33 $\frac{l}{s}$, while each of the two smaller fans at the side provides more than 9.4 $\frac{l}{s}$. So, a total of around $33+2*9.4 \approx 51.8$ $\frac{l}{s}$ was provided. Approximately, three times the stated requirement. However, the device became unresponsive again, and it had to be sent back to the manufacturer, where it is still being repaired.

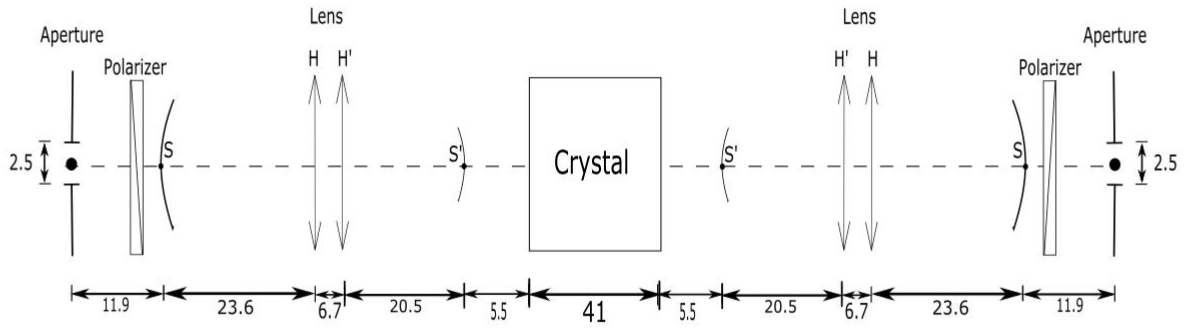


Figure 3.8: Detailed scheme showing the final optical configuration of the system.

3.3 Optical design of the system

As it was theoretically justified in the Section 2.4, a telecentric confocal configuration is our choice, besides, to help to get rid of the stray light (all beams except first order transmitted beam), polarizers in a cross-polarized configuration are also included in the system. The final detailed scheme is shown in Figure 3.8. Now, it will be explained how such a scheme was conceived.

The main restriction of the system is imposed by the AOTF. As it was mentioned in the former section (3.2), it has a field of view given by its acceptance angle of $\theta = \pm 2°$. This condition must be satisfied to have parallel beams in the crystal and therefore, achieve optimal Bragg diffraction. Hence, satisfying this condition determines the size of the circular aperture, in particular, its radius which will be referred to as r_D. Figure 3.9 presents an scheme used for the calculation of r_D. The green rays (dashed lines) are refracted by the lens and, therefore, will be parallel when entering the crystal. On the other hand, the red rays (dotted lines) are not affected by the lens, their angle θ respect to the parallel rays should be less than 2 degrees (the acceptance angle).

$$\tan\theta = \frac{r_D}{f} \Longrightarrow r_D = f\tan\theta \tag{3.2}$$

Using the specifications of our lenses $f = 35mm$ (provided in the next Section 3.3.1), the optimal size of the pinhole is: $r_D = 35 * \tan(2) = 1.22mm$.

From Section 2.4 we had: "the system is telecentric, as the apertures are located at the focal planes; and it is confocal, as there is a common focal point between the input and the output

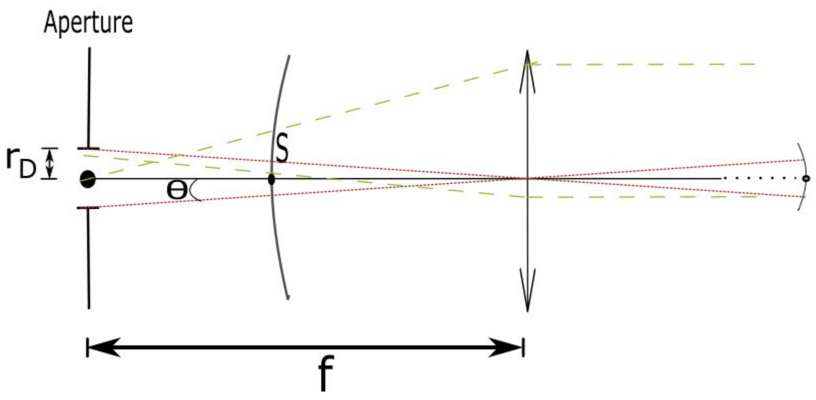


Figure 3.9: Scheme for the calculation of the aperture radius.

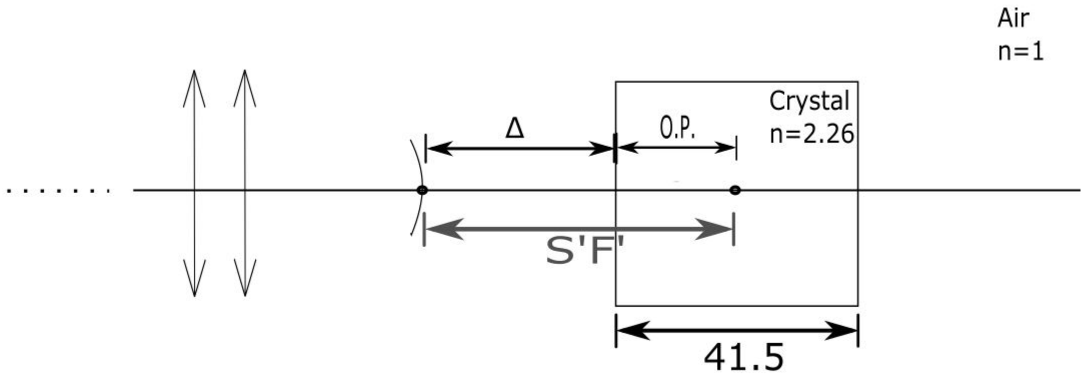


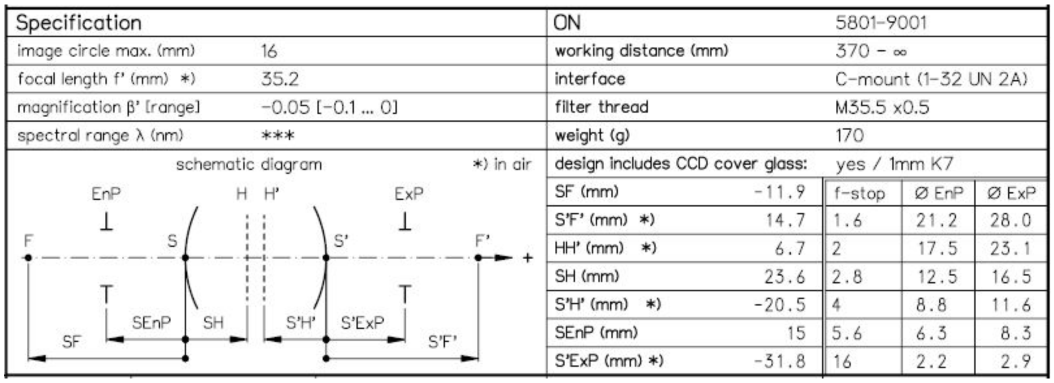
Figure 3.10: Scheme for the calculation of the length of the optical path inside the crystal.

lenses, at the middle of the crystal". Therefore, the focal length of the lenses will determine their position relative to the AOTF, as well as the position of the apertures relative to the lenses.

Length of the optical path inside the crystal is way shorter than its physical length due to its refraction index. In particular, the crystal has a length $l = 41.5mm$, as shown in Figure 3.10 which serves as a scheme to calculate the position of the lenses. Then its central point, that should agree with the back frontal plane of the lenses, is placed at $l/2 = 20.75mm$ from the entrance. But the distance *viewed* by the light is OP (Optical Path):

$$OP = \frac{l/2}{n_c} = \frac{20.75}{2.2565} = 9.20mm \tag{3.3}$$

where n_c is the refraction index of the crystal $(2.2565$ for the wavelength $\lambda = 644)$. Therefore, the



Specification		ON	5801-9001
image circle max. (mm)	16	working distance (mm)	370 – ∞
focal length f' (mm) *)	35.2	interface	C-mount (1-32 UN 2A)
magnification β' [range]	-0.05 [-0.1 ... 0]	filter thread	M35.5 x0.5
spectral range λ (nm)	***	weight (g)	170
schematic diagram	*) in air	design includes CCD cover glass:	yes / 1mm K7

		f-stop	Ø EnP	Ø ExP
SF (mm)	-11.9			
S'F' (mm) *)	14.7	1.6	21.2	28.0
HH' (mm) *)	6.7	2	17.5	23.1
SH (mm)	23.6	2.8	12.5	16.5
S'H' (mm) *)	-20.5	4	8.8	11.6
SEnP (mm)	15	5.6	6.3	8.3
S'ExP (mm) *)	-31.8	16	2.2	2.9

Figure 3.11: Main dimensions and important distances.

lens must be placed in a position satisfying:

$$S'F' = \Delta + OP \Longrightarrow \Delta = S'F' - OP. \tag{3.4}$$

where S'F' is the distance between the back surface of the lens and its back focal plane.

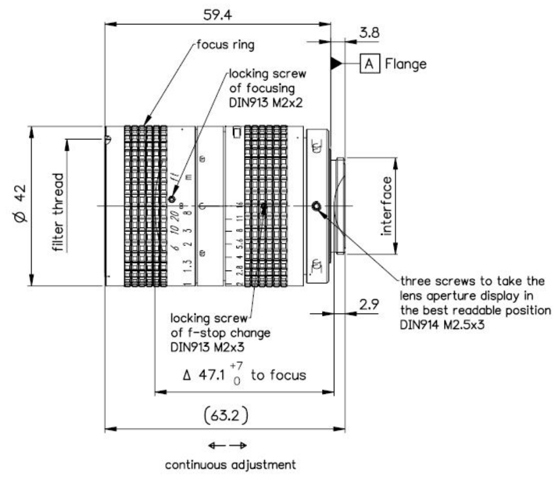
A further physical restriction comes from the cross-polarized configuration. A polarizer must be situated between the lens and its front focal point. In many commercial lenses such a position is not accesible, and thus, they can not be used in our system.

3.3.1 Lenses

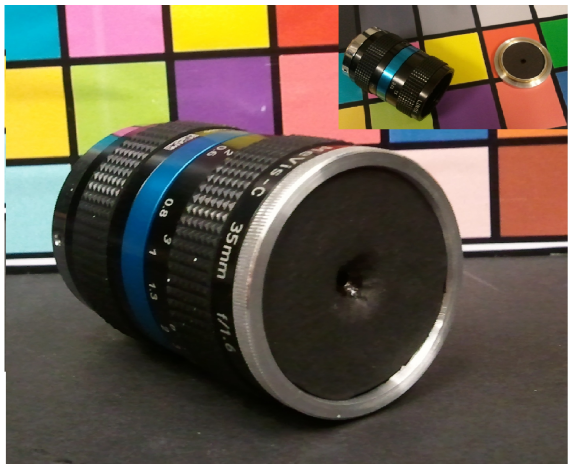
MeVis-35 mm lenses are specifically developed to be used with high resolution sensors (up to 12 Mpixel, while the camera Zyla has a 5.5 Mpixel sensor), according to the brochure QIOPTIQ (2010). Their transmission range is 450-950 nm. From Equation 3.5, and using the distances specified by the manufacturer shown in Figure 3.11, the distance from the crystal surface to the back surface of the lens Δ must be:

$$\Delta = S'F' - OP = 14.7 - 9.2 = 5.50mm. \tag{3.5}$$

Also, from Figure 3.11 follows that the aperture should be placed at a distance $SF = 11.9mm$ from the frontal surface of the lense. It can be seen in Figure 3.12, that the space situated between the lens and the optimal position of the aperture can be accessed. Therefore, polarizers can be included there.



(a) Lenses outline



(b) Picture.

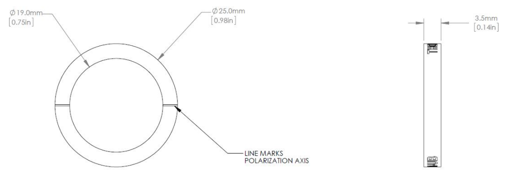
Figure 3.12: Lenses MeVis-35 mm. (a) Outline. (b) Picture of lens including pinhole. Detail showing lens and pinhole separetely.

3.3.2 Polarizers

Polarizers in the cross-polarized configuration are used in the AOTF system to eliminate the unwanted spectra. To achieve the best possible elimination, polarizers with very high extinction ratio must be used. A broadband wire grid polarizers (WP25M-UB, Thorlabs, shown in Figure 3.13) were tested, with specified extinction ratio 1.000:1 and AR coating in 250 nm to 4 μm. The advantage of these polarizers are that they are made into a circular shape, they are thin (approx. 2 mm) and they are resistant to scratching. The first test of the AOTF imaging performed showed that using the wire grid polarizers the unwanted spectrum was not completely eliminated.

In a first approach, polarizers were placed between lenses and crystal, however the performance achieved using such a configuration was lower than expected, while placing them in their final position (between pinholes and lenses) led to significantly higher performance.

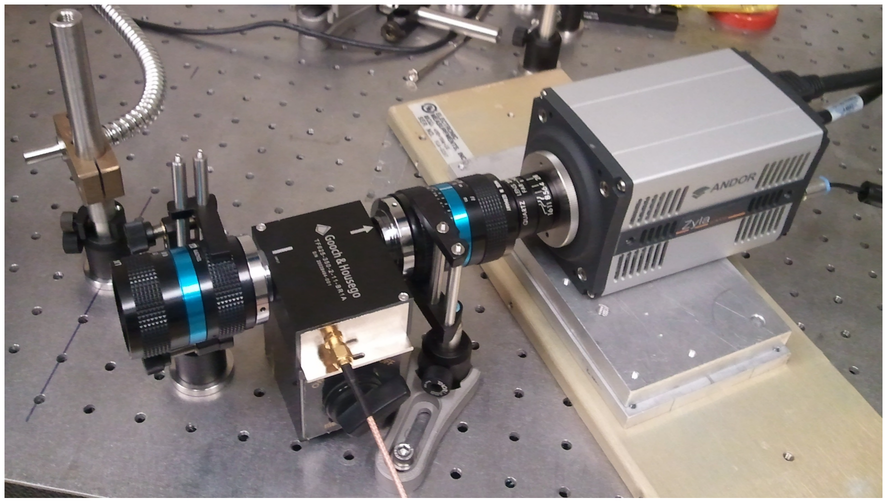
Finally, images of the system are shown in Figure 3.14.



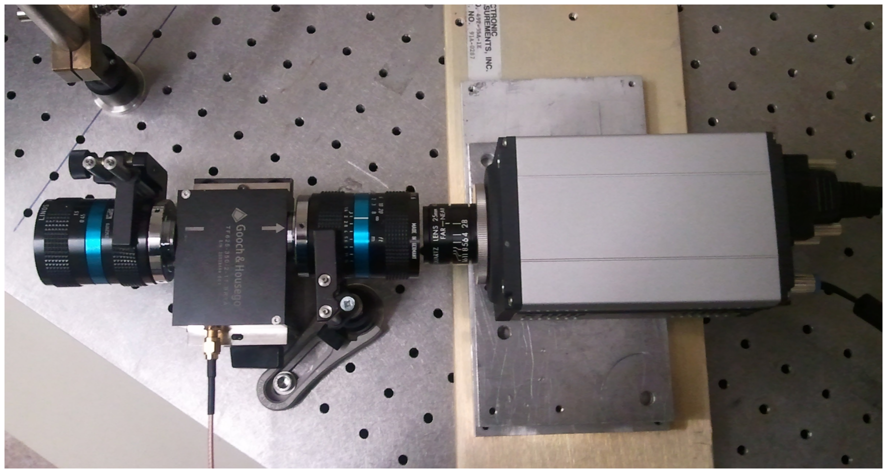
(a) Scheme.

(b) Picture.

Figure 3.13: Polarizer WP25M-UB Thorlabs: (a) scheme and (b) picture. Provided in Thorlabs (2014).



(a) Aerial lateral picture of the system.



(b) Vertical picture of the system.

Figure 3.14: (a) (b) View of the hyperspectral imaging system including: (from left to right) front lens, AOTF, back lens, camera lens and camera. Pinholes are missing, they should be attached to the front and back lenses as in Figure 3.12(b).

Chapter 4

Software

As an important element in the construction of the whole hyperspectral imaging system, an integrated software able to control the system was developed. Its creation and features are treated within this Chapter, while its code is shown in Appendix A.

Both the camera and the AOTF driver have their own software provided by their respective manufacturers. However, although they may be useful for testing operations on a first stage, they can only provide a low performance working together, due to the fact that it takes a long time for a human user to change parameters comparing to how fast a computer can do it. Therefore, an integrated software that controls both the camera and AOTF, and provides processing tools is desirable.

To control the camera, a Software Development Kit provided by Andor is available. It is a simplified Aplication Programming Interface (API) that helps to reduce software development time, and it is documented in detail. It allows to access the current state and limits of camera features and it provides enough functionality to fully control the behaviour of the camera. Its potential can be exploded by simple C programs that can make use of the library atcorem.lib, thus having access to several sets of functions, each controlling a particular aspect of camera control. There are sections in the API for opening a handle to a camera, for buffer management and for accessing the features that every camera exposes.

On the other hand, as it was commented in Section 2.5, Matlab was the platform of our choice to build the Graphic User Interface (GUI) due to the processing algorithms and resources it provides. MEX-files provide the required interface between the Matlab user interface and the camera control routines written in C. An interested reader may find further information about MEX-files and the API provided to control the camera in AppendixMEX files and Andor SDK, which is also useful to fully understand the code.

4.1 Graphical User Interface

A Graphical User Interface (GUI) provides point-and-click control of software applications, eliminating the need to learn a language or type commands in order to run the application. The GUI described here was created using GUIDE, the design environment for user interfaces provided by Matlab.

On the top left corner of the user interface five recognizable icons are placed, as it can be seen in Figures 4.1 to 4.6(b), which allow the user to open or save a hyperspectral image, zoom in, zoom out and navigate through an image displayed in the axes. The last three allow the user to interact with the image in a basic way. It is important to point out that the User Interface should be a useful tool to acquire hyperspectral images, but the further processing of the data is not thought to be done within the GUI. Hence, it is important that it allows to store an image cube to be later processed in Matlab. Using the icon save, an image can be stored at any location of the hard drive as a *.mat file, that could be accessed it in a future moment and easily load into Matlab by using the function `load()` (see Matlab help for further information). It can also be load to the application by using the icon 'Open'.

The reader may consider the difference between the process of saving an image cube and each time the GUI stores a HSI (or any other variable) to be shared within the application as application-data. A mistake made during the earliest stages of developing the GUI was to store variables and data necessary to run the application in the hard drive as a *.mat file, even if that very data was supposed to be load soon by another function. It was not hard to realize that this process of saving-loading variables repeteadly was enormously slowing down the execution of the GUI. Instead, Matlab provides four ways to share data within the application: in an object property called UserData, using the `guidata()` function, as application data, and nesting the callback functions. The last option could not be used as it requires a complete change in the way the GUI is programmed, and it is not recommended. Like the main concern back then was about speeding up the GUI, the other three options were tested to save and load a big matrix while using Matlab `tic, toc` functions to set a timer. It turned out that all them required similar times, but at least 10 times faster than before. Therefore, after carefully reading the documentation, the most convenient one was used: store data as application data, which uses functions such as `setappdata()`, `getappdata()` or `isappdata()`. It is more convenient because it allows to save several variables at a time (the other two do not support this), and it is not necessary to keep record of in which object a variable is stored (as it is with `guidata`).

On the top right corner, beside the axes that occupy most of the GUI, there is a box named *Image Acquisition*. Inside, there is a listbox that allows to select some wavelengths of those shown there. A button called 'List' allows to switch the wavelengths selecting mode to one in which the user should specify the starting and ending wavelengths and step between wavelengths. After switching, the previously named 'List' button is called 'Range' and, by clicking on it, the listbox of wavelengths is again available. In a similar way, the Triggering mode can be switched by clicking on the button Software, which changes its name to Hardware. See that the string defining each of these buttons describes the *current* state, not the result of clicking the button. It only works in that way for these two buttons that have a description string on top of them, i. e. Triggering mode, Wavelength selection. In a glance, the user can see, for example on Figure 4.1, that software triggering mode is activated, and that the wavelengths should be selected from the listbox. But, to switch to hardware triggering, the user should click the button 'Software'.

The buttons 'Live View' and 'Acquire' are also inside the *Image Acquisition* box. The first one provides on live video from the camera, at a low frame rate, around 4 frames per second. Figure 4.2 is a screenshot of the GUI working in Live View mode, see that the button name has changed to 'Stop Preview'. It is useful to do adjustments such like focusing or pointing to the target, that the user needs to do before getting an image cube. For doing this there were several difficulties. A loop in Matlab is used that gets a frame from the camera and displays it in the axes, however the execution of the loop was really slow and the frames were not refreshed in the display. To solve this, the function `drawnow()` is included at the end of the loop, that forces Matlab to refresh the image displayed; and the function to get the frame was simplified to a minimum: it just sends a Software trigger to the camera and gets the memory address of a new frame (see A.2.5).

To make this possible, however, the camera must be properly set before the loop: prepare the buffers where the frames will be stored, establish a maximum number of frames to be acquired, get the camera into Acquisition mode, which is a highly reactive state in which the camera responses very fast; and after the loop: basically, releasing the buffers and Stop Acquisition. This function could be improved by using circular buffers and setting the camera into CycleMode, however, it is not straightforward how to update the frames in the GUI.

On the other hand, ’Acquire’ gets a HSI, using the selected wavelengths as a parameter to properly set the AOTF response before starting the exposure of each frame. An important goal was to exploit the high frame rates that the camera provides. Approaches in which a Matlab loop gets one frame at a time were early discarded as they introduce huge delays between one acquisition an the next one. A much better approach is to rely on the MEX-file to fully get the image cube which will be then send back to Matlab. Hence, the first step is to allocate memory for the image cube, in a single huge buffer, because the MEX interfacing requires it; this big buffer is divided into as many as the number of images to be acquired. Then, the starting address of each buffer is sent to the camera. Then, an acquisition starts, which sets the camera in a highly reactive mode. Triggers are sent to the camera, either software or hardware triggers as described in the next section (4.2), that keeps the frame in its own memory. Once the acquisition is finished, the camera writes into the buffers provided before and frees its memory. Then these buffers are processed to stract metadata information and send the HSI to Matlab. Therefore, three stages can be distinguished: setting camera, taking images, and releasing data. ’Setting the camera’ takes virtually no time comparing to the other two. The ’taking images’ stage is optimized, it gets frames as fast as Software or Hardware triggering allow to do it, which is a great thing. While the last stage delays the output of the function considerably. After the user presses ’Acquire’ the images are quickly taken, although the further operations delay the visible ouput, i. e.frames shown in the GUI. As an example, 50 full frames, with $exposuretime = 0.01s$ were acquired using Software triggering: it took 1 second to take the images, 0.66 seconds to write them in the buffers, and 0.26 to show them in the GUI.

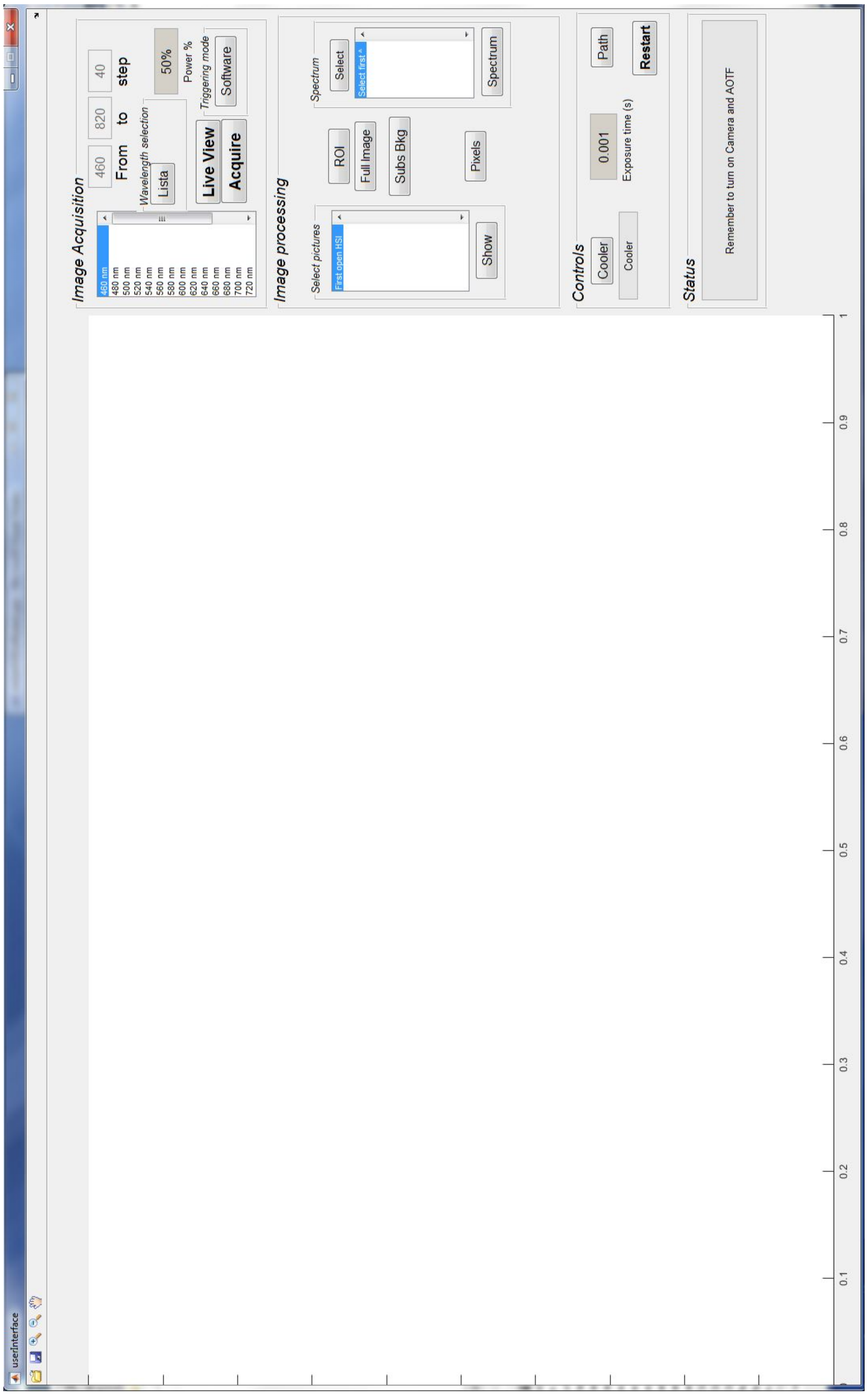
It has been mentioned that metadata information is extracted from each frame. In fact, it is timing information that provides a time reference to know when each frame was acquired relative to the first one. The timestamp associated with a frame contains the clock cycle at which the exposure for the frame started. As the frequency of the clock is available via the integer feature TimestampClockFrequency, the clock cycle information can be *translated* to actual time using:

$$FrameTime[s] = \frac{CycleNumber}{ClockFrequency} \frac{[cycle]}{[\frac{cycle}{s}]} \tag{4.1}$$

This transformation may not be totally accurate due to differences between $ClockFrequency$ as it is stated by the camera, and the *real* frequency of the clock. Anyway, timestamp information might be useful for several things, for example: to see if all the frames within an image cube are taken uniformly in time or not (the operative system may be interfering in the process); time dependant processes might be studied, such as trying to infer the heart pulse of a person.

Outcome of the Acquire function is shown at Figure 4.3(a). In the example, although four frames are taken, only two are shown by default to optimize the axes area working. However, the user can select which frames to show on the listbox Select Pictures, inside the box *Image processing*. It is important that the user has full control to display any frame within the image cube to, for example, visually check if there is barrel distortion at any given wavelength. The result of selecting the four frames is shown in Figure 4.3(b). Similarly, it has been seen that the user sometimes needs to have detailed information of a region in an image. For that purpose, besides the zoom in and zoom out option, the Matlab tool `impixelregion()` is integrated. Clicking on Pixels opens a figurel to navigate through the pixels of an image (Figure 4.4(a)).

When only one frame is selected to be displayed there are some features available. Firstly, a histogram can be shown by clicking on a pixel (Figure 4.4(b)). Values of the corresponding row and column are shown in inferior and lateral axes, respectively. It provides a visual indicator of the sharpness of an image, therefore, the user can study, for example, if a region is more blurred than others within an image, or if some of the frames of the image cube present a higher resolution. Secondly, a ROI can be defined (Figure 4.5(a)) to work with just a small area of the HSI (Figure 4.5(b)), which is thought to reduce processing times. Also, using the sub-box *Spectrum*, pixels from the HSI can be selected (Figure 4.6(a)) and its spectrum be shown pressing 'Spectrum' (Figure 4.6(b)). This is done plotting the value of that pixel for each different wavelength. A variation may be included which allows the user to select an area instead of single pixels, then, the value assigned for each wavelength would be the average of the pixels in the area.



From 460 to 820 step 40

460 nm
480 nm
500 nm
520 nm
540 nm
560 nm
580 nm
600 nm
620 nm
640 nm
660 nm
680 nm
700 nm
720 nm

Power % 50%

Triggering mode Software

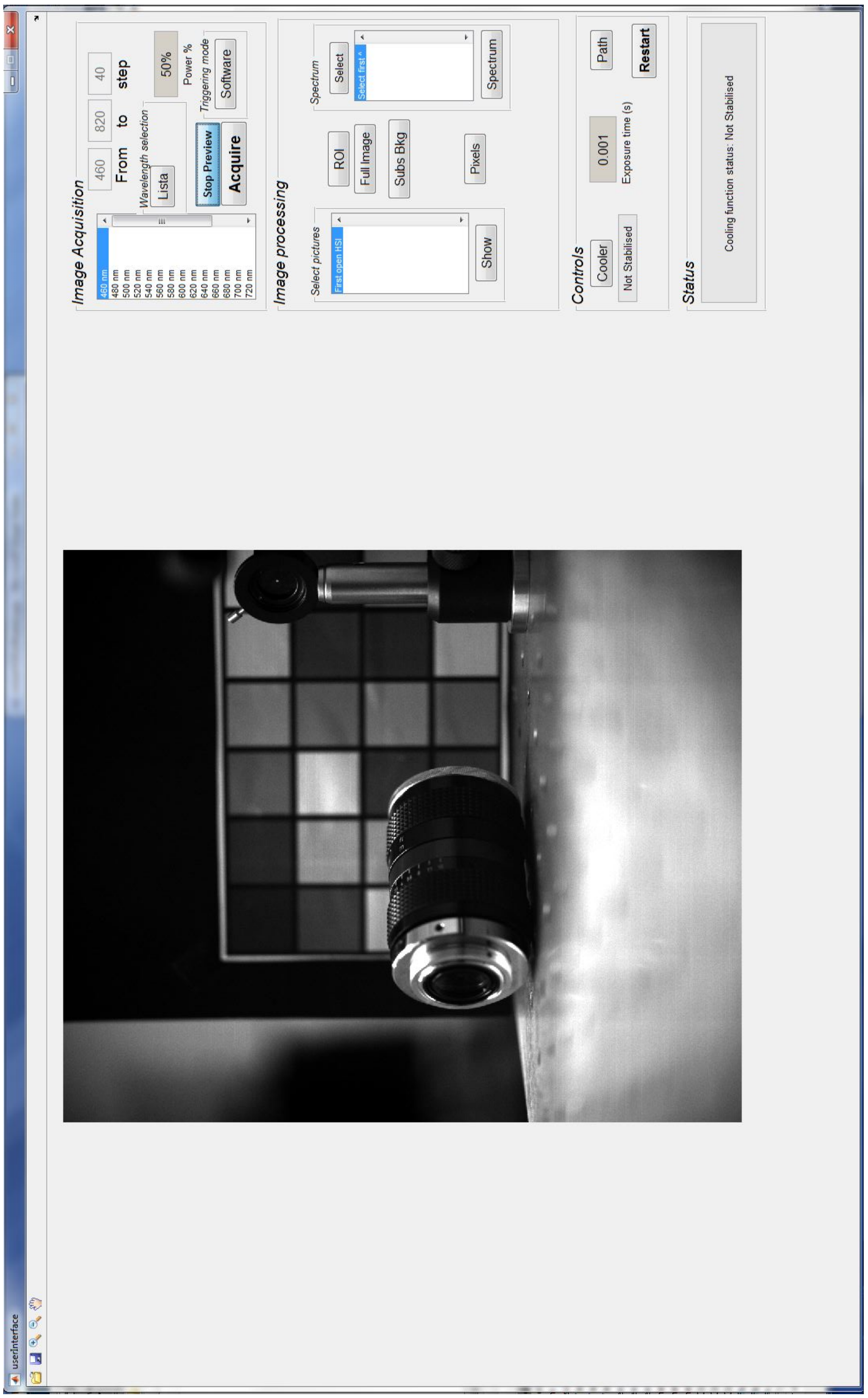
First open HSI

Select first

Exposure time (s) 0.001

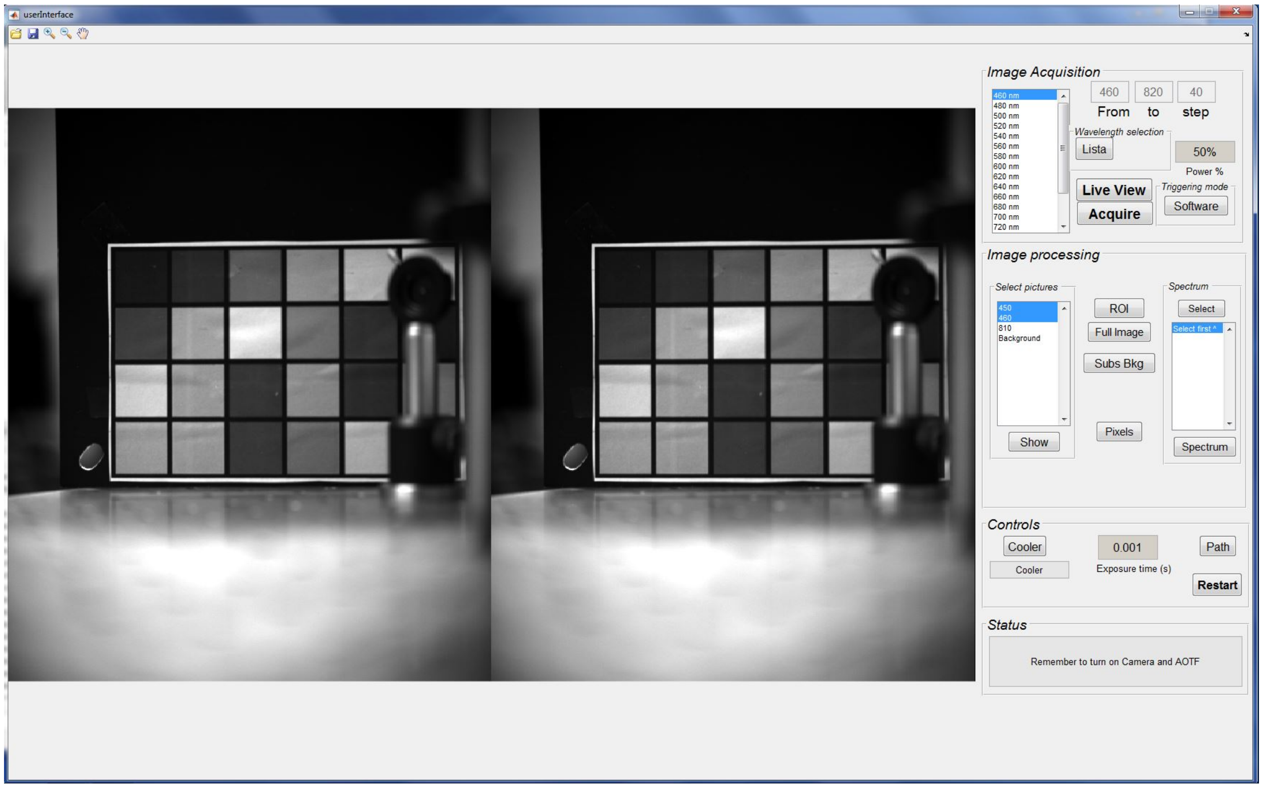
Remember to turn on Camera and AOTF

Figure 4.1: View of the App right after openning.



Cooling function status: Not Stabilised

Figure 4.2: Displaying video: LiveView mode.



460 From 820 to 40 step

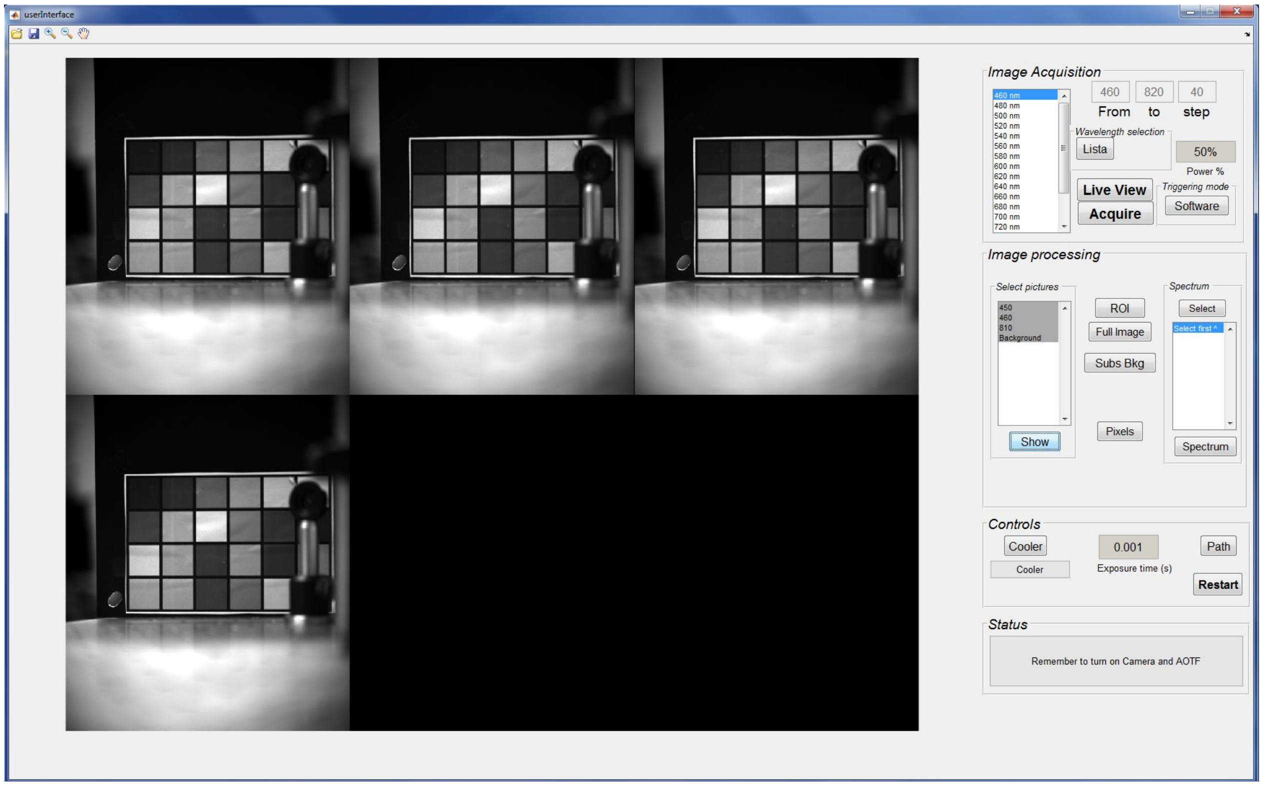
50% Power %

Triggering mode: Software

0.001 Exposure time (s)

Remember to turn on Camera and AOTF

(a)



460 From 820 to 40 step

50% Power %

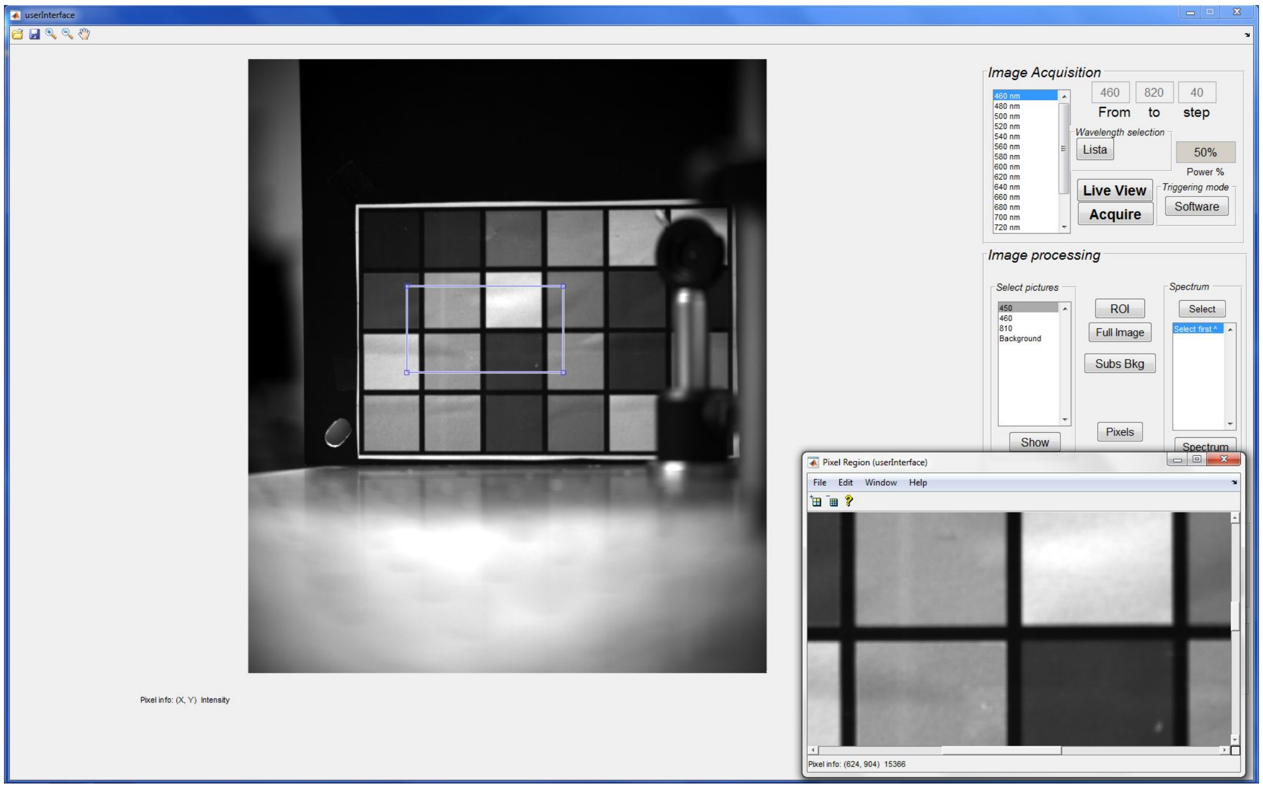
Triggering mode: Software

0.001 Exposure time (s)

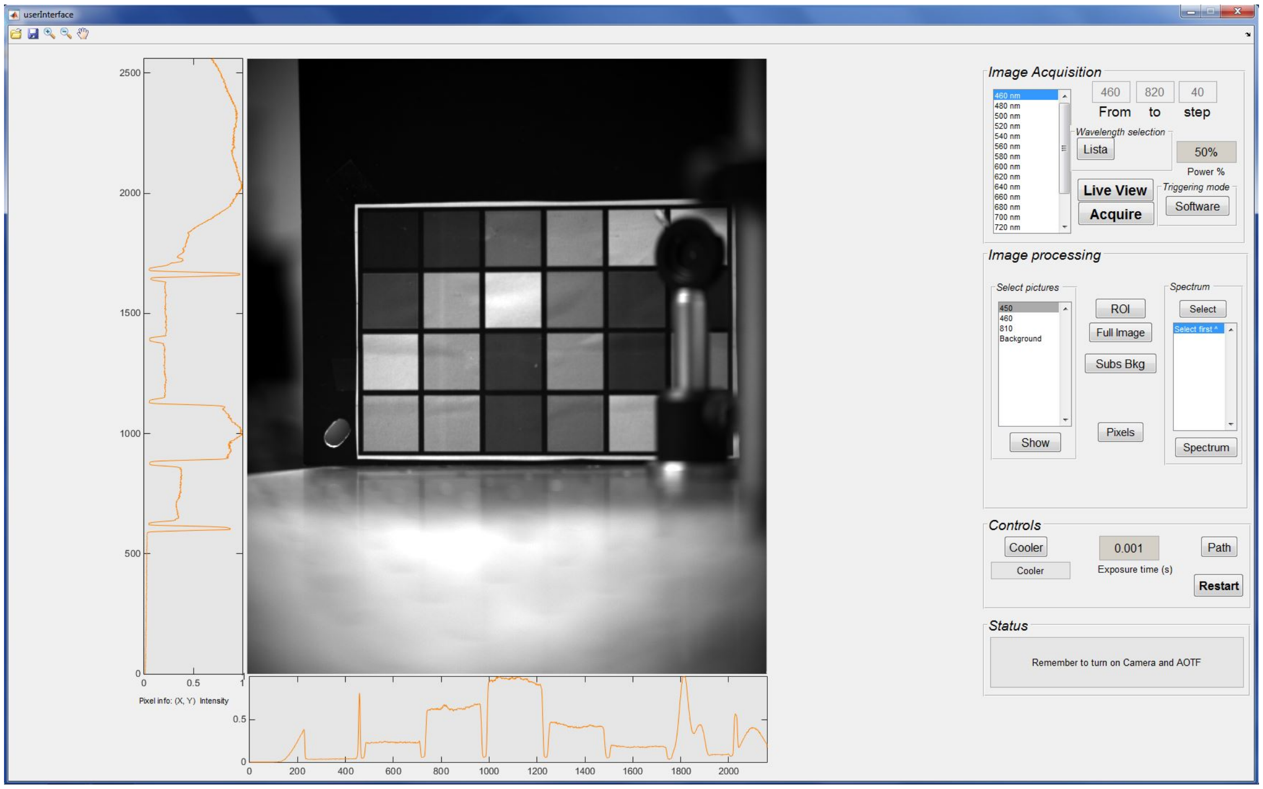
Remember to turn on Camera and AOTF

(b)

Figure 4.3: (a) Result of selecting three wavelenghts and pressing Acquire. The two frames selected in 'Select pictures' are shown. (b) Now, the four frames are selected, so they are all shown.

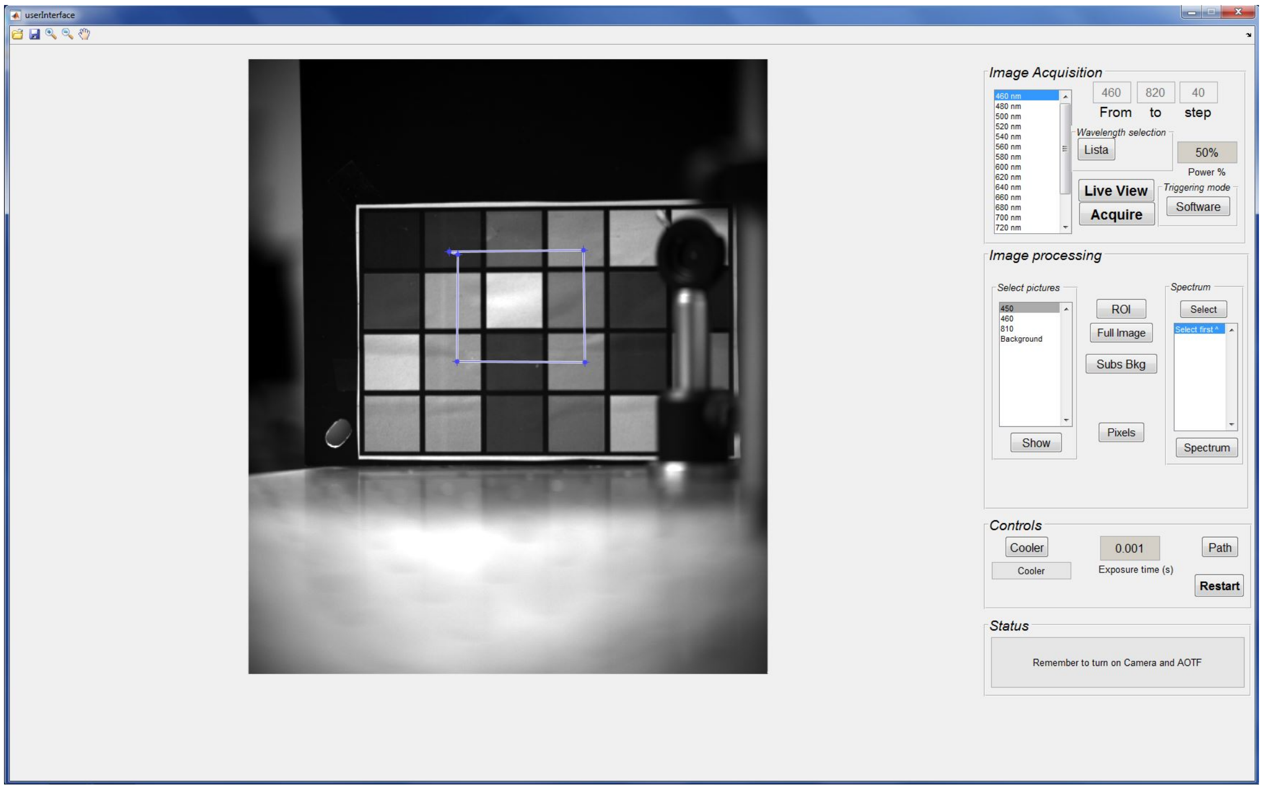


(a)



(b)

Figure 4.4: (a) Pressing Pixels allows to take a closer look to any desired area. (b) If there is only one frame shown, clicking on a pixel produces Histograms to appear.



From 460 to 820 step 40

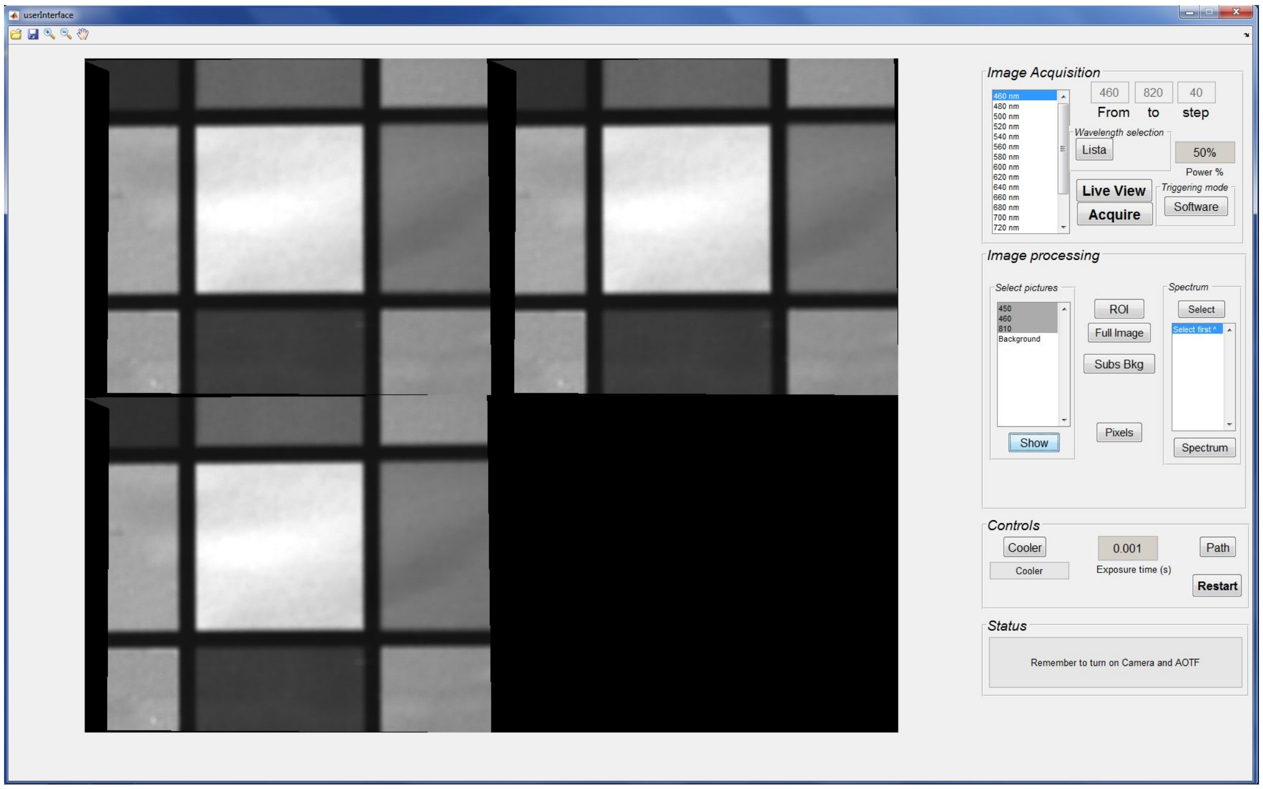
Power % 50%

Triggering mode: Software

Exposure time (s) 0.001

Remember to turn on Camera and AOTF

(a)



From 460 to 820 step 40

Power % 50%

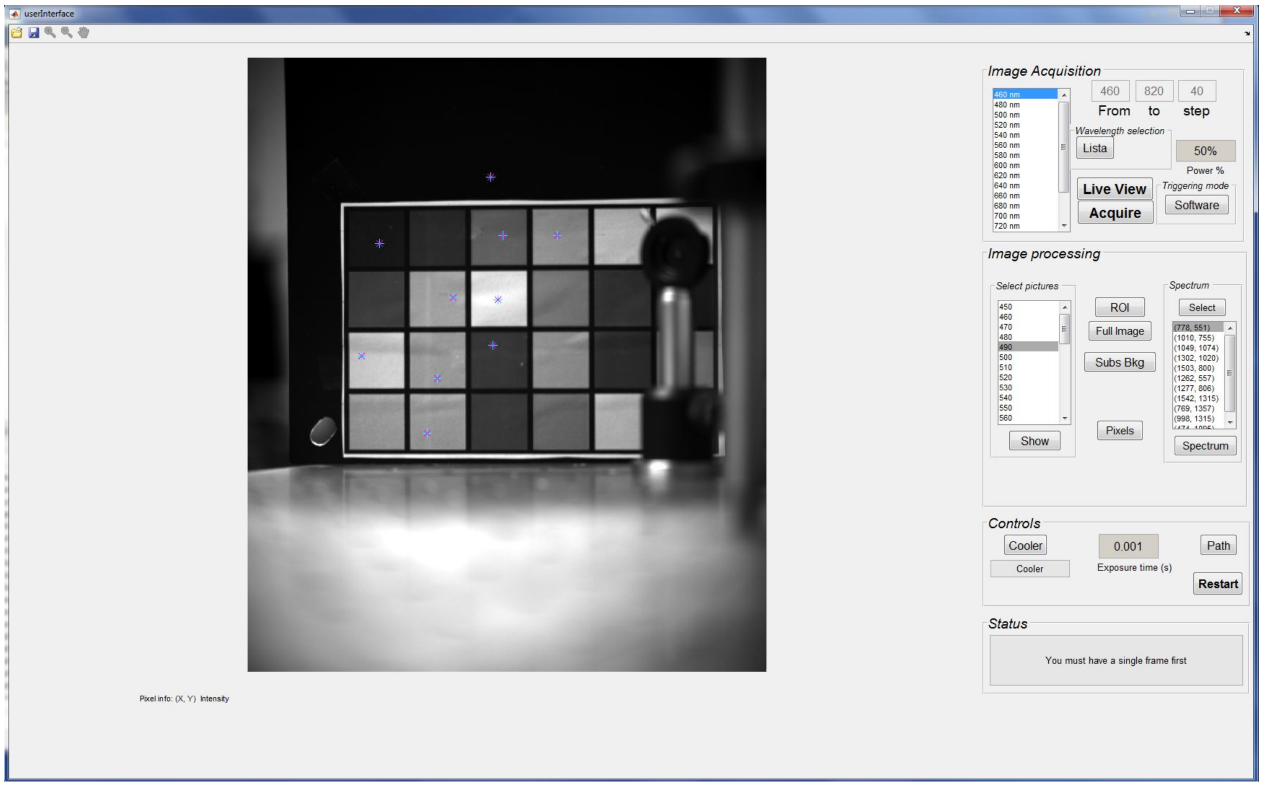
Triggering mode: Software

Exposure time (s) 0.001

Remember to turn on Camera and AOTF

(b)

Figure 4.5: (a) A Region Of Interest is selected. (b) Result of applying the ROI.



460 820 40

From to step

50%

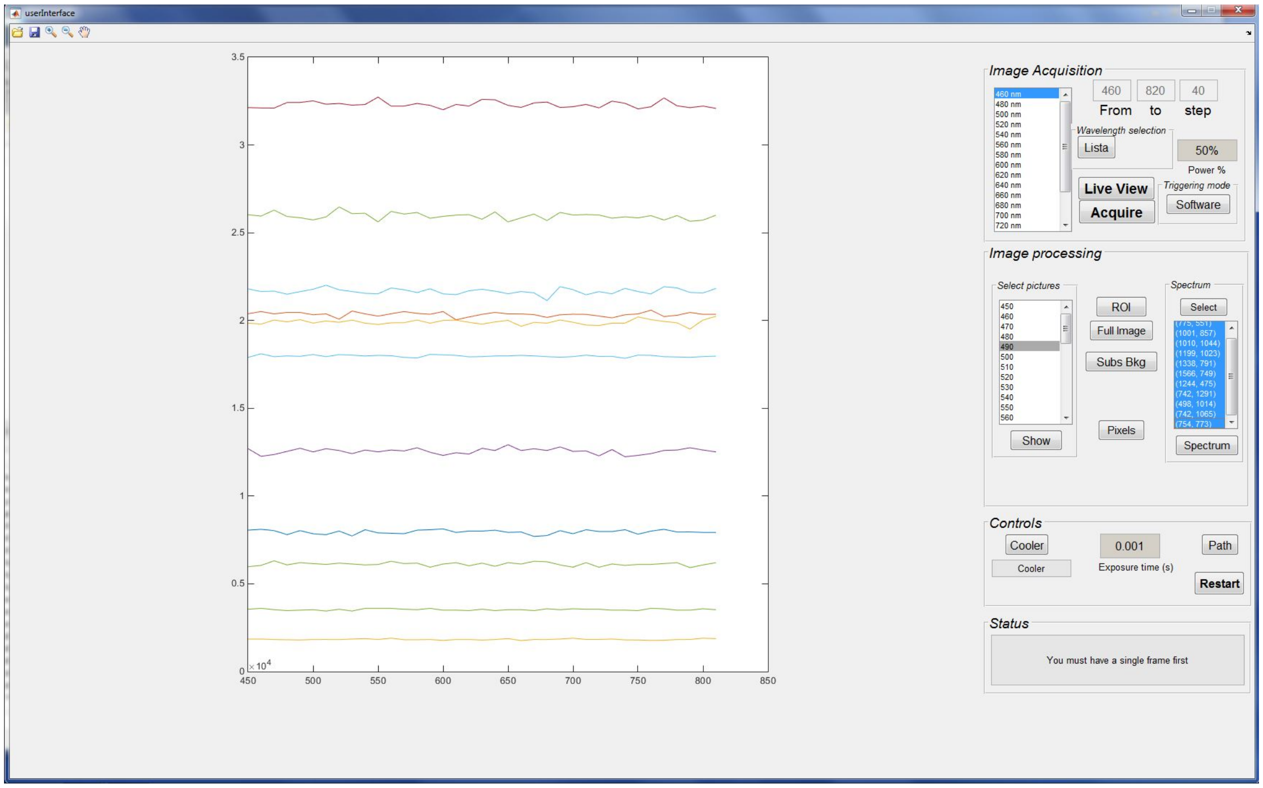
Power %

Triggering mode: Software

0.001 Exposure time (s)

You must have a single frame first

(a)



460 820 40

From to step

50%

Power %

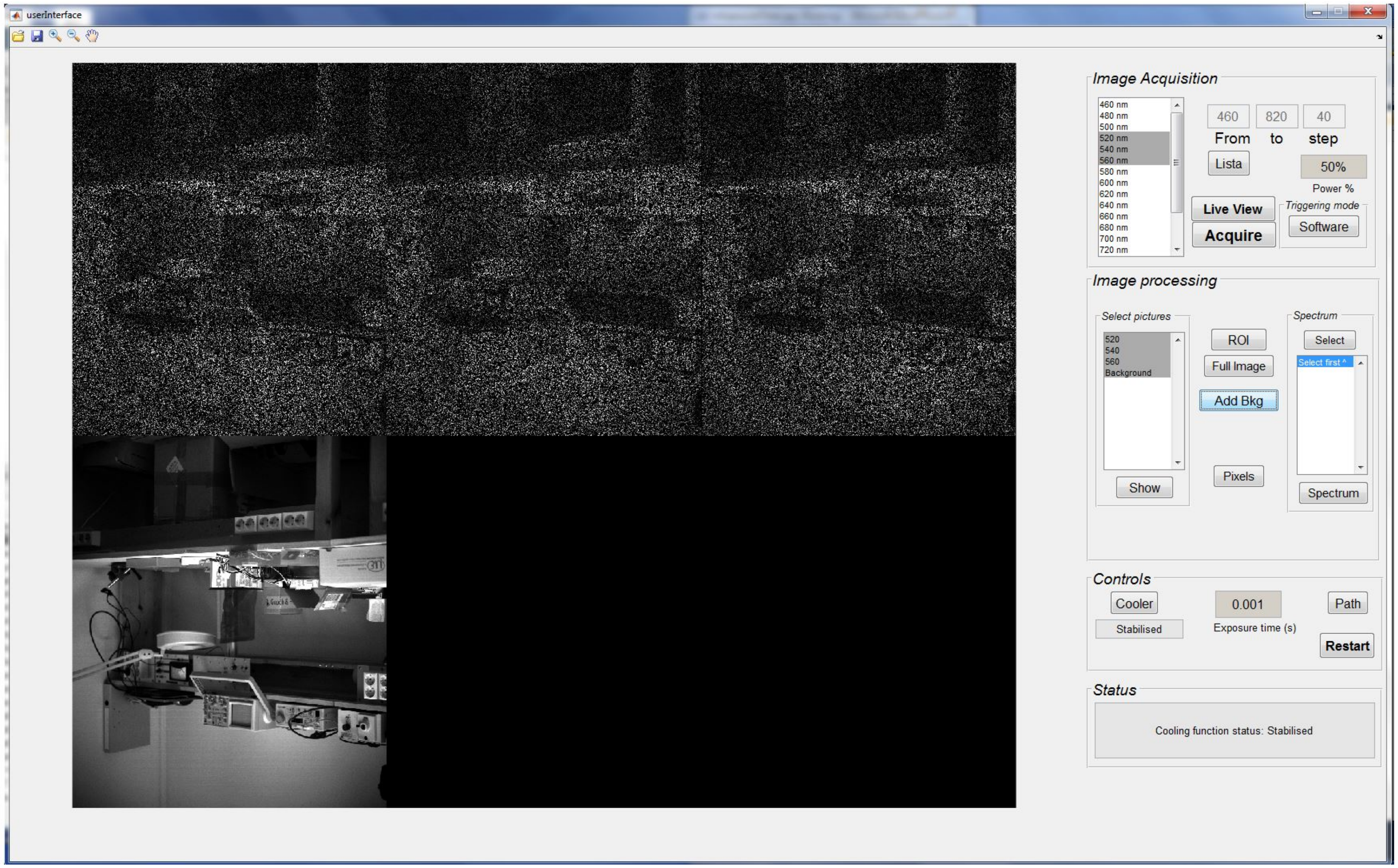
Triggering mode: Software

0.001 Exposure time (s)

You must have a single frame first

(b)

Figure 4.6: (a) Some pixels can be selected by clicking on "Spectrum: Select". (b) Among the pixels selected before, a subset can be chosen and its spectrum plotted.



From 460 to 820 step 40

Power % 50%

Triggering mode Software

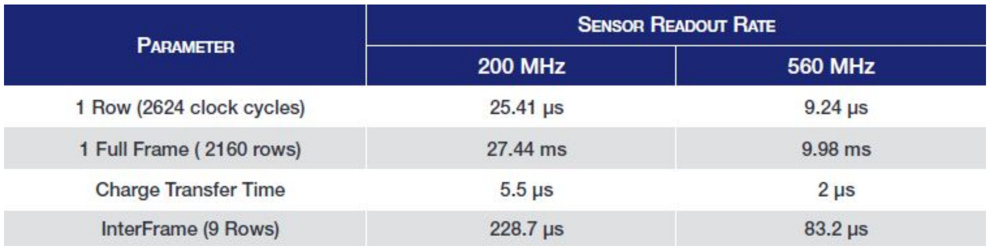
Select pictures: 520, 540, 560, Background

Cooler: Stabilised

Exposure time (s) 0.001

Cooling function status: Stabilised

Figure 4.7: The background frame can be substracted (and added again). (As these are not really HSI, because the AOTF is not working, the three frames are almost equal to the background image, thus, they are lost after substracting.



PARAMETER	SENSOR READOUT RATE	
	200 MHz	560 MHz
1 Row (2624 clock cycles)	25.41 µs	9.24 µs
1 Full Frame (2160 rows)	27.44 ms	9.98 ms
Charge Transfer Time	5.5 µs	2 µs
InterFrame (9 Rows)	228.7 µs	83.2 µs

Figure 4.8: Timing parameters based on sensor clock speed for Global Shutter mode.

4.2 Synchronization

In order to get a hyperspectral image, the camera and RF driver must work synchronously. In particular, an image should not be taken before the AOTF is selecting the correct wavelength. In a similar way, after an image is acquired, the filter should change its frequency response to select the following wavelength to conform the HSI. For that purpose, two alternatives are proposed in this section. Unfortunately, none of them could be fully implemented, due to problems with the RF driver (already described in 3.2.1), which was not operative for most of the developing time of the system. In particular, functions in C to communicate with the RF driver must be written. Appendix C presents a *starting point* to develop this software, which uses the protocol COM RS232.

4.2.1 Software triggering

In this mode the camera and software are in a high state of readiness and can react extremely quickly to a trigger event issued via software. Figure 4.8 presents the values of timing parameters used in the following description (InterFrame time, Frame readout time). They depend on Sensor Clock Speed and are also valid for the next section. The process involved when the systems is in Software triggering mode is as follows:

1. Camera is prepared for acquisition: buffers are queued.
2. Inmediately-executed comand is sent to the RF to select a wavelength.
3. Loop:
    (a) Software trigger is sent to the camera.

   (b) Exposure takes place. Software must wait while this happens: `Sleep(exposure_time)`.

   (c) Inmediateley-executed comand is sent to the RF to change wavelength. At the same time the camera is reading the new frame from the sensor.

   (d) Wait: `Sleep(duration)` where `duration` should be, at least, 1 InterFrame time plus 1 Frame readout time.

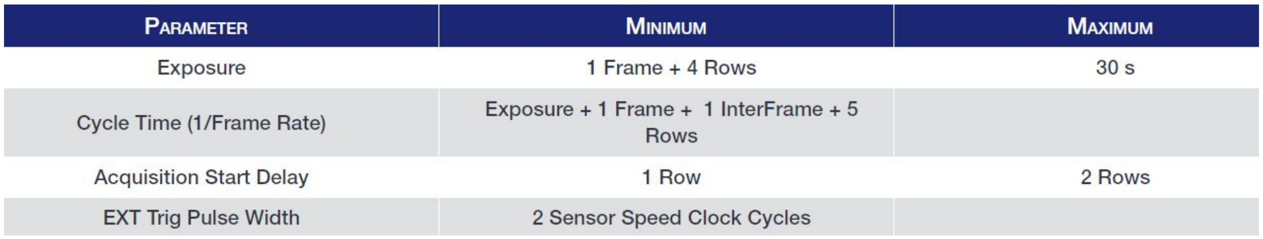
   (e) Back to step (a).

4. Acquisition is finished, buffers are read.

Figure 4.9 presents the theoretical performance achieved by the camera when working in this mode. It can be seen there that the exposure time is constrained to be higher than $1Frame + 4Rows = 10.017$ ms. While Cycle Time, which is the time since an acquisition starts until the next one does, is at least, $MinCycleTime = Exposure + 1Frame + 1Interframe + 5Rows = 20.126$ ms (taking the minimum exposure time possible). Hence, the maximum FrameRate that could be achieved, for full frame size, is: $MaxFrameRate = 1/MinCycleTime = 49.68$. With smaller frame sizes faster acquisition can be obtained.

In the current implementation, the communication with the RF driver is skipped. Therefore, as shown in Appendix A.2.3, line 254, its necessary that the application waits between sending a Software trigger and the next one: otherwise, as the camera is not ready to start a new acquisition, the trigger events sent while an acquisition is running are missed and the number of frames acquired will be lower than expected. So far, implementing a wait time by: `Sleep(CycleTime)`, with Cycle Time as described in Figure 4.9, leads to ideal performance. Specifically, 50 frames are acquired in 1.02 seconds, using the minimum exposure time possible, although it takes around 0.8 seconds more to have them available in the GUI. This delay was mentioned in the former Section and its caused by different operations: reading frames from buffers, interface with Matlab and showing them in the axis.

Once the communication with RF is included is important to consider the following. First, the filter must not change its frequency response while the exposure is going, that is why (b) states 'Software must wait while this happens: `Sleep(exposure_time)`'. Second, once the exposure wait is over, the filter can update its frequency response, i. e. a comand can be sent to the RF driver. The updating process takes around 0.2 ms as claimed by Gooch&Housego (2014b).



Parameter	Minimum	Maximum
Exposure	1 Frame + 4 Rows	30 s
Cycle Time (1/Frame Rate)	Exposure + 1 Frame + 1 InterFrame + 5 Rows	
Acquisition Start Delay	1 Row	2 Rows
EXT Trig Pulse Width	2 Sensor Speed Clock Cycles	

Figure 4.9: Software triggering timing parameters.

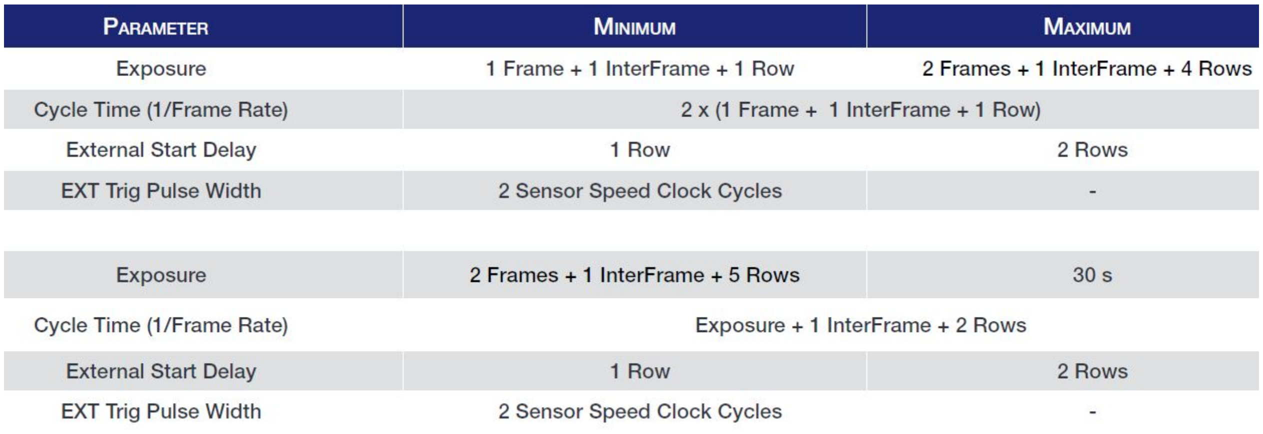
While it happens, the camera is reading a frame from the sensor: $1Frame = 9.98$ ms. As this is a much longer operation, by its end, the AOTF filter should be already selecting the next wavelength. Hence, after a wait of `Sleep(1 Frame + 1 Interframe + 5 Rows)`, which completes the Cycle Time (as it has already waited: Exposure time), another trigger could be sent.

Performance should be almost the same as the one achieved previously, as the process is basically the same. The difference is that now the waiting time is splitted in two different waits (one as long as the exposure, and the other similar to the readout time of the sensor) and including instructions to control the RF driver in between, which should be processed two or three order of magnitude faster so, virtually, they have no impact. However, to make sure that the trigger events are not missed, the waiting times could be made a little bit longer: sacrifying performance to make sure that the HSI are correctly acquired.

4.2.2 Hardware triggering

Hardware triggering has not been implemented, but it has some features that might make it useful for the system in the future. It allows the camera to work on *Overlap mode* in which the camera can accept a trigger to begin the next exposure prior to the signal frame readout from the sensor completing. It does not result in higher frame rates, i. e. the maximum frame rate that can be achieved is 49 frames per second for full size images (2560 x 2160 pixels), but exposure times can be longer for a given frame rate, for example, to get 49 frames in a second, the exposure time can be up to $2Frames + 1Interframe + 4Rows = 20.08$ ms. This is almost two times the maximum exposure time that Software triggering allows to work at 49 frames per second. Hence, an exposure can represent a higher percentage of the Cycle Time. It provides a 100% sensor duty cycle.

Figure 4.10 presents timing parameters of the camera when working in this mode. It can be



Parameter	Minimum	Maximum
Exposure	1 Frame + 1 InterFrame + 1 Row	2 Frames + 1 InterFrame + 4 Rows
Cycle Time (1/Frame Rate)	2 x (1 Frame + 1 InterFrame + 1 Row)	
External Start Delay	1 Row	2 Rows
EXT Trig Pulse Width	2 Sensor Speed Clock Cycles	-
Exposure	2 Frames + 1 InterFrame + 5 Rows	30 s
Cycle Time (1/Frame Rate)	Exposure + 1 InterFrame + 2 Rows	
External Start Delay	1 Row	2 Rows
EXT Trig Pulse Width	2 Sensor Speed Clock Cycles	-

Figure 4.10: Global Shutter External Triggering Timing Parameters (Overlap On) - Cycle Time Dependent on Exposure. For exposure times longer than 2 Frames + 1 Interframe + 5 Rows, the cycle time increases with exposure.

seen there that, for exposure times longer than $2Frames + 1Interframe + 5Rows = 20.089$ ms, the Cycle Time, and therefore, the frame rate, depend on the exposure. But for values between 10.07 and 20.089 of the exposure time, the same, and optimal frame rate (49 frames per second) is achieved.

Ingvaldsen (2012) showed how Hardware triggering can be implemented for a similar system which used the same AOTF driver, but different camera. All the instructions must be sent to the driver prior to acquisition, which stores them. Then, for every incoming pulse, the driver executes the next instruction stored changing the frequency response of the filter. The Multi I/O timing cable pin outs (5V TTL) of the camera, in particular, input External Trigger (pin 7) and outputs ARM (pin 1) and FIRE (pin 4) should be used, together with Camera in (pin 1) and Camera Out (pin 2) of the driver. The input pulse of the camera External Trigger should be at least of duration 2 sensor clocks: $PulseDuration = 2 * \frac{1}{ClockFrequency} = 2 * 1/(560MHz) = 3.57$ ns, if working at 560 MHz. Output ARM from the camera indicates when the camera is ready to accept an incoming trigger pulse. No information about the duration of the input pulse Camera in is provided by Gooch&Hosego, it may be necessary to contact them. The process should go as follows:

1. Camera is prepared for acquisition: buffers are queued.

2. Instructions to be stored are sent to the RF, to indicating the sequence of wavelengths.

3. Loop:

   (a) Pulse is sent to the RF driver: pin Camera in, an instruction is executed.

   (b) After a programmed time, the RF driver sends ouput pulse: pin Camera out.

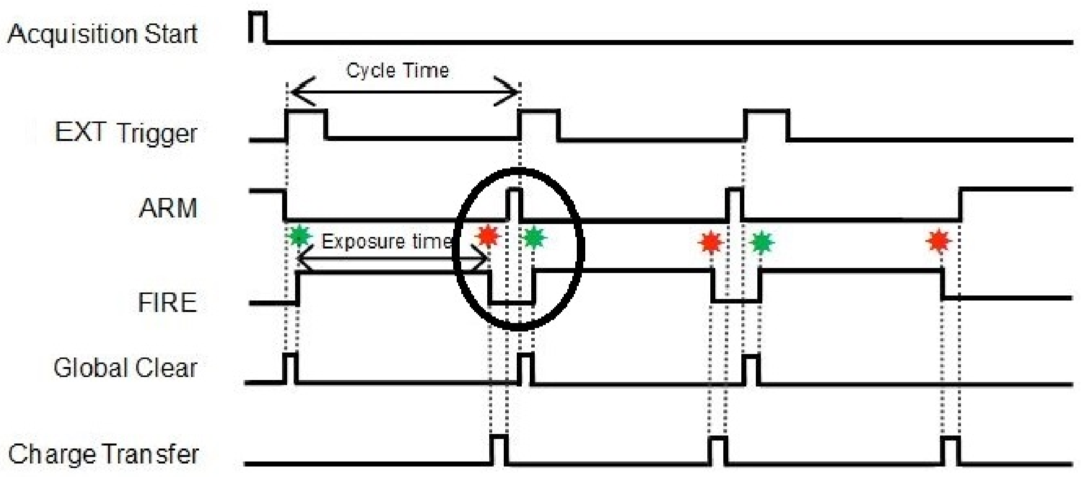
   (c) When output pulse Camera out is received from the RF driver, if ARM is active (it should be), External trigger pulse is sent to camera. Otherwise, wait until ARM is active and then send External trigger.

   (d) Exposure starts. When it ends, output FIRE presents a negative edge, this negative edge must trigger (a).

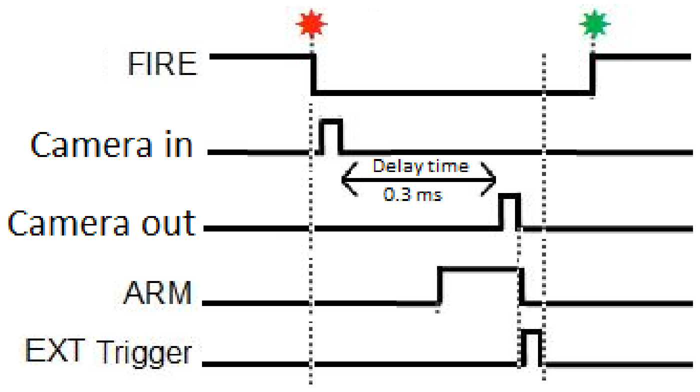
4. Acquisition is finished, buffers are read.

See that step (d) takes way longer time than the other three. Specifically, the exposure can last from a minimum of 10.072 ms up to 30 s, while (a), (b) and (c) should, ideally, take place in the remaining time of the Cycle Time, this is $1 Interframe + 2 Rows = 101.68$ μs. This is not possible since the minimum programmable time between receiving a pulse in Camera in, and sending a pulse Camera out allowed by the RF driver is 300 μs. This value should be used: it almost ensures that the camera will be ready (ARM active) to receive a new trigger, and it gives enough time to the filter to change before starting a new acquisition.

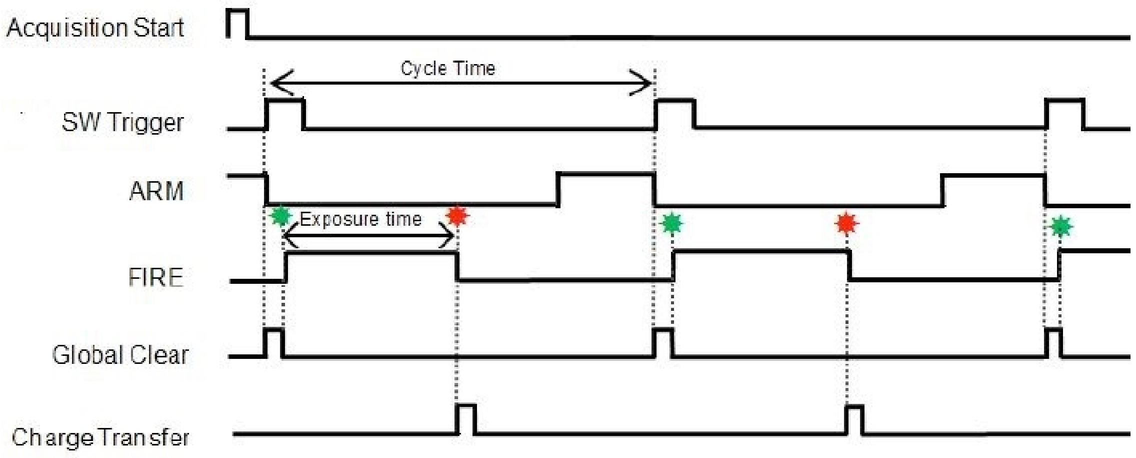
Figure 4.11 shows a timing diagram of the camera working in Hardware triggering mode (a), a zoom showing the signals used to synchronize the camera with the AOTF (b), and the timing diagram of the camera in Software triggering mode (c), useful to see the Cycle that the camera follows on each mode, in particular, the different fraction of the Cycle Time that the exposure time represents. The signals global clear and charge transfer are used by the camera to drain charge from every pixel before an acquisition and to transfer charge from the pixel to the measurement node and effectively end the exposure, respectively.



(a) Timing diagram of the camera in Hardware triggering.



(b) Timing diagram of the system.



(c) Timing diagram of the camera in Software triggering.

Figure 4.11: (a) Timing diagram of the camera when working in Hardware triggering (Overlap mode on). (b) Zoom of the circled area in (a), showing the signals used to synchronize the system. (c) Timing diagram of the camera working in Software triggering mode.

Chapter 5

System characterization and experiments

In this Chapter four tests are presented to analyze the performance of the system. Since the system was never working, due to the problems with the AOTF driver that have already commented, some of these tests could never be done, while others were carried out using a different system to illustrate the procedure.

5.1 Spectral analysis

The first of tests to be performed is a spectral analysis of the system. White standard or a white source of light can be used as a target, and be placed in front of the front lens. It is important that the target spectrum is known, at least, around the band to be analyzed. A spectrometer situated at the output aperture and connected to a computer collects spectral data for different wavelengths in the range: 400-900 nm. The working range claimed by the filter manufacturer is 450-800, but an extended range is selected to figure out which the system limits are. See that this test is basically the same as that presented in 3.2, but now using the whole system and not just the filter.

A data set is composed by the measurements of the spectrometer consisting of a different output spectrum (function of λ), for every tested frequency (selected on the RF driver): $Y(f_{AOTF}, \lambda)$. The same spectrometer may be used to analyze the spectrum of the light emitted by the target, which is the input of the system $X(\lambda)$. Once the spectral data is available we have, for any system: $Y(\lambda) = H(\lambda) * X(\lambda)$, where $H(\lambda)$ is the frequency response of the system.

Therefore, $H(f_{AOTF}, \lambda)$ can be calculated, which is the frequency response of the system for a selected frequency as:

$$H(f_{AOTF}, \lambda) = \frac{Y(f_{AOTF}, \lambda)}{X(\lambda)} \tag{5.1}$$

Then, for every fixed f_{AOTF}, the λ_{max} maximizing $H(f_{AOTF}, \lambda)$ indicates the wavelength around which the filter is centered for the different frequencies used. A graph can be constructed then, plotting λ vs. f, similar to that shown in 3.6(a). Also, λ_a and λ_b can be found, for every fixed f_{AOTF}, satisfying:

$$H(f_{AOTF}, \lambda_a) = \frac{1}{2} max\{H(f_{AOTF}, \lambda)\} = \frac{1}{2} H(f_{AOTF}, \lambda_{max}) \tag{5.2}$$

$$H(f_{AOTF}, \lambda_a) = H(f_{AOTF}, \lambda_b) \tag{5.3}$$

$$\lambda_a < \lambda_b \tag{5.4}$$

And hence, providing the full width at half maximum (FWHM) bandwidth of the system, for every different f_{AOTF}, as:

$$FWHM(f_{AOTF}) = \lambda_b(f_{AOTF}) - \lambda_a(f_{AOTF}) \tag{5.5}$$

A graph can be built plotting $FWHM(f_{AOTF})$ against f_{AOTF} or λ, as that shown in 3.6(b), which gives us information about the spectral width of the filter for every selected wavelength.

5.2 Power optimization

A further step on calibration of the system consists of optimizing the power selected at the RF driver, and therefore, the amplitude of the acoustic wave applied to the crystal, to maximize the diffraction efficiency. This procedure is presented in Joan Vila-Frances (2010). The diffraction efficiency is the relation between the RF output intensity and the light source intensity for each different wavelength. To have access to these values an spectrometer can be placed at the output of the back lens, in a similar way as in test 5.1. But it can also carried out using the camera, always with the same integration time, and see how image intensity varies when changing RF

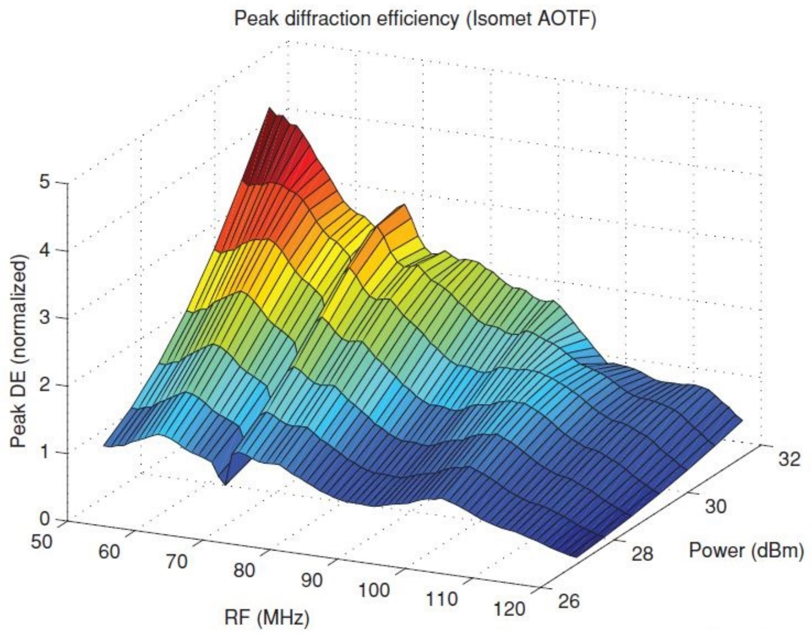


Figure 5.1: Diffraction efficiency as a function of driving signal characteristics. Taken from Joan Vila-Frances (2010).

power. Then, the filter is configured to work over all its frequency range, and for each frequency, many different powers are used.

In fact, due to similiarities between this and the test explained in 5.1 (they even share part of the data set if the spectrometer is used instead of the camera), it is recommendable to carry them in parallel. By doing so, dependance of the FWHM bandwidth with power could be studied, although RF power should not affect it very much.

As a result, the diffraction efficiency as a function of frequency (or wavelength) and power is obtained, similar to that presented in the mentioned article and shown in Figure 5.1. It can be seen in that particular case, that for frequencies between 50 and 95 MHz, the highest efficiency corresponds to maximum power. However, for frequencies in the range 95-120 MHz this is not true anymore, and the optimal power is lower than the maximum power. This data set should be stored, then, before starting every exposure, the optimal signal power is selected as that matching the maximum of the diffraction efficiency surface at the given frequency.

5.3 Wavelength dependent resolution

The second test evaluates how the resolution varies among the frames taken at different wavelengths. For that purpose, the camera is focused at a certain wavelength, for instance $\lambda = 650$ nm, and a HSI of the target is taken in the range 400-900 nm. Then, contrast of every frame is calculated, obtaining: $c(\lambda)$. Which can then be plot as a function of λ. Its result is a curve between 0 and 1, whose maximum is placed at the wavelength for which the camera was focused and that decreases with distance to the focusing wavelength. Such a graph provides information about how Chromatic aberrations described in Section 2.3.2 are affecting the system. In particular, it shows how defocused each frame is depending on the wavelength at which is taken, and therefore, how much longitudinal aberrations are shifting the plane of best image sharpness.

Contrast of an image can be calculated in several ways. All them include getting a constrast image an averaging over the pixels of this image. To obtain a constrast image, every pixel p_0 within an image is substituted by a function of:

$$d_i = \frac{|p_0 - p_i|}{|p_0 + p_i|} \quad \text{for,} \quad i = 1, 2, ..., 8. \tag{5.6}$$

where $[p_1 p_2 \dots p_8]$ are the eight neighbour pixels of p_0. This function of d_i could be, for instance: the maximum, $p_0 = max\{d_i\}$; or the average, $p_0 = \frac{1}{8}\sum_{i=1}^{8} d_i$.

5.4 Grid aberrations

The third test consists of imaging a grid pattern to evaluate which aberrations are present in the system. In particular, it is checked if there is a lateral shifting of the grid for different wavelengths, important because it destroys the spacial-spectral correlation. As well as, if any of the radial distortions corrupt the grid pattern.

Although this test could not be performed with our hyperspectral imaging system either, we had access to a system provided by Gooch&Hosego and we carried out the test, which will be used to illustrate how it can be done. Matlab code used to analyze data obtained with this test is shown in Appendix D.1. The system was made of a camera HSI-440C with 8 mm lens, a LED lamp LR45-90, Optometron, custom made combination of white LEDs and 850 nm LEDs. A polarizer was used to cancel out specular reflection, diffusor and 19 cm lens: where the object was placed.

To acquire the required data, a piece of graph paper from a notebook was carefully placed on a metal plate, as even and unwrinkled as possible. It was then illuminated and imaged using exposure times that avoid saturation to appear in the images. Using the same exposure times, two hyperspectral images were taken: one of a gray standard sample of a size similar to that of the grid, and another with the camera objective completely covered to get image of the dark current of the sensor. The dark HSI is then substracted from both the grid and gray standard image cubes. Then, these two can be combined to create a reflectance image, which is the one that will be studied.

To measure if there is any radial distortion present in the system, the reflectance image is modified. First, the beginning of a line in the grid is found, and a straight line is drawn starting there, i. e. the value 0 (black) is assigned to all the pixels in that column or row, depending if it is a vertical or horizontal line. The process is repeated for several lines distributed all over the grid pattern, it is desirable to draw two lines showing the center of the image which should suffer no distortion. It is already possible to visually appreciate how some of the lines are distorted, see Figures 5.3(a) and (b), but in order to provide a measurment, another modification of the data set is done. Now, supposing we are working with a vertical line of the grid whose *associated* straight line has been drawn, 6 pixels on each side of the straight line are considered, and for every row, a value of 0 (completely black) is assigned to that whose value is minimum (the darkest). Besides, the square of the distance from that pixel to the straight line is computed and stored in the 2D array `verticalDistortion` whose size is $N_{wavelengths} \quad x \quad N_{verticalLines}$. Each element (i,k) of `verticalDistortion` corresponds to $Sum_{i,k}$ as shown in Figure 5.2. A similar process is followed for horizontal lines. This is repeated for every straight line drawn in a frame, and for all frames. As a consequence, `verticalDistortion(i,k)` contains the sum of the distances of the computally colored pixels in the frame i, to *their* straight line k. Figure 5.2 illustrates the process.

As a result, two arrays `verticalDistortion` and `horizontalDistortion` are available. The edited images are shown in Figure 5.3. See that the lines drawn with the second technique are not really lines: they are just dots. As long as the resolution is high enough, like in the first

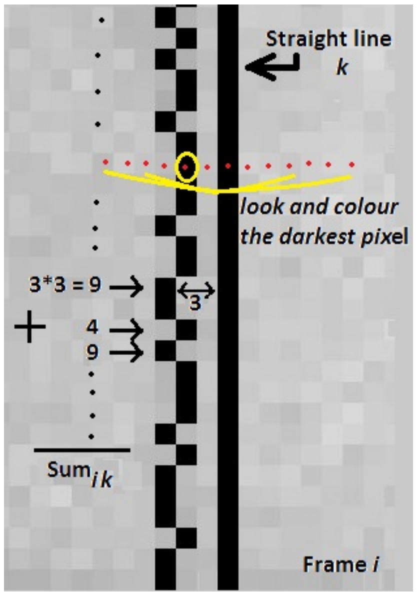


Figure 5.2: Scheme explaining how the lines following the grid are drawn. For each row, 6 pixels on each side of the straigh line (as the red dots) are considered and the darkest is coloured. The square of its distance in pixels to the straight line contributes to the sum: $Sum_{i,k}$.

two cases (a) and (b), they follow the grid pattern pretty accurately, and hence they will provide a good measurement. However, when the image is blurred, like in (c), they do not match the grid. The measurement they provide is too noisy, with an average value way higher than what the result should be. For this, they can not be considered as reliable data. More complex techniques to, computationally, *draw* the pattern could be used obtaining better results. For instance, morphological operations for image processing in Matlab, like *skeleton* or *erode*.

The vectors `verticalDistortion` and `horizontalDistortion` can then be used to analize the performance. Figure 5.4 presents three graphs obtained using them. The first two are built in a similar way. For six different wavelengths, the value of the distortion of each horizontal (or vertical) line is plotted against its position in the image. In particular, against the position (row or column) of its straight line associated. Then they are normalized in a way that a distortion of 1 means no distortion. For (a), horizontal distortion, the center of the grid is around the row 460, therefore, minimum distortion is present there for every wavelength; while for (b), vertical distortion, this happens for the column 500. It can clearly be seen that distortion increases with distance from the center.

On the other hand, (c) presents the average distortion for every frame comparing to the wavelength at which it was taken. The result is first smoothed by a 5-taps low pass filter, each sample is replaced with the average of five samples around it. For this system, there is more horizontal distortion than vertical, in the range in which our measure is reliable. Above 700 nm, the results provided are corrupted and no conclusion can be made out of them, in part because the lamp emited almost no light between 750 and 800 nm.

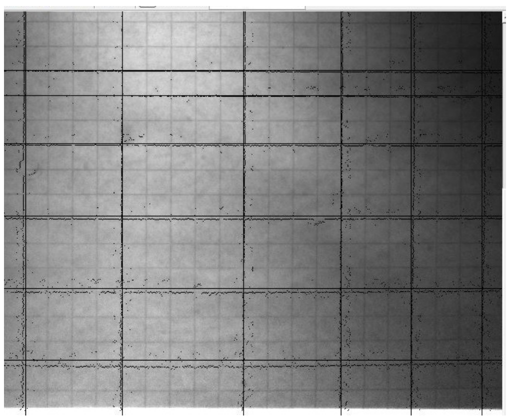
To see if there is any lateral shift, the central column is studied. It should present a constant value of distortion along the spectra as it does not suffer barrel distortion. However, if all the grid is shifted laterally, a higher value of distortion will be obtained: there will be a non zero distance between the straight line drawn, and where the grid line actually is. Figure 5.5 is a plot of the values of distortions of the central column depending on the wavelength. The result is first smoothed by a moving average filter of span 3. Inside the range from 450 nm to 680, it seems to vary. After carefully looking at the images, the shifting looks to be confirmed, however it is no bigger than 1 pixel in the range 450-650 nm, and not bigger than 2 pixels between 650-680. No conclusions can be made above 680.

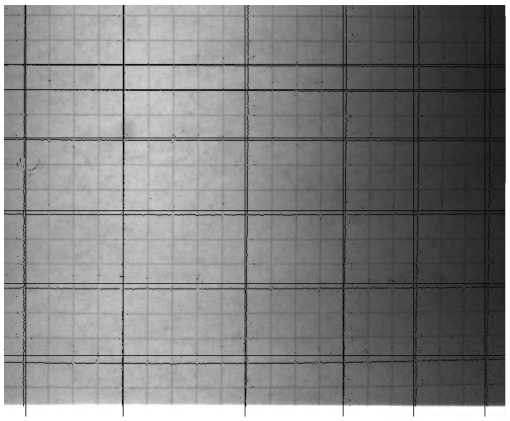
5.5 Color chart spectra

A final test was performed using the hyperspectral imaging system provided by Gooch&Hosego to evaluate how accurately spectral information can be obtained from it. An equivalent test might be performed with the system described in this work.

The set up is the same to that used in 5.4 for the grid test. Again, the Matlab code is available in Appendix D.2. The color chart shown in 5.6 was carefully placed on the metal plate as the grid was for the former test. It was imaged using exposure times that avoid saturation to appear in the images. Using the same exposure times, a hyperspectral image of a gray standard sample and another one measuring the dark current of the sensor were taken. Spectral data using spectrometer (S2000, Ocean Optics) was also acquired for each of the colored squares. However, the color chart surface turned out to be really shiny. Therefore, the measured spectra could be slightly corrupted due to specular reflection, especially for long wavelengths, which are more affected by this phenomenon.

The image cube of the color chart is converted into a reflectance image in Matlab. Then, three regions, each one including the whole square, are defined using the tool `impixelregion`


(a) Frame at 450 nm.


(b) Frame at 608 nm.

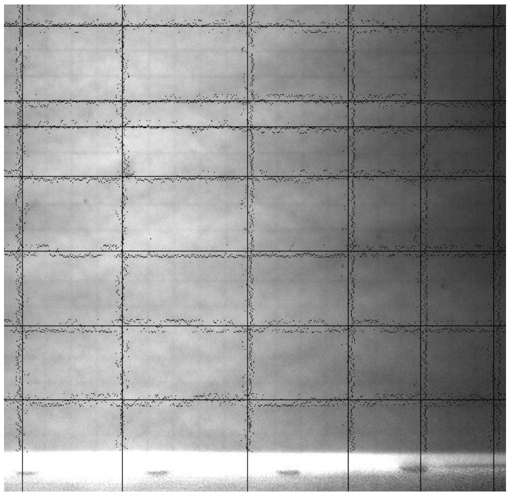
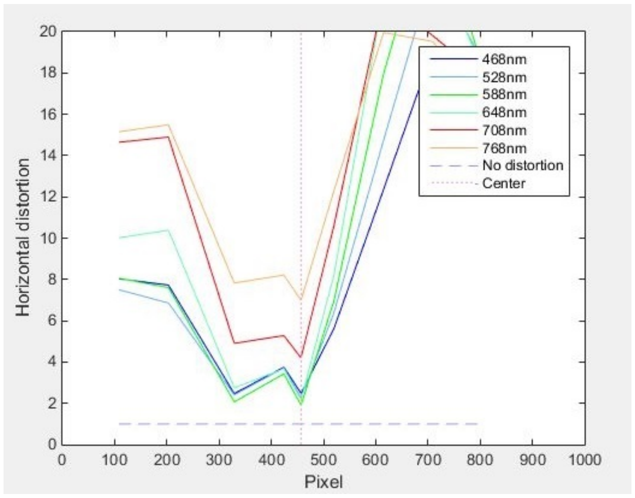
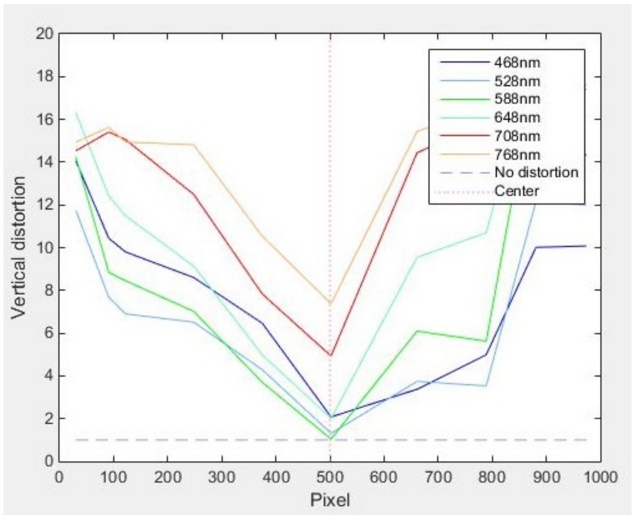

(c) Frame at 768 nm.

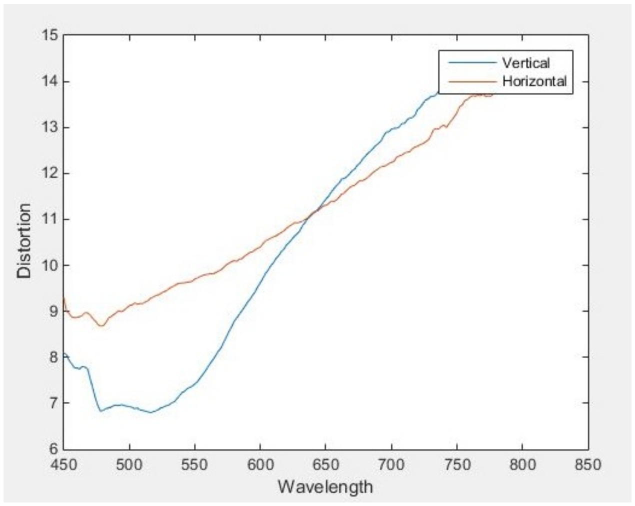
Figure 5.3: (a) (b) (c) are three frames of the image cube corresponding to 450, 608 and 768 nm. Straight lines and minimum dots are shown.



(a) Horizontal distortion vs. row.



(b) Vertical distortion vs. column.



(c) Average frame distortion vs wavelength.

Figure 5.4: (a) Horizontal distortion calculated for all horizontal lines drawn in a image, and plotted depending on which position (which row) each occupies in the image. (b) Vertical distortion, same way. (c) Average distortion within in a frame depending on wavelength.

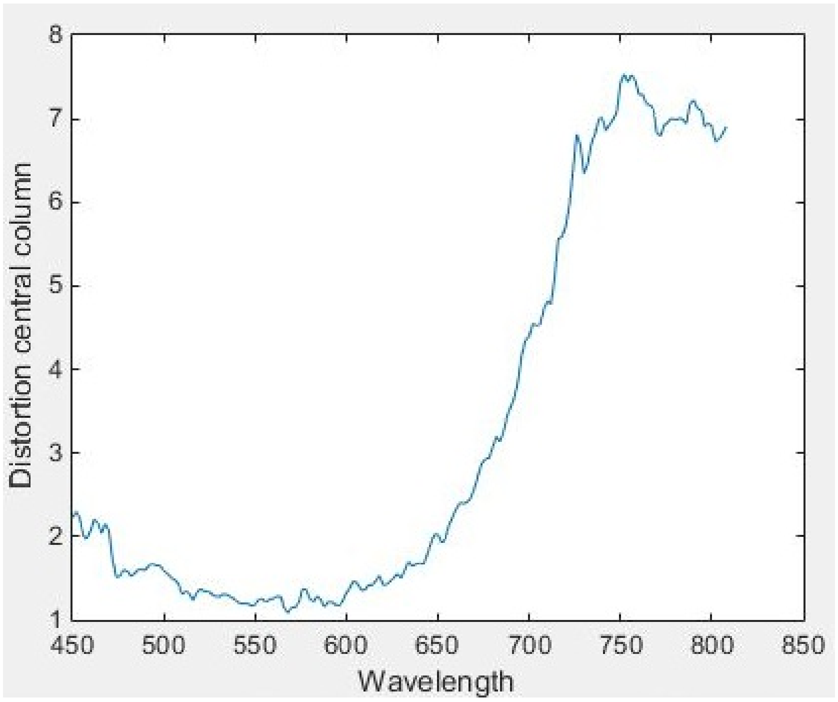


Figure 5.5: Distortion of the central column for different wavelengths.

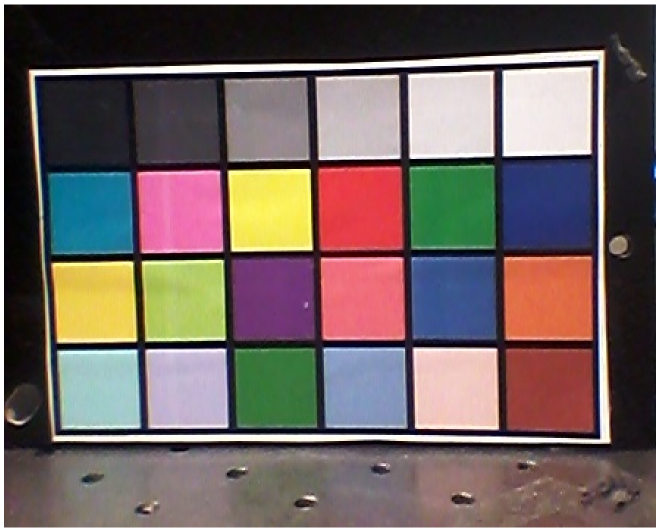
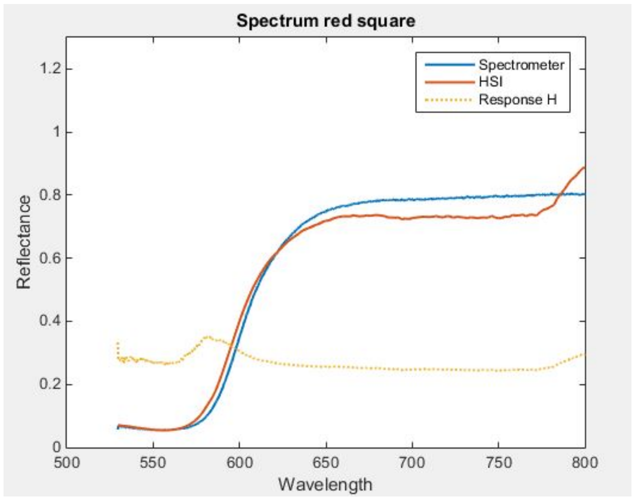


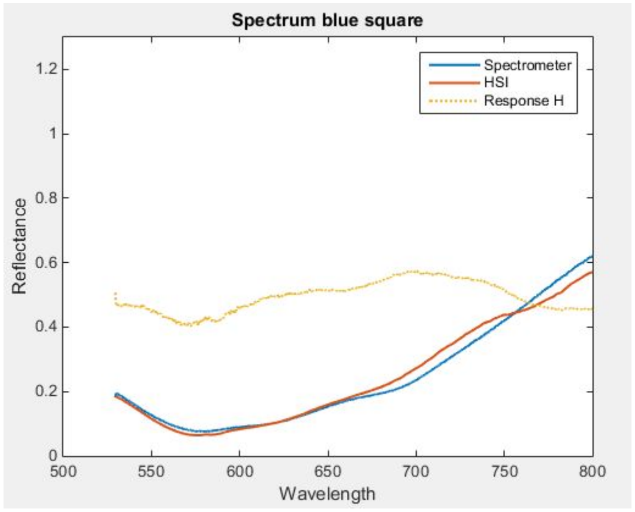
Figure 5.6: Color chart used for the test.

for the squares: red in the second row, blue in the third row and green in the fourth row. The coordinates of these regions make possible to extract them as three different hyperspectral images. Each frame of the three image cubes are then normalized to weight equally all the pixels within the frames. Finally, averaging over all the pixels within a frame produces a vector containing spectral information, which is then compared with that obtained by the spectrometer.

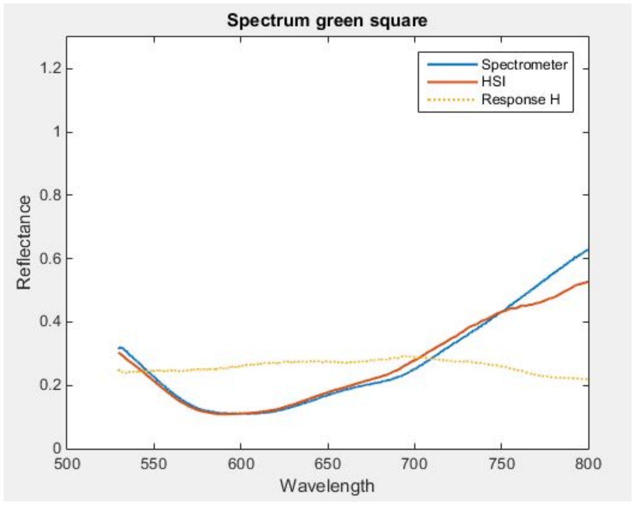
After properly interpolating the data to have equal vector sizes, the graphs shown in Figure 5.7 are obtained. See that the matching between both lines is quite good for all them (5.7(a), (b) and (c)). Discrepancies may be due the commented specular reflection. And the fact that there was not much light between 700 and 800 nm due to the LED lamp spectrum, which is consistent with the fact that they are more relevant above 700 nm. Frequency response of the system $H(\lambda) = Y(\lambda)./X(\lambda)$, considering the spectrometer data as input $X(\lambda)$, and the HSI measurement as output $Y(\lambda)$ is also plotted. By taking the mean of $H(\lambda)$, HSI spectra can adequately be scaled multiplying $Y(\lambda)$ by $1/mean(H(\lambda))$, as it has been done before plotting them. It can be seen that the frequency response is pretty flat for all cases which indicates a good match between the spectral information.



(a)



(b)



(c)

Figure 5.7: (a) (b) and (c) show the spectral information of the red, blue and green squares respectively, as obtained by the spectrometer and the HSI, see legend. In yellow, frequency response, H, of the system is shown for all them.

Chapter 6

Discussion

A hyperspectral imaging system has been designed and built within this work, considering both the hardware and software parts of the system. A high perfomance design has been achieved, although it is not yet operative. The optical design is based on V. B. Voloshinov and Yukhnevich (2012), and it uses an AOTF with improved reduction of optical side lobes, as stated by C Stedham (2008). Its main features are the high spatial resolution it offers, and, its fast speed acquiring images. Comparing with the work carried out in Ingvaldsen (2012), where it was affirmed that the system could acquire 512 x 512 pixels images at a frame rate of 17 frames per second, this system should get 2560 x 2160 pixels images at 49 frames per second. Besides, if hardware triggering is implemented, a 100% duty cycle of the sensor can be provided, which optimizes collection efficiency. This is important since illumination is a sensitive issue. Such an improvement in performance is due to the high capabilities offered by the sCMOS camera. It must be mentioned, that the HSI system designed in the mentioned Master Thesis (Ingvaldsen, 2012), made use of a similar AOTF and driver provided by Gooch&Housego, and it presented the same troubles. The instruments became unresponsive and the system could never worked properly. There must be something wrong with those devices.

Joan Vila-Frances (2006) and Joan Vila-Frances (2010) also deals with building a HSI using off-the-self components. Some of the techniques presented there to optimize the system, are included or adapted in this work. When presenting performance, those articles do not consider speed, instead, spectral and spatial domains are studied. Tests performed with our system, point to a good performance regarding these aspects.

As the system is not fully working yet, the first step should be make it operative. First, implementing the software triggering (4.2.1) once the RF driver is available, as specified in Section 4.2.1. The matlab code presented in Appendix C should serve as a starting point. Once this is done, if it is considered to be beneficial, the system might be improved using hardware triggering, as pointed in 4.2.2. Professor Amund Skavhaug offered his support to design the actual implementation. With the system operative, the tests described in Section 5 should be carried out to assess its performance. Besides, the power optimization method should be integrated. In particular, a vector containing the optimal power for every wavelength might be stored as a *.mat file and loaded when starting the application, for instance in callback `acqui_CreateFcn()`.

After this first stage, other features may be included. When working with the borrowed system from Gooch&Housego a few useful functionalities were discovered. The user should be able to specify an exposure time for every wavelength, and it is highly desirable that these times can be saved and load from a file, for instance a plain text file (*.txt): the first line of the file could contain just the number of wavelengths (number of lines in the file - 1), then each line specifies a wavelength and the exposure time to be applied when taking that frame (separated by an space or comma).

An algorithm could be included to estimate optimal exposure time for each wavelength, i. e. avoiding saturation to appear in the image. Joan Vila-Frances (2010) mentions: "The exposure time is automatically adjusted to maximize the dynamic range of the sensor. A first image with a short exposure time is acquired so that the software can analyze the histogram of the image and estimate the optimal exposure time". A similar procedure may be studied. Also, an iterative decision-directed algorithm may solve the problem: an image with a testing exposure time is taken, depending on how *far* it is from being saturated, a new exposure time is selected. This is done several times until a good enough result is obtained.

For certain medical applications, an image cube is not of interest, instead, it is necessary to look repeteadly at certain wavelengths, which is not possible now. The application should provide more freedom to choose a spectral profile. Again, being able to save and load this information from a file (for instance a text file *.txt) may be of interest. Not being this the most common case, the user may be forced to introduce the data manually.

When analyzing the data, reflectance image is a key element. The User Interface could provide support to acquire the three image cubes (data, grey/white standard and dark current) and process them to obtain a reflectance HSI. For this purpose, a button called Reflectance can be included. It must be clear for the user which target should place at each moment (grey, dark current or object), for instance by displaying it in the status console. Finally, a tool that allows to select an area within an image should work together with the 'Spectrum' feature, so that, the spectrum averaged over an area can be displayed. This task is performed in Appendix D.2 which may be taken as a model to integrate the function in the GUI.

Chapter 7

Conclusion

This work has provided me with a good insight in different real-life troubles, that must be faced, when it comes to design a system. Besides, I have learnt about programming, which is not my field of expertise, and hyperspectral imaging, a really interesting topic that was completely unknown for me before starting this work.

This document contains valuable information for anybody interested in building a hyperspectral imaging system. Important theoretical knowledge behind such a technology is introduced in Chapter 2, and its translation to a real design is presented in the following chapter. A User Interface was designed, that may serve as a model. Several improvements are already proposed, that may be taken into account when designing a similar software. Also, the tests proposed here are valid to assess the performance of any HSI system.

Finally, for that one who continues with the development of this system, useful information including code and explanations about how to use MEX files and the API, is provided in the Appendices. All this, should help Lise L. Randeberg's group to have an operative, high performance HSI system in a near future.

Appendix A

Graphic User Interface: code

In this appendix, we present the code written to create the Graphical User Interface. In a first section the Matlab code is presented, then the mex files, written in C and basic to control the camera, are shown in a second section. More information about how to interface MEX files and Matlab, and about the API used to control the camera can be found in AppendixB.

A.1 Matlab code

The main code of the User Interface is shown first. Then, the Matlab functions that are called from the User Interface appear in the same order as they do in the code, each of them in a different subsection whose title is the name of the function (and the file *.m without the extension). But before a few details are explained to avoid its repetition as comments along the code. Every function called:

```
function name_CreateFcn(hObject, eventdata, handles)
% hObject       handle to name (see GCBO)
% eventdata     reserved - to be defined in a future version of MATLAB
% handles           structure with handles and user data (see GUIDATA)
```

is executed during object creation, after setting all properties. The example also describes the meaning of the arguments hObject, eventdata and handles which remains the same all along the code. Callbacks functions appear many times within the code, their sintaxis is $name\_Callback$.

They are executed on user interaction with the element of the GUI they represent, or they can be called as normal functions. Their arguments are hObject, eventdata and handles and they have the same meaning as in the commented example. Further information about these and other functions such as OpeningFcn or OutputFcn can be found at the Matlab documentation (Mathworks, 2015).

A.1.1 userInterface

```
function varargout = userInterface(varargin)
% USERINTERFACE MATLAB code for userInterface.fig
%      USERINTERFACE, by itself, creates a new USERINTERFACE or raises ...
    the existing
%      singleton*.
%
%      H = USERINTERFACE returns the handle to a new USERINTERFACE or ...
    the handle to
%      the existing singleton*.
%
%      USERINTERFACE('CALLBACK',hObject,eventData,handles,...) calls ...
    the local
%      function named CALLBACK in USERINTERFACE.M with the given input ...
    arguments.
%
%      USERINTERFACE('Property','Value',...) creates a new ...
    USERINTERFACE or raises the
%      existing singleton*.  Starting from the left, property value ...
    pairs are
%      applied to the GUI before userInterface_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property ...
    application
%      stop.  All inputs are passed to userInterface_OpeningFcn via ...
    varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%       instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help userInterface

% Last Modified by GUIDE v2.5 01-Jun-2015 16:32:19

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @userInterface_OpeningFcn, ...
                   'gui_OutputFcn',  @userInterface_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before userInterface is made visible.
function userInterface_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% Choose default command line output for userInterface
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
```

```
% --- Outputs from this function are returned to the command line.
function varargout = userInterface_OutputFcn(hObject, eventdata, handles)
% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes during object creation, after setting all properties.
function acqui_CreateFcn(hObject, eventdata, handles)
exposure_time = 0.001;
% Open the camera. Evalc allows us to get the "mexprintf("  ");" comments
% inside the mex file
message = evalc('[Hndl, fail] = mOpenCamera(exposure_time);');
if fail == 0
    action = 'on';
    status = cooling(Hndl, action); % activate the cooling mechanism
    setappdata(hObject,'Hndl',Hndl); % store the handle to camera as ...
        application data
end
setappdata(hObject,'message',message);



% --- Executes during object deletion, before destroying properties.
function acqui_DeleteFcn(hObject, eventdata, handles)
Hndl = getappdata(handles.acqui, 'Hndl');
resul = mCloseCamera(Hndl); % when the GUI is closed, the camera must ...
    be closed

% --- Executes on button press in acqui.
function acqui_Callback(hObject, eventdata, handles)
% this is an important function as it allows to acquire a hyperspectral ...
    image
% the wavelengths are specified through the buttons list or range
% it must show the HSI
```

```

list_o_rang = get(handles.list_o_rango, 'Value');
% check if we are defining the wavelength range with list or range
if list_o_rang == 0
    % We define the wavelength with the list
    contents = cellstr(get(handles.list,'String'));
    selection = get(handles.list,'Value');
        for i = 1:length(selection)
             wavelen_string{i} = contents{selection(i)}(1:3);
             wavelengths(i) = str2double(wavelen_string{i});
        end

    %Now we have a vector wavelengths with the lambdas that we want ...
        (double)
    %and another with that number shown as a string
else
    % We define the wavelength with the range
    lamb_start = str2double(get(handles.lamb_start,'String'));
    lamb_end   = str2double(get(handles.lamb_end,'String'));
    lamb_step  = str2double(get(handles.lamb_step,'String'));
    wavelengths = [lamb_start:lamb_step:lamb_end];
end

% the power of the sound wave can be changed as well
mPow = get(handles.power, 'String');
    if mPow(length(mPow))=='%'
    mPow = str2double(mPow(1:length(mPow)-1));
    else
        mPow = str2double(mPow);
    end

% if there is ROI information is not relevant anymore
if isappdata(hObject, 'ROI')==1
    rmappdata(hObject, 'ROI');
end

```

```
% get handle to camera and show cooler status
Hndl = getappdata(handles.acqui, 'Hndl');
message  = getappdata(handles.acqui, 'message');
set(handles.Console,'String', message);
cooler_Callback(handles.cooler,[],handles);

%frequency in (MHz)
frequencies = (sqrt(sin(200./wavelengths)+1)).^(10.55)*20.91950466;
Nlam = length(wavelengths);
Nlam = Nlam+1; % Nlam +1 because of the Background image

% Allocating memory for the image cube and returned vectors. It speeds up
HSI = uint16(0);
HSI(2560,2162,Nlam)=0;
dimensions = uint32(0);
dimensions(1,7)=0;
timestamps = uint64(0);
timestamps(1,Nlam)=0;

if strcmp(get(handles.triggering_mode, 'String'), 'Software')
    [dimensions, HSI, timestamps] = mGetHSISoftware(Hndl, wavelengths);
else
    % Here the Hardware triggering procedure could be implemented
end

if HSI ≠ 0
    % We store the image cube and its wavelengths info
    setappdata(hObject, 'HSI', HSI);
    setappdata(hObject,'wavelengths',wavelengths);

    % Depending on the number of pictures acquired the number of shown ...
        frames
    % will vary. It is because the displayed image looks better
    len = length(wavelengths);
    fin_display = finDisplay(len);
    list_images = num2cell(wavelengths);
    list_images{1, length(wavelengths)+1} = 'Background';
    set(handles.sel_pict, 'Value', 1); % Value must be in range
    set(handles.sel_pict, 'String', list_images);
    value = [1:fin_display];
    set(handles.sel_pict, 'Value', value);

    % Function to show several frames in the axis
    show_pict_Callback(handles.show_pict, [], handles);
else
    set(handles.Console,'String','HSI was not acquired!!')
end


% --- Executes on button press in liveView.
function liveView_Callback(hObject, eventdata, handles)
% This will usually be the first button the user presses. To focus the
% camera and point to the target. Live video should be displayed

test = get(hObject, 'String');
if strcmpi(test,'Live View')==1 % check if preview is ON
    %Activate preview
    list_o_rang = get(handles.list_o_rango, 'Value');
    if list_o_rang == 0
        % We define the wavelength with the list
        contents = cellstr(get(handles.list,'String'));
        selection = get(handles.list,'Value');
        wavelen_string{1} = contents{selection(1)}(1:3);
        wavelength = str2double(wavelen_string{1});
    else
        % We define the wavelength with the range
        wavelength = str2double(get(handles.lamb_start,'String'));
    end
    mPow_str = get(handles.power, 'String');
    if mPow_str(length(mPow_str))=='%'
    mPow = str2double(mPow_str(1:length(mPow_str)-1));
    else
    mPow = str2double(mPow_str);
    end
    % CHECK
    %  AOTF
    % [ser]=initAOTF(mPow);
    % set(handles.Console,'String','AOTF and CAM initialized');

    %set frequency
%     liveflag = 0; channel = 6;
%     freq = Lam2Freq(wavelength);
    % [chFRhex] = GetChRF(liveflag, channel, freq, mPow); %%% MPOW ...
       SHOULD BE AMPLITUDE. CHECK
    % [out] = sendHex2RF(ser,chFRhex);

    if isappdata(hObject, 'ROI')==1
        rmappdata(hObject, 'ROI'); % the data from a former ROI is not ...
           relevant anymore
    end
    Hndl = getappdata(handles.acqui, 'Hndl'); % get the handle for camera
    message  = getappdata(handles.acqui, 'message');
    set(handles.Console,'String', message);
    cooler_Callback(handles.cooler,[],handles); % show cooler status

    % Change the button string to Stop Preview, it will change the
    % behaviour of the callback next time it's pressed
    set(hObject,'FontSize',10);
    set(hObject,'String','Stop Preview');



    axes(handles.display);
    set(handles.display,'Position', [66 11.385 144 65.4615]);
    set(handles.display, 'ButtonDownFcn', {@display_ButtonDownFcn, ...
       handles});

    resul = mSetLiveView(Hndl); % camera settings previous to LiveView
    if resul == 1
```

```
        HSI = uint16(0);
        HSI(2560,2160)=0; % allocating memory for the frame
        dimensions=uint32([2160 2560 11059200]); % setting the dimensions
        %of the frame that will be acquired

        % Loop: a frame is obtained and shown until Stop Preview is pressed
        while strcmp(get(hObject,'String'), 'Stop Preview')==1
            tic,
            message = evalc('HSI = mGetFrameLiveView(Hndl, dimensions);')
            handles.display = imshow(imadjust(HSI));
            drawnow;
            toc
            % tic, toc shows in console the time required to execute the
            % sequence between tic and toc. I've used it to optimize speed
            % many times. It is left here as a reminder: the LiveView
            % function can probably be optimized through "Circular Buffers"
            % as explained.
        end
    end

else % Stop Preview (not Live View) has been pressed
    Hndl = getappdata(handles.acqui, 'Hndl');
    resul = mStopLiveView(Hndl);
    set(handles.Console,'String','Closed AOTF. Stopped preview');
    set(hObject,'FontSize',13);
    set(hObject,'String','Live View');
end


function lamb_start_Callback(hObject, eventdata, handles)
% Validate that the text in the lamb_start field converts to a real number
lamb_start = str2double(get(hObject,'String'));
if isnan(lamb_start) || ¬isreal(lamb_start) || lamb_start < 400 || ...
        lamb_start > 1000
    % isdouble returns NaN for non−numbers and lamb_start cannot be complex
    % Disable the Acquire button and explain why in the GUI console
    set(hObject,'String','Error')
    set(handles.acqui,'Enable','off')
    set(handles.Console,'String','Start must be a number between 400 ...
        and 1000')
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
else
    % Enable the Acquire button with its original name
    set(handles.acqui,'Enable','on')
    uicontrol(handles.lamb_end)
end


function lamb_end_Callback(hObject, eventdata, handles)
% Validate that the text in the lamb_end field converts to a real number
lamb_end = str2double(get(hObject,'String'));
if isnan(lamb_end) || ¬isreal(lamb_end) || lamb_end < 400 || lamb_end > ...
    1000
    % isdouble returns NaN for non-numbers and lamb_start cannot be complex
    % Disable the Acquire button and explain why in the GUI console
    set(hObject,'String','Error')
    set(handles.acqui,'Enable','off')
    set(handles.Console,'String','End must be a number between 400 and ...
        1000')
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
else
    % Enable the Acquire button with its original name
    set(handles.acqui,'Enable','on')
    uicontrol(handles.lamb_step)
end


function lamb_step_Callback(hObject, eventdata, handles)
lamb_step = str2double(get(hObject,'String'));
```

```
if isnan(lamb_step) || ¬isreal(lamb_step) || lamb_step < 0 || lamb_step ...
    > 1000
    % isdouble returns NaN for non-numbers and lamb_start cannot be complex
    % Disable the Plot button and explain why in the GUI console
    set(hObject,'String','Error')
    set(handles.acqui,'Enable','off')
    set(handles.Console,'String','Step must be a number between 400 and ...
        1000')
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
else
    % Enable the Plot button with its original name
    set(handles.acqui,'Enable','on')
    uicontrol(handles.acqui);
end


% --- Executes on button press in list_o_rango.
function list_o_rango_Callback(hObject, eventdata, handles)
% selects if the wavelenghts vector is defined as a list or a range
state = get(hObject, 'Value');
if state == 0
    % List mode enabled
    set(hObject,'String','List');
    set(handles.lamb_start,'Enable','off');
    set(handles.lamb_end,'Enable','off');
    set(handles.lamb_step,'Enable','off');
    set(handles.list,'Enable','on');
else
    % Range input enabled
    set(hObject,'String','Range');
    a = double.empty(1,0);
    set(handles.list,'Value', a); %we take out the selection
    set(handles.list,'Enable','off');
    set(handles.lamb_start,'Enable','on');
    set(handles.lamb_end,'Enable','on');
    set(handles.lamb_step,'Enable','on');
end


function power_Callback(hObject, eventdata, handles)
%This callback should allow to change the power
% of the signal generated by the driver (not finished)

power_str = get(hObject,'String');
% Value of power is get no matter if symbol % is typed or not
if power_str(length(power_str))=='%'
    power = str2double(power_str(1:length(power_str)-1));
else
    power = str2double(power_str);
end

if isnan(power) || ¬isreal(power) ||  power > 100
    % isdouble returns NaN for non-numbers and lambda cannot be complex
    % Disable the Plot button
    set(hObject,'String','Error');
    set(handles.liveView,'Enable','off');
    % Give the edit text box focus so user can correct the error
    uicontrol(hObject)
else
    % Enable the Plot button with its original name
    set(handles.liveView,'Enable','on');
    power_str = strcat(num2str(power), '%');
    set(hObject,'String',power_str)

    wavelength = str2double(get(handles.liveV_wavelength,'String'));

    path_root = get(handles.path_STARTING_POINT, 'UserData');
    path = [path_root '\ser.mat'];
    load(path);

%     %set frequency CHECK
%     liveflag = 0; channel = 6;
%     freq = Lam2Freq(wavelength);
%     [chFRhex] = GetChRF(liveflag, channel, freq, power); %%% MPOW ...
    SHOULD BE AMPLITUDE. CHECK
%     [out] = sendHex2RF(ser,chFRhex);
end


% --- Executes during object creation, after setting all properties.
function Console_CreateFcn(hObject, eventdata, handles)
% This is a small console where info can be shown
if ispc && isequal(get(hObject,'BackgroundColor'), ...
    get(0,'defaultUicontrolBackgroundColor'))
     set(hObject,'BackgroundColor',[0.9 0.9 0.9]); %that is grey color
end
set(hObject,'String','Remember to turn on Camera and AOTF');


% --- Executes on button press in restart.
function restart_Callback(hObject, eventdata, handles)
% Camera is closed and opened again, AOTF should be as well
mModifiedCloseCamera();
acqui_CreateFcn(handles.acqui, [], handles)
%Here AOTF
 %
 %


% --- Executes on button press in subs.
function subs_Callback(hObject, eventdata, handles)
% The Background frame is substracted to all the others. Its behaviour
% changes if we press again the button: Add Bkg
HSI = getappdata(handles.acqui, 'HSI');
%HSI without substracting is kept in case operation must be undone
setappdata(handles.acqui, 'undo', HSI);

```

```
wavelengths = getappdata(handles.acqui, 'wavelengths');

if wavelengths==0 % if wavelengths is empty so is HSI
    set(handles.Console, 'String', 'You must load an HSI first');
else
     Nlam = length(HSI(1,1,:));

    if strcmp(get(hObject, 'String'), 'Subs Bkg')%substraction
        for i=1:Nlam-1
            HSI(:,:,i) = (HSI(:,:,i)-HSI(:,:,Nlam));
        end
        set(hObject, 'String', 'Add Bkg'); % behaviour can be changed
    else
        for i=1:Nlam-1
            HSI(:,:,i) = (HSI(:,:,i)+HSI(:,:,Nlam));%adding
        end
        set(hObject, 'String', 'Subs Bkg');
    end

    setappdata(handles.acqui, 'HSI', HSI);

    show_pict_Callback(handles.show_pict, [], handles);% show it
end


% --- Executes on button press in show_pict.
function show_pict_Callback(hObject, eventdata, handles)
% It shows the current HSI or the current ROI
axes(handles.histH); cla reset; set(handles.histH,'Visible', 'off');
axes(handles.histV); cla reset; set(handles.histV,'Visible', 'off');
axes(handles.display); cla reset; % axes must be resets

% get the current HSI or ROI(if any)
if isappdata(handles.acqui, 'ROI') == 0
    HSI = getappdata(handles.acqui,'HSI');
    wavelengths = getappdata(handles.acqui,'wavelengths');
```

```
else
    ROI = getappdata(handles.acqui,'ROI');
    HSI = ROI.little_img;
    wavelengths = ROI.wavelengths;
end


if wavelengths==0 % if wavelengths is empty so is HSI now
    set(handles.Console, 'String', 'You must load or take a HSI first');
else
    % only the selected frames in the listbox are shown
    contents = cellstr(get(handles.sel_pict,'String'));
    selection = get(handles.sel_pict,'Value');
    wavelengths = 0;
    for i = 1:length(selection)
         wavelengths(i) = str2double(contents{selection(i)});
         selected(:,:,i) = HSI(:,:, selection(i));
    end

    if length(selection) == 1 % if there is only one frame to show
        %we just use the matlab function imshow
        set(handles.display,'Position', [66 11.385 144 65.4615]);
        selected = imadjust(squeeze(selected));

        axes(handles.display);
        handles.display = imshow(selected);
        pixelInfo = impixelinfo(handles.display);
        set(pixelInfo,'Position', [180 100 150 23]);

        % if we are showing only one frame, its stored as Selected frame
        % to be used if pixel Histogram is required
        set(handles.roi,'Enable', 'on');
        setappdata(handles.acqui,'Selected_frame', selected);

        set(handles.display, 'ButtonDownFcn', {@display_ButtonDownFcn, ...
           handles});

    else % if there are more than one we call our function showImages
        set(handles.display,'Position', [-0.2 5.154 268.2 71.77]);

        axes(handles.display);
        handles.display = showImages(selected);

        if isappdata(handles.acqui, 'Selected_frame')
            rmappdata(handles.acqui, 'Selected_frame');
        end
    end

end


% --- Executes on mouse press over axes background.
function display_ButtonDownFcn(hObject, eventdata, handles)
   % Histogram of the pixel the user clicks on is shown in new axes
   if isappdata(handles.acqui, 'Selected_frame')
       % Get the single frame shown (not a montage of several frames)
       frame = getappdata(handles.acqui, 'Selected_frame');

       axesHandle  = get(hObject,'Parent');
       coordinates = get(axesHandle,'CurrentPoint'); % get coordinates
       coordinates = round(coordinates(1,1:2));

       set(handles.histH,'Visible', 'on'); % enable histH and histV axis
       set(handles.histV,'Visible', 'on');

       % Get the data of the full row and column the pixel belongs to
       H_plot = [];
       V_plot = [];
       H_plot = double(frame(coordinates(2), :))./65535;
       V_plot = double(frame(:, coordinates(1)))./65535;

       % Plot them
```

```
        H = [1:length(H_plot)];
        axes(handles.histH);
        plot(H, H_plot, 'LineStyle', '-', 'Color', [1 0.5 0]);
        axis([0 2160 0 1])
        set(handles.histH,'Color', [0.9 0.9 0.9]);

        V = [1:length(V_plot)];
        axes(handles.histV);
        plot(V_plot, V, 'LineStyle', '-', 'Color', [1 0.5 0]);
        axis([0 1 0 2560])
        set(handles.histV,'Color', [0.9 0.9 0.9]);
    end

% --- Executes on button press in roi.
function roi_Callback(hObject, eventdata, handles)
% It allows to select a region of interest in the acquired HSI

% Get the data we are going to work with
HSI = getappdata(handles.acqui,'HSI');
wavelengths = getappdata(handles.acqui,'wavelengths');
if isappdata(handles.acqui,'ROI')
    HSI = getappdata(handles.acqui,'ROI');
    HSI = HSI.little_img;
end
if isappdata(handles.acqui,'Selected_frame')
    img = getappdata(handles.acqui,'Selected_frame');
else
    img(:,:) = imadjust(squeeze(HSI(:,:,1)));
    set(handles.sel_pict, 'Value', 1);
end

if wavelengths ≠ 0 % if wavelengths is not empty
    axes(handles.display);
    mask = roipoly(img); % draw the ROI
    % Get coordinates of ROI and define a small rectangle including the ROI
    [row col] = find(mask);
    xmin = col(1);
    xmax = col(length(col));
    ymin = min(row);
    ymax = max(row);

    umask = ¬mask;

    Nlam = length(wavelengths)+1;
    % Prepare the mask to filter and keep only the ROI
    little_mask = mask(ymin:ymax, xmin:xmax);
    little_umask = ¬little_mask;
    % allocate memory for the small image that will
    % contain the ROI of the normal image
    little_img = zeros(ymax-ymin+1, xmax-xmin+1, Nlam, 'uint16');

    % Filter
    for i=1:Nlam
     little_img(:,:,i) = HSI(ymin:ymax, xmin:xmax, i);
     low(i) = min(min(little_img(:,:,i)));

     h=[0];      %%%CHECK Useful code to filter ROI
     little_img(:,:,i) = roifilt2(h, squeeze(little_img(:,:,i)), ...
         little_umask);
%           [row col] = find(little_img(:,:, 1, i));
%           for k=1:length(col)
%              pixels_in_ROI(k) = little_img(row(k),col(k), 1, i);
%           end
     high(i) = max(max(little_img(:,:,i)));
%          h=[1];      %%% Useful code to filter ROI
%          img(:,:,1,i) = roifilt2(h, img(:,:,1,i), mask);
    end
    % Store parameters associated to ROI as a struct
    ROI = struct('HSI', HSI, 'wavelengths', wavelengths, 'mask', mask, ...
       'little_img', little_img, 'little_mask', little_mask);
    setappdata(handles.acqui,'ROI', ROI);

    show_pict_Callback(handles.show_pict, [], handles);% show the ...
        filtered images
else % if there are none or more than one frames selected (shown)
    set(handles.Console, 'String', 'You must load an HSI first, and ...
        select one frame!');
end


% --- Executes on button press in full_Image.
function full_Image_Callback(hObject, eventdata, handles)
% After defining a ROI this function allows us to come back to the original
% full size image: by deleting the ROI information, show_pict does the rest
if isappdata(handles.acqui, 'ROI')
    rmappdata(handles.acqui, 'ROI');
end
if isappdata(handles.acqui, 'Seleceted_frame')
    rmappdata(handles.acqui, 'Seleceted_frame');
end
show_pict_Callback(handles.show_pict, [], handles);


% --- Executes on button press in cooler.
function cooler_Callback(hObject, eventdata, handles)
% this function gets the status of the cooling function from the camera and
% display it on the cool_status
Hndl = getappdata(handles.acqui, 'Hndl');
action = 'get';
message = evalc('status = cooling(Hndl, action);');
set(handles.cool_status, 'String', '...........');
pause(0.1);
set(handles.cool_status, 'String', status);
set(handles.Console, 'String', message);


% --- Executes during object creation, after setting all properties.
function path_starting_point_CreateFcn(hObject, eventdata, handles)
```

```
%We need to have a .mat file with a string variable called ...
    path_STARTING_POINT
%Then we store it as a property of path_STARTING_POINT and we can get it.
path = ...
    'C:\Users\vnir1600\Documents\MATLAB\Variables\path.mat';%%%CRITIC ...
    PATH POINT
load(path);
set(hObject,'UserData', path);

% --- Executes on button press in path_starting_point.
function path_starting_point_Callback(hObject, eventdata, handles)
%Store the current path_STARTING_POINT in the property UserData so that ...
    we can get it.
%If user presses this button, new path_STARTING_POINT will be selected
path = uigetdir();
if path ≠ 0
    cd(path);
    path_variable = [path '\path.mat']
    save(path_variable, 'path');
    set(hObject,'UserData', path);
end


% --- Executes on button press in pixelRegion.
function pixelRegion_Callback(hObject, eventdata, handles)
% it allows to use Matlab functionality impixelregion to have
% a closer look at the pixels of an image
pixelRegion = impixelregion(handles.display);
set(pixelRegion,'Position', [1280 10 600 360]);


% -----Executed if user presses save icon (in the top left)--------
function save_icon_ClickedCallback(hObject, eventdata, handles)
% Saves a HSI as .mat file to harddrive it takes some time, it provides an
% standard name but the user can change it

% Get an standard name: HSimag + date
id_time = timeString(1);
path_ima = get(handles.path_starting_point, 'UserData');
Images_string = '\Images    ';
path_ima(35:44)=Images_string(1:10);
path = strcat(path_ima, '\HSimag', id_time,'.mat');
if exist(path, 'file') == 2 % if we take HSI in the same minute they ...
    will have the same name and
   path = strcat(path_ima, '\HSimag', id_time,'a','.mat');
    if exist(path, 'file') == 2 % we would overwrite the last one. With ...
        this code that is avoided
        path = strcat(path_ima, '\HSimag', id_time,'b','.mat');
            if exist(path, 'file') == 2
             path = strcat(path_ima, '\HSimag', id_time,'c','.mat');
            end
    end
end
% Let the user choose if he wants the standard or another
[FileName,PathName] = uiputfile(' ', 'Select a file to save HSI in: ', ...
    path);
if ischar(FileName) && ischar(PathName)
    HSI = getappdata(handles.acqui,'HSI');
    wavelengths = getappdata(handles.acqui,'wavelengths');
    path = [PathName FileName];
    save(path, 'HSI', 'wavelengths');
    set(handles.Console,'String', 'HSI stored');
else
    set(handles.Console,'String', 'You must select a valid path');
    save_icon_ClickedCallback(hObject, [], handles);
end


% -----Executed if user presses open icon (in the top left)--------
function open_icon_ClickedCallback(hObject, eventdata, handles)
% it allows to open a HSI stored as .mat file, it first open the "usual"
% folder but the user can choose
```

```
path_root = get(handles.path_starting_point, 'UserData');
Images_string = '\Images    ';
path_root(35:44) = Images_string(1:10);

[file path] = uigetfile('*.mat', 'Select HyperSpectral Image', path_root);
if ischar(file) && ischar(path)
    path = strcat(path, file);
    load(path);

    if isappdata(hObject, 'ROI')==1
        rmappdata(hObject, 'ROI');
    end
    setappdata(handles.acqui, 'HSI', HSI);
    setappdata(handles.acqui,'wavelengths',wavelengths);

    % We update the List Box with the images and call show_pict_Callback
    len = length(wavelengths);
    fin_display = finDisplay(len);
    list_images = num2cell(wavelengths);
    list_images{1, length(wavelengths)+1} = 'Background';
    set(handles.sel_pict, 'Value', 1); % Value must be in range
    set(handles.sel_pict, 'String', list_images);
    value = [1:fin_display];
    set(handles.sel_pict, 'Value', value);

    show_pict_Callback(handles.show_pict, [], handles)
else
    set(handles.Console,'String', 'You must select a valid file');
    open_icon_ClickedCallback(hObject, [], handles);
end


% --- Executes on button press in spectrum.
function spectrum_Callback(hObject, eventdata, handles)
% If User has selected one or several pixels, their spectrum (the value of
% the pixel for the different frames is plotted.
```

```
content = get(handles.list_pix,'String');
if ¬strcmp(content, 'Select first ^')
    if isappdata(handles.acqui, 'Selected_frame')
    rmappdata(handles.acqui, 'Selected_frame'); % so that it doesnt ...
        fail if we click select again
    end

    selected = get(handles.list_pix,'Value');
    row_col = get(handles.list_pix, 'UserData');
    for i=1:length(selected)
        row(i) = row_col.row(selected(i));
        col(i) = row_col.col(selected(i));
    end

    if ¬isempty(row) && ¬isempty(col)
        HSI = getappdata(handles.acqui, 'HSI');
        wavelengths = getappdata(handles.acqui, 'wavelengths');

        for i=1:length(selected) % get spectrum
            spectrum(i,:) = HSI(row(i), col(i), :);
            %spectrum2(i,:) = HSI(:, col(i), row(i));
        end
        axes(handles.display); cla reset; % axes must be reset

        % Plot all the spectrums
        axes(handles.display);
        for i=1:length(selected)
        plot(wavelengths, spectrum(i,1:length(wavelengths)));
        hold on
        end
        axis('fill');
    end
end


% --- Executes on button press in select_pixels.
```

```
function select_pixels_Callback(hObject, eventdata, handles)
 % User can select some pixel in a frame to after that plot their spectrum.
 % Requires that there is only one frame shown
 % (to make possible the pixel selection).

 axes(handles.display);

if isappdata(handles.acqui, 'Selected_frame')

       frame = getappdata(handles.acqui, 'Selected_frame');

       [col row ¬] = impixel();

       for i=1:length(col)
           list {1, i} =  ['(' num2str(row(i)) ', ' num2str(col(i)) ')'];
       end

       set(handles.list_pix, 'Value', 1); % Value must be in range
       set(handles.list_pix, 'String', list);
       row_col = struct('row', row, 'col', col);
       set(handles.list_pix, 'UserData', row_col);
else
    set(handles.Console, 'String', 'You must have a single frame first');
end


% --- Executes on button press in triggering_mode.
function triggering_mode_Callback(hObject, eventdata, handles)
% It allows to select the triggering mode
string = get(hObject, 'String');
if strcmp(string,'Software') == 1
    set(hObject,'String','Hardware');
else
    set(hObject,'String','Software');
end

```

```
function exposureTime_Callback(hObject, eventdata, handles)
Hndl = getappdata(handles.acqui, 'Hndl');
exposure_time = get(hObject, 'String');
set(hObject, 'String', '.....');
pause(0.1);
exposure_time = str2double(exposure_time);
message = evalc('[resul, exp_applied] = mExposureTime(Hndl,  ...
    exposure_time);');
set(hObject, 'String', num2str(exp_applied));
set(handles.Console, 'String', message);
```

A.1.2 cooling

```
%% COOLING

function [status] = cooling(handle, action)
% Action may be:
% 'on' or 1: it means that we turn on the cooling
% 'off' or 0: it means that we turn off the cooling
% 'get' or 2: we get the actual status
% handle is the handle to the camera
% status is the output. A string obtained from mCooling.

if (strcmp(action, 'on'))
    action = 1;
end
if (strcmp(action, 'off'))
    action = 0;
end
if (strcmp(action, 'get'))
    action = 2;
end

if (action ≠ 1 & action ≠ 0 & action ≠ 2)
```

```
    disp('WARNING! Wrong input "action" ');
else
    status = mCooling(handle, action);
end
end
```

A.1.3 finDisplay

```
function [fin_display] = finDisplay(len)
% Returns how many images should be shown to avoid black squares
% on the montaged image. Its logic is based on testing how the montaged
% image looks like for n frames, and using instead k frames (with k as
% close as possible to n)
% len should be the number of frames in the HSI
fin_display = len;

if (len == 3 || len == 4)
    fin_display = 2;
end
if (len == 5 || len == 7 || len == 11 || len == 14 || len == 19)
    fin_display = len + 1; %we show the background too
end
if (len == 9 || len == 10)
    fin_display = 8;
end
if (len == 13)
    fin_display = 12;
end
if (len == 16 || len == 17 || len == 18)
    fin_display = 15;
end
if (len>20)
    fin_display = 20;
end
```

```
end
```

A.1.4 showImages

```
% showImages
function [handler] = showImages(HSI)

% INPUT:
% HSI - we need the Hyperspectral image
% OUTPUT:
% handler to the montaged Image

Nlam = length(HSI(1,1,:));
% Match our HSI with the format required by montage
for i=1:Nlam
    aux(:,:,1,i) = HSI(:,:,i);
    aux(:,:,1,i) = imadjust(aux(:,:,1,i));
end
handler = montage(aux); %call montage
end
```

A.1.5 timeString

```
% timeString(hyphen)
function [id_time] = timeString(hyphen)
% it gets the system time and converts it to a string suitable to be the
% name of a file or variable

% Hyphen: if we want a hyphen or we need that it only has underscores
% (variables can't contain hyphen in the name, but .mat files can)

clk = clock;
```

```
id_tim = num2str([clk(2) clk(3) clk(4) clk(5)]);
len = length(id_tim);

k=0;
for h=1:3
for i=1:len
   if id_tim(i)==' '
      if id_tim(i+1)==' '
         id_tim(i:len-1) = id_tim(i+1:len);
         k=k+1;
      end
   end
end
end

j=0;
for i = 1:len-k+1
    if id_tim(i)==' ' && j≠1
    id_tim(i)='-';
    j=1;
    end
    if id_tim(i)==' ' && j==1
    id_tim(i)='_';
    j=2;
    end
end
espacio = '           ';
id_tim(len-k+1:len) = espacio(1:k);
id_time(1:len-k) = id_tim(1:len-k);
if hyphen == 0
    for i=1:length(id_time)
        if id_time(i)=='-'
            id_time(i)='_';
        end
    end
end
```

```
end
```

A.2 Mex files

All the mex files are shown here, in the same order as they appear in the code in the former section. Again, for each mex file (or mex function) there is a subsection whose title is the name of the mex file.

A.2.1 mOpenCamera

```
/*******************************************************************
* [double handle, double warning] = mOpenCamera(double *exposure_time);
*
* Output: it provides a handle to camera
*
* Open the camera, enable cooling and get the serial number.
*******************************************************************/
#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"

using namespace std;

double openCamera(mxArray *out_Handle, double *exposure_time);

void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double *warning;

    mxArray *out_Handle, *warning_out;
    double *exposure_time;

    /* Get input*/
    exposure_time = mxGetPr(prhs[0]);

    /*Allocate memory for output*/
    warning = static_cast<double *> (mxMalloc(sizeof(double)));

    /*Prepare output*/
    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    out_Handle = plhs[0];

    /*Call openCamera (below)*/
    warning[0] = openCamera(out_Handle, exposure_time);

    /*output*/
    plhs[1] = mxCreateDoubleMatrix(1, 1, mxREAL);
    warning_out = plhs[1];
    mxSetPr(warning_out, warning);

}

double openCamera(mxArray *out_Handle, double *exposure_time){

    int i_retCode;
    double *d_Hndl;
    double warning=0;

    /*Initialise Library to control camera*/
    i_retCode = AT_InitialiseLibrary();
    if (i_retCode != AT_SUCCESS) {
        mexPrintf("WARNING! Error initialising library");
        warning = 1;
    }
    else {
        AT_64 iNumberDevices = 0;
        i_retCode =AT_GetInt(AT_HANDLE_SYSTEM, L"Device Count", ...
            &iNumberDevices);
        if (iNumberDevices ≤ 0) {
            mexPrintf("WARNING! No cameras detected");
            warning = 1;
        }
        else {
            AT_H Hndl;
            /*Open camera and get handle*/
            i_retCode = AT_Open(0, &Hndl);

            /*Allocate memory for output and assign value*/
            d_Hndl = static_cast<double *> (mxMalloc(sizeof(double)));
            d_Hndl[0] = static_cast<double> (Hndl);//lo paso de AT_H a ...
                double
            mxSetPr(out_Handle, d_Hndl);

            if (i_retCode != AT_SUCCESS) {
                mexPrintf("WARNING! Error getting handle");
                warning = 1;
            }
            else {
                mexPrintf("Successfully initialised camera: ");

                /*Set exposure time if camerra successfully intialised*/
                i_retCode = AT_SetFloat(Hndl, L"ExposureTime", ...
                    *exposure_time);

                /*Check if it has been succesfully set*/
                double exp_time_applied;
                i_retCode = AT_GetFloat(Hndl, L"ExposureTime", ...
                    &exp_time_applied);
                if (*exposure_time != exp_time_applied){
                    warning = 1;
                    mexPrintf("WARNING!! ");
                }
```

```

                    /*Enable metadata*/
                    i_retCode = AT_SetBool(Hndl, L"MetadataEnable", 1);
                    i_retCode = AT_SetBool(Hndl, L"MetadataTimestamp", 1);

                    /*Serial Number*/
                    AT_WC serialnumber[64];
                    i_retCode = AT_GetString(Hndl, L"Serial Number", ...
                        serialnumber, 64);
                    if (i_retCode == AT_SUCCESS) {

                        char serial_number[64];
                        wcstombs(serial_number, serialnumber, 64);  // I ...
                            have serial number as AT_WC and as char[64]
                        mexPrintf("%s   ", serial_number);
                    }
                    else {
                        mexPrintf("NO-SERIAL-NUM ");
                        warning = 1;
                    }
                    mexPrintf("Exposure time set to: %f  ", exp_time_applied);

                }
            }
        }
    return warning;
}
```

A.2.2 mCloseCamera

```
/************************************************************************
* int resul = mCloseCamera(double handle);

* Close camera and finalise library.
```

```
**********************************************************************/
#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"

using namespace std;

void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double *handle, *d_retCode;
    mxArray *resul_out;

    /* Get input and allocate memory for output*/
    handle = mxGetPr(prhs[0]);
    d_retCode = static_cast<double *> (mxMalloc(sizeof(d_retCode)));

    AT_H Hndl;
    Hndl = static_cast<AT_H> (*handle);

    /*Close camera and write the return code to the output*/
    d_retCode[0] = static_cast<double> (AT_Close(Hndl));

    /*Finalise library*/
    AT_FinaliseLibrary();

    /*Output*/
    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    resul_out = plhs[0];
    mxSetPr(resul_out, d_retCode);
}
```

A.2.3 mGetHSISoftware

```
/**********************************************************************
* [dimensions, HSI, timestamp] = mGetHSISoftware(double handle, double ...
    *wavelengths);

dimensions = [Nlam, pix_height, pix_width, B_ImageSize, B_stride, ...
    success, transposed]
HSI is the pix_height*pix_width*Nlam
*
*
* Keep in mind:
* <> Use 0-based indexing as always in C or C++
* <> Indexing is column-based as in Matlab (not row-based as in C)
* <> Use linear indexing.  [x*dimy+y] instead of [x][y]
**********************************************************************/
#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"
#include <windows.h>

#include <time.h>

//#include "initAOTF.cpp"

using namespace std;

typedef unsigned long long uint64;

void getImage(unsigned char *HSI, mwSize *dimensions, double handle, ...
    int *wavelengths);
uint64 extractMetadata(unsigned char* metadata, int length_metadata);

void mexFunction(int nlhs, mxArray *plhs[],
     int nrhs, const mxArray *prhs[])
```

```
{
    mxArray  *dimensions_out, *HSI_out, *timestamp_out;
    double *handle;
    unsigned char *HSI;
    unsigned short *uint16_HSI;
    mwSize *dimensions;
    int *wavelengths, Nlam;

    int i_retCode;
    AT_64 height, width, stride, ImageSizeBytes;

    /*Get inputs*/
    handle = mxGetPr(prhs[0]);

    AT_H Hndl;
    Hndl = static_cast<AT_H> (*handle);

    wavelengths = static_cast<int *> (mxGetData(prhs[1]));
    Nlam = static_cast<int> (mxGetN(prhs[1]));
    Nlam = Nlam + 1;


    /*Allocate memory for dimensions*/
    dimensions = static_cast<mwSize *> (mxMalloc(7 * sizeof(mwSize)));

    /*Get image dimensions, it could be easily edited to set dimensions
     instead of get them, and thus get customed sized images*/
    i_retCode = AT_GetInt(Hndl, L"AOIHeight", &height);
    i_retCode = AT_GetInt(Hndl, L"AOIWidth", &width);
    i_retCode = AT_GetInt(Hndl, L"ImageSizeBytes", &ImageSizeBytes);
    i_retCode = AT_GetInt(Hndl, L"AOIStride", &stride);
    dimensions[0] = static_cast<mwSize> (Nlam);
    dimensions[1] = static_cast<mwSize> (height);
    dimensions[2] = static_cast<mwSize> (width);
    int i_imageSize = static_cast<int>(ImageSizeBytes);
    dimensions[3] = static_cast<mwSize> (i_imageSize);
    dimensions[4] = static_cast<mwSize> (stride);

    /*Allocate memory and align the image*/
    HSI = static_cast<unsigned char *> (mxMalloc((Nlam  * i_imageSize + ...
       8) * sizeof(unsigned char)));
    //HSI = reinterpret_cast<unsigned char *>( ...
       (reinterpret_cast<unsigned long>( HSI ) + 7 ) & ¬0x7);


    /*Initialize AOTF*/
//  i_retCode = initAOTF(50);

    /*We get the image*/
    getImage(HSI, dimensions, *handle, wavelengths);


    if (dimensions[5] == 1){

        /*Image was succesfully acquired*/
        uint64 *timestamp;
        timestamp = static_cast<uint64 *> (mxMalloc(Nlam * ...
           sizeof(uint64)));

        /*Stract the metadata string*/
        int fin, init, len;
        unsigned char *metadata;
        init =  2*dimensions[1]*dimensions[2];
        fin = dimensions[3];
        len = fin-init;

        metadata = new unsigned char[len];

        int offset=0;
        for (int k=0; k < Nlam; k++){
            offset =  k * dimensions[3];
            for(int i=init; i<fin; i++){
                metadata[i−init] = HSI[i+offset];
            }
            /*Get the timestamp from the Metadata string*/
            timestamp[k] = extractMetadata(metadata, len);
        }

        /*Set the transposed flag*/
        dimensions[6] = 1;

        /*OUTPUT*/

        /*Dimensions*/
        plhs[0] = mxCreateNumericMatrix(1, 7, mxUINT32_CLASS, mxREAL);
        dimensions_out = plhs[0];
        mxSetData(dimensions_out, dimensions);

        /*Image*/
        uint16_HSI = reinterpret_cast<unsigned short*>(HSI);

        const mwSize dims_zero[3] = {0, 0, 0};
        plhs[1] = mxCreateNumericArray(3, dims_zero, mxUINT16_CLASS, ...
            mxREAL);

        HSI_out = plhs[1];

        mxSetData(HSI_out, uint16_HSI);

        mwSize dims[3];
        dims[2] = Nlam;
        dims[0] = dimensions[2];
        dims[1] = dimensions[1]+2; //because of the metadata

        i_retCode = mxSetDimensions(plhs[1], dims, 3);

        /*Timestamp*/
        plhs[2] = mxCreateNumericMatrix(1, Nlam, mxUINT64_CLASS, mxREAL);
        timestamp_out = plhs[2];
        mxSetData(timestamp_out, timestamp);
        mxSetM(plhs[2], 1);
        mxSetN(plhs[2], Nlam);

        /*Free allocated memory*/
        delete metadata;
    }
    else{

        /*OUTPUT in case image acquisition fails*/

        /*Dimensions, in the flag dimensions[5] we have the ...
           notification that something failed*/
        plhs[0] = mxCreateNumericMatrix(1, 7, mxUINT32_CLASS, mxREAL);
        dimensions_out = plhs[0];
        mxSetData(dimensions_out, dimensions);

        /*Image just a 0*/
        plhs[1] = mxCreateNumericMatrix(1, 1, mxUINT16_CLASS, mxREAL);
        HSI_out = plhs[1];
        uint16_HSI = static_cast<unsigned short *> ...
            (mxMalloc(1*sizeof(unsigned short)));
        uint16_HSI[0] = 0;
        mxSetData(HSI_out, uint16_HSI);
        mxSetM(plhs[1], 1);
        mxSetN(plhs[1], 1);


        /*Timestamp just a 0*/
        plhs[2] = mxCreateNumericMatrix(1, 1, mxUINT64_CLASS, mxREAL);
        timestamp_out = plhs[2];
        uint64 *timestamp;
        timestamp = static_cast<uint64 *> (mxMalloc(1*sizeof(double)));
        timestamp[0] = 0;
        mxSetData(timestamp_out, timestamp);
        mxSetM(plhs[2], 1);
        mxSetN(plhs[2], 1);

        mxFree(HSI);
    }

    /*for(int i = 0; i < Nlam; i++) delete[] frames[i];*///CHECK
        return;
}


void getImage(unsigned char *HSI, mwSize *dimensions, double handle, ...
    int *wavelengths){

    int i_retCode;
    AT_H Hndl;
    Hndl = static_cast<AT_H> (handle);

    /*Frames is needed to keep the starting address of each frame*/
    unsigned char **frames;
    frames = new unsigned char* [dimensions[0]];

    /*for(int i = 0; i < Nlam; i++) {
        frames[i] = new unsigned char [i_imageSize];
    }*/

    frames[0] = HSI;
    int size;
    size = static_cast<int>(dimensions[3]);

    /*Queue first buffer, then queue all the others*/
    i_retCode = AT_QueueBuffer(Hndl, HSI, size);

    int k=1;
    while (k < dimensions[0]){
        frames[k] = &HSI[k*dimensions[3]];
        i_retCode = AT_QueueBuffer(Hndl, &HSI[k*dimensions[3]], ...
           dimensions[3]);
        k++;
    }

    /*Set proper frame count*/
    AT_64 frameCount;
    frameCount = static_cast<AT_64> (dimensions[0]);
    i_retCode = AT_SetInt(Hndl, L"FrameCount", frameCount);
    i_retCode = AT_GetInt(Hndl, L"FrameCount", &frameCount);

    /*Set trigger mode*/
    int Index;
    AT_WC TriggerMode[256];
    i_retCode = AT_SetEnumIndex(Hndl, L"TriggerMode", 4);
    i_retCode = AT_GetEnumIndex(Hndl, L"TriggerMode", &Index);
    i_retCode =AT_GetEnumStringByIndex(Hndl, L"TriggerMode", Index, ...
       TriggerMode, 256);

    /*Check if camera is already acquiring*/
    AT_BOOL boole;
    i_retCode =AT_GetBool(Hndl, L"CameraAcquiring", &boole);

    /*Estimate waiting time: based on camera specifications*/
    double exp_time;
    AT_GetFloat(Hndl,L"ExposureTime", &exp_time);
    int waiting;
    waiting = static_cast<int> (1000*(exp_time+0.011));

    /*Start acquisition*/
    i_retCode = AT_Command (Hndl, L"AcquisitionStart");

    /* Set a clock to check performance*/
    clock_t trigger = clock();

    /*Send the software triggers*/
    for(int i = 0; i<dimensions[0]; i++){

        i_retCode = AT_Command(Hndl, L"SoftwareTrigger");


        /* *
         * Here should be placed the code to
         * communicate with the AOTF and select
         * wavelength for each frame.
         * The last one is the Background Image
         *
         */

        /*Sleep may be required to avoid losing Software triggers*/
        Sleep(waiting);
    }

    mexPrintf("\nTime trigger: %.9fs\n", (float)(clock() - ...
       trigger)/CLOCKS_PER_SEC);

    /*Set another clock to see which part is a bottle neck*/
    clock_t getImage = clock();

    unsigned char* WaitBuffer;
    int i_waitBufferSize;

    dimensions[5] = 1;

    /*Check that all the buffers have been written*/
    for (int j=0; j<dimensions[0]; j++){

        i_retCode = AT_WaitBuffer(Hndl, &WaitBuffer, &i_waitBufferSize, ...
            3000);

        if ( WaitBuffer != frames[j] ) {
            dimensions[5] = 0;
```

```
                j = dimensions[0]-1;
            }
        }

        /*Stop acquisition free buffers*/
        i_retCode = AT_Command (Hndl, L"AcquisitionStop");
        i_retCode = AT_Flush(Hndl);


        delete[] frames;
        mexPrintf("\nTime getImage: %.9fs\n", (float)(clock() - ...
            getImage)/CLOCKS_PER_SEC);

    }

uint64 extractMetadata(unsigned char* metadata, int length_metadata){

        /*Get the timestamps*/
        int *i_field, info[6], index, i_ticks[2];
        uint64 *ticks, timestamp;

        i_field = reinterpret_cast<int *>(metadata);
        index = length_metadata / 4;
        index--;
        for(int i=0; i<6; i++){
            info[i] = i_field[index-i];
            if(i==3 || i==2) i_ticks[i-2]=i_field[index-i];
        }

        ticks = reinterpret_cast<uint64 *>(i_ticks);
        timestamp = ticks[0];
        return timestamp;
    }
```

A.2.4 mSetLiveView

```
/*********************************************************************
* int resul = mSetLiveView(double handle);
     *input: handle to camera
     *output: resul contains if the operation has been succesfull or not

* It prepares the camera to start the Live View acquisition which
* means that frames will be acquired using a "light" function that does
* not do anything else
********************************************************************/

#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"
using namespace std;


void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double *handle, *d_retCode;
    mxArray *resul_out;
    int i_retCode;

    /*Get input*/
    handle = mxGetPr(prhs[0]);

    /*Allocate memory for output*/
    d_retCode = static_cast<double *> (mxMalloc(sizeof(d_retCode)));

    /*Hndl must be AT_H when passed to camera*/
    AT_H Hndl;
    Hndl = static_cast<AT_H> (*handle);

    /*Give a value to the output it may be changed later*/
    d_retCode[0] = 0;

    /*Make sure that the camera frameCount limit is high enough*/
    AT_64 frameCount = 1000;
    i_retCode = AT_SetInt(Hndl, L"FrameCount", frameCount);
    i_retCode = AT_GetInt(Hndl, L"FrameCount", &frameCount);

    /*Disable Metadata as it won't be used*/
    i_retCode = AT_SetBool(Hndl, L"MetadataEnable", 0);
    i_retCode = AT_SetBool(Hndl, L"MetadataTimestamp", 0);

    /* Make sure camera is in Software trigger mode*/
    int Index;
    AT_WC TriggerMode[256];
    i_retCode = AT_SetEnumIndex(Hndl, L"TriggerMode", 4);
    i_retCode = AT_GetEnumIndex(Hndl, L"TriggerMode", &Index);
    i_retCode = AT_GetEnumStringByIndex(Hndl, L"TriggerMode", Index, ...
        TriggerMode, 256);

    /* Tell camera to start an acquisition*/
    i_retCode = AT_Command(Hndl, L"AcquisitionStart");

    /*Check success change output if necessary*/
    if ( i_retCode == AT_SUCCESS){
        d_retCode[0] = 1;
        mexPrintf("Acquisition running");
    }
    else
    {
        mexPrintf("WARNING!! ");
    }
    /*Assign output*/
    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    resul_out = plhs[0];
    mxSetPr(resul_out, d_retCode);
```

```
}
```

A.2.5 mGetFrameLiveView

```
/***********************************************************************
* We get a frame, the AOTF should be already set up
*
* [imagen] = mGetFrameLiveView(double handle, double dimensions);

dimensions = [pix_height, pix_width, B_ImageSize, B_stride, success, ...
    transposed]

* Keep in mind (for all mex files):
* <> Use 0-based indexing as always in C or C++
* <> Indexing is column-based as in Matlab (not row-based as in C)
* <> Use linear indexing.  [x*dimy+y] instead of [x][y]
**********************************************************************/
#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"
#include <time.h>
#include <windows.h>

using namespace std;

/*Define uint64 for convenience*/
typedef unsigned long long uint64;

void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    mxArray *image_out;
    double *handle;
    unsigned char *image;
    unsigned short *uint16_image;
    mwSize *dimensions;

    int i_retCode;
    AT_64 height, width, stride, ImageSizeBytes;

    /* Count the time it takes to execute this function, useful
    * to improve it and analise its performance*/
    clock_t getImage = clock();

    /*Get inputs*/
    handle = mxGetPr(prhs[0]);
    dimensions = static_cast<mwSize *> (mxGetData(prhs[1]));

    /*Hndl must be AT_H when passed to camera*/
    AT_H Hndl;
    Hndl = static_cast<AT_H> (*handle);

    /*Allocate memory */
    int imageSizeBytes;
    imageSizeBytes = dimensions[2];
    image = static_cast<unsigned char ...
        *>(mxMalloc((imageSizeBytes+8)*sizeof(unsigned char)));

    /*Align the image in memory, usually not necessary: compiler does it*/
    //image = reinterpret_cast<unsigned char*>( ...
        (reinterpret_cast<unsigned long>( image ) + 7 ) & ¬0x7);

    /*Queue a buffer (image) and send a SoftwareTrigger to get an image*/
    i_retCode = AT_QueueBuffer(Hndl, image, dimensions[2]);

    AT_BOOL acqui;
    i_retCode = AT_GetBool(Hndl, L"CameraAcquiring", &acqui);

    i_retCode = AT_Command(Hndl, L"SoftwareTrigger");

    /* Receive the last buffer available*/
    unsigned char* WaitBuffer;
    int i_waitBufferSize;
    i_retCode = AT_WaitBuffer(Hndl, &WaitBuffer, &i_waitBufferSize, 3000);

    /* Check that its address is the same as that of image*/
    if ( WaitBuffer != image ) {
        mexPrintf("WARNING!!");
    }

    /*OUTPUT*/

    /*Image: first convert the buffer from char to unsigned short*/
    uint16_image = reinterpret_cast<unsigned short*>(image);
    plhs[0] = mxCreateNumericMatrix(0, 0, mxUINT16_CLASS, mxREAL);
    image_out = plhs[0];

    /* Assign the output*/
    mxSetData(image_out, uint16_image);
    mwSize row;
    row = dimensions[0];
    //row+=2;
    mxSetM(plhs[0], dimensions[1]);
    mxSetN(plhs[0], row);

    /*Calculate the time that took the function execution and print it*/
    mexPrintf("\nTime getImage: %.9fs\n", (float)(clock() - ...
       getImage)/CLOCKS_PER_SEC);
    return;
}
```

A.2.6 mStopLiveView

```
/*******************************************************************
* int resul = mStopLiveView(double handle);
     *input: handle to camera
     *output: resul contains if the operation has been succesfull or not

* It ends the acquisition that has been going on to quickly get frames
* to be shown in live view mode.
*******************************************************************/
#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"
using namespace std;


void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double *handle, *d_retCode;
    mxArray *resul_out;
    int i_retCode1, i_retCode2;

    /*Get input*/
    handle = mxGetPr(prhs[0]);

    /*Allocate memory for output pointer*/
    d_retCode = static_cast<double *> (mxMalloc(sizeof(d_retCode)));

    /*Hndl must be AT_H when passed to camera*/
    AT_H Hndl;
    Hndl = static_cast<AT_H> (*handle);

    /*Give a value to the output it may be changed later*/
    d_retCode[0] = 0;

    /*Stop acquisition and free the Queue buffer for next acquis*/
    i_retCode1 = AT_Command (Hndl, L"AcquisitionStop");
    i_retCode2 = AT_Flush(Hndl);

    /* Check if success */
    if (i_retCode1 == AT_SUCCESS && i_retCode2 == AT_SUCCESS){
        d_retCode[0] = 1;
        mexPrintf("Acquisition stopped");
    }
    else
    {
        mexPrintf("WARNING!! ");
    }
    /* Enable again Metadata Timestamp information in the frames*/
    i_retCode1 = AT_SetBool(Hndl, L"MetadataEnable", 1);
    i_retCode1 = AT_SetBool(Hndl, L"MetadataTimestamp", 1);

    /*Assign output value*/
    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    resul_out = plhs[0];
    mxSetPr(resul_out, d_retCode);
}
```

A.2.7 mModifiedCloseCamera

```
/*******************************************************************
*  mModifiedCloseCamera();

* Close camera and finalise library, when we dont know the Handle.
* it just tries with several handles until it succees
******************************************************************/

#include <matrix.h>
```

```
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"
using namespace std;

void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double d_retCode;

    AT_H Hndl=101;
    d_retCode = 1;

    /*Try different handle until it succeeds*/
    while (d_retCode != 0 && Hndl < 200){
        d_retCode = static_cast<double> (AT_Close(Hndl));
        Hndl++;
    }

    AT_FinaliseLibrary();
}
```

A.2.8 mCooling

```
/**********************************************************************
* Enable cooling
*
* double status = mCooling(double handle, double action);

action:
1: turn on the cooling
0: turn off the cooling
2: get the cooling status
```

```

status:
0 cool off
1 cool on
2 error
********************************************************************/
#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"
using namespace std;

void cooling(char *status_out, double handle, double action);

void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    char *status_out;
    double *handle, *action;

    /*Get inputs*/
    handle = mxGetPr(prhs[0]);
    action = mxGetPr(prhs[1]);

    /*Allocate memory for output*/
    status_out = static_cast<char *>(mxCalloc(256, sizeof(char)));

    /*Call cooling function*/
    cooling(status_out, *handle, *action);

    /*Output*/
    plhs[0] = mxCreateString(status_out);

    return;
}
```

```

void cooling(char *status_out, double handle, double action){

    int i_retCode=-1, temperatureStatusIndex = 0, temperatureCount = 0;
    double temperature = 0;
    AT_H Hndl;
    Hndl = static_cast<AT_H> (handle);

    /*Perform action*/
    if(action==1){
    i_retCode = AT_SetBool(Hndl, L"SensorCooling", AT_TRUE);
    }
    if(action==0){
    i_retCode = AT_SetBool(Hndl, L"SensorCooling", AT_FALSE);
    }

    /*Print result*/
    if(i_retCode == AT_SUCCESS || i_retCode == -1){
        mexPrintf("Cooling function status: ");
    }
    else{
        mexPrintf("WARNING! Switching cooling status failed ");
    }
    /*Get status*/
    AT_WC temperatureStatus[256];
    AT_GetEnumIndex(Hndl, L"TemperatureStatus", &temperatureStatusIndex);
    AT_GetEnumStringByIndex(Hndl, L"TemperatureStatus", ...
       temperatureStatusIndex, temperatureStatus, 256);

    char tempStatus[256];
    wcstombs(tempStatus,temperatureStatus,512);
    mexPrintf("%s", tempStatus);

    /*Write the status in the output string*/
    int k;
    for ( k=0; k<256; k++){
        status_out[k] = tempStatus[k];
    }
}
```

A.2.9 mExposureTime

```
/**********************************************************************
* int resul = mExposureTime(double handle, double exposure_time);

* Change exposure time
**********************************************************************/
#include <matrix.h>
#include <mex.h>
#include "atcore.h"
#include <iostream>
#include "stdlib.h"
using namespace std;


void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double *handle, *exposure_time_in, *d_retCode, *exp_time_applied;
    mxArray *resul_out, *exp_out;
    int i_retCode;

    /*Get inputs*/
    handle = mxGetPr(prhs[0]);
    exposure_time_in = mxGetPr(prhs[1]);

    /*Allocate memory*/
    d_retCode = static_cast<double *> (mxMalloc(sizeof(d_retCode)));
    exp_time_applied = static_cast<double *> (mxMalloc(sizeof(d_retCode)));

    AT_H Hndl;
    Hndl = static_cast<AT_H> (*handle);

    /*Set exposure time*/
    i_retCode = AT_SetFloat(Hndl, L"ExposureTime", *exposure_time_in);

    d_retCode[0] = 0;

    /*Check that it has been successfully set*/
    AT_GetFloat(Hndl, L"ExposureTime", exp_time_applied);
    if (*exposure_time_in == *exp_time_applied && i_retCode == AT_SUCCESS){
        d_retCode[0] = 1;
        mexPrintf("Exposure time set to: %f", *exp_time_applied);
    }
    else
    {
        mexPrintf("WARNING!! ");
    }

    /*Outputs*/
    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
    resul_out = plhs[0];
    mxSetPr(resul_out, d_retCode);

    plhs[1] = mxCreateDoubleMatrix(1, 1, mxREAL);
    mxSetPr(plhs[1], exp_time_applied);
}
```

Appendix B

MEX files and Andor SDK

The purpose of this appendix is to explain how MEX files and the Software Developing Kit provided by Andor work. Therefore, it can be seen as a complement to the former Appendix, as it may help the reader to get a better understanding of the code presented there.

B.1 MEX-Files

It was mentioned in Section 2.5.1 that Matlab can call functions written in C using MEX-files which are dynamically linked subroutines that the Matlab interpreter loads and executes. The MEX-file contains only one function or subroutine that can be called from the Matlab program using the name of the file. Further information might be found at the Matlab online documentation (Mathworks, 2015). An example is presented to illustrate how MEX-files work. It is in fact a part of the program and, thus, it can be found in the Appendix, Section A.2.2. The reader may consult the source code there to get a more complete view. First of all, MEX files require the use of the libraries Matrix and MEX, therefore the lines:

```
#include <matrix.h>
#include <mex.h>
```

are included at the beginning of the source code to include those header files. The file mex.h contains the Matlab API function declarations. Then, in a similar way to the `main()` function that is present in every program in C, Matlab uses the gateway routine, `mexfunction()`, as the entry point to the function. The name of the source file containing mexFunction is the name of the MEX-file (where here MEX-file takes the meaning of dynamically linked subroutine, see 2.5.1 to refresh about the different meanings of MEX-file) , and, hence, the name of the function called in Matlab. The signature for `mexfunction()` is:

```
void mexfunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
```

The meaning of each parameter, according to the official Matlab documentation, is:

- `nlhs`: Number of output (left-side) arguments, or the size of the `plhs` array.
- `plhs`: Array of output arguments.
- `nrhs`: Number of input (right-side) arguments, or the size of the `prhs` array.
- `prhs`: Array of input arguments.

`prhs` and `plhs` are declared as type mxArray *, which means they point to Matlab arrays. They are vectors that contain pointers to the arguments of the MEX-file. The keyword `const`, which modifies `prhs`, means that the MEX-file does not modify the input arguments. Therefore, the input parameters (found in the `prhs` array) are read-only and must not be modified. The Matlab language works with a single object type, for all kind of variables, the Matlab array. In C, it is declared to be of type `mxArray`. The `mxArray` structure contains several information about the array: its type and dimensions and the data associated with it. Besides, if it is numeric, whether the variable is real or complex; if it is sparse, its indices and nonzero maximum elements; and if it is a structure or object, the number of fields and field names. The API functions in the Matrix Library are needed to access the `mxArray` structure and they allow to create, read and query information about the Matlab data in the MEX-files.

Like Matlab functions, a MEX-file gateway routine passes Matlab variables by reference. However, these arguments are C pointers. A *pointer* to a variable is the *address* (location in memory) of the variable. Information about working with pointers in C can be found in Kernighan and Ritchie (1988). In order to prevent memory leaks which provoke unexpected results, memory must be managed carefully. For that purpose, some rules should be followed when treating with an mxArray:

- If it is an input argument, it exists outside the scope of the MEX-file. Memory must not be freed for any mxArray in the `prhs` parameter.
- If it is an output argument, the memory allocated and the data necessaries to create an mxArray exist beyond the scope of the MEX-file. Memory must not be freed for any mxArray returned in the `plhs` (output) parameter either.
- If it is a local variable, the memory that is first allocated using functions such as `mxCreate*`, `mxCalloc` or associated functions, must be deallocated. Functions such as `mxDestroyArray` or `mxFree` can be used.

In fact the Matlab memory manager keeps a record of all memory allocated by the function and automatically frees it when control returns to the Marlab prompt. However, it is more efficient and recommended to perform this task manually within the MEX-file.

Continuing with the example, this function is called from Matlab as: `resul = mCloseCamera(handle)` where `resul` is the output parameter (its type in C is integer) and `handle` is the input parameter (type `double`). Therefore, the parameters passed to the `mexfunction` are: `nlhs = nrhs ... = 1`, as there is one input parameter and one output parameter. Their addresses are passed as `mxArray` pointers in `prhs[0]` for the input parameter (`handle`) and `plhs[0]` for the output parameter (`resul`). We have inside `mexfunction()`:

```
    double *handle, *d_retCode;
    mxArray *resul_out;

    /* Get input and allocate memory for output*/
    handle = mxGetPr(prhs[0]);
    d_retCode = static_cast<double *> (mxMalloc(sizeof(d_retCode)));
```

`handle` and `d_retCode` are C pointers whose declared type is `double`. `mxGetPr(prhs[0])` allows to access the real data in the `mxArray prhs[0]` by providing the starting address. Now, handle is a pointer to the starting address of the data in prhs[0]. Once the starting address is known, any other element in the array can be accessed. `mxGetData(prhs[0])` can be used in a similar way, as in Section A.2.3 line 50. See the official Matlab documentation (Mathworks, 2015) to know about the specific differences between these functions. In the line 6 of the code showed above, `mxMalloc(sizeof(d_retCode))` is used to allocate dynamic memory. It has the same role as that of the ANSI C `malloc` function, but it has to be used instead of that one when working with MEX-files. Later in the code, once `d_retCode[0]` has a value assigned, the output is created and assigned data. So far, plhs[0] is a null pointer that points at nothing because the output `resul` has not been created yet. This is solved here:

```
1      /*Output*/
2      plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
3      resul_out = plhs[0];
4      mxSetPr(resul_out, d_retCode);
```

`mxCreateDoubleMatrix(m, n, mxREAL)` creates an m-by-n real `mxArray` and initializes each element to 0. Therefore, after line 2, `plhs[0]` points to a 1-by-1 mxArray initialized to zero. Then with line 3, the `mxArray` `resul_out` also points there. `mxSetPr(resul_out, d_retCode)` sets the real data of the output array (`resul_out`, or what is the same `plhs[0]`) with the data stored in `d_retCode`. `d_retCode` must be in dynamic memory, as we have shown that it is. See that the third line could be skipped by using directly `mxSetPr(plhs[0], d_retCode)`, however it is a bit more clear using `resul_out`.

All the MEX-files found in the Appendix A.2 and developed within this work have a structure similar to the one that has been commented here. In some cases the actual functions used to treat inputs, outputs or `mxArray`'s may differ from those shown here, depending on the data type or dimensions. However, they will be equal in essence and, in any case, Mathworks (2015) explains the specific features and reasons behind using every mx-function.

Once the source code is finished, it must be compiled to create an executable program. The Matlab command `mex('-g', 'mCloseCamera.cpp')` does this work producing a binary file whose extension is platform dependent. Options available for this command can be consulted in the Matlab documentation. Some of them will be used later to include libraries, while the

`'-g'` option shown here adds symbolic information and is necessary for debugging.

Debugging is the process of finding and eliminating bugs, or defects, in a computer program that make it behave unexpectedly. It is an important stage on the developing process. Both Matlab and Microsoft Visual Studio, which is the integrating development environment used to write the C code, offer tools to debug their code. However, as the MEX-files are called from Matlab, they can not be debugged independently. Fortunately, Microsoft Visual Studio offers the solution, which is attaching to a process running outside of Visual Studio, in our case the process Matlab. Then, both tools work together and breakpoints can be set both in the Matlab part or in the MEX-file to perform the required debugging.

B.2 Andor SDK

As introduced before, Andor provides a Software Development Kit which is a well documented API that allows to control the camera. It has been designed under the idea that integrating the camera is just one component of a larger system solution. Andor® and Andor® (2014a) are the manuals of the SDK and the camera, respectively, in which the reader may find deeper information and examples regarding this and the next section (4.2).

The API needs some dynamic-link libraries (*.dll files) given with the kit to work. These files must be placed in the same folder which the application is running from. They are architecture specific, in our case we had to use the 64-bits versions. Then, the next step is to include the header file `"atcore.h"` in the source code and include the library *atcorem.lib*. Therefore, the line `#include "atcore.h"` appears at the beginning of all our MEX-files. Besides, to successfully compile the MEX-file the sintax of the `mex` command must vary. Now, it is:

```
mex('-g', ipath, library, 'atcorem.lib', 'mCloseCamera.cpp')
```

where `ipath` and `library` are two strings taking the following form:

```
path = 'C:\Users\vnir1600\Documents\MATLAB';
ipath = ['-I' path];
```

```
library = ['-L' path];
```

The very first API call must be `AT_InitialiseLibrary`, and the very last call must be `AT_FinaliseLibrary`. These functions will prepare the API for use and free resources when no longer needed. In the User Interface they are called in `[handle, warning] = mOpenCamera(exposure_` and in `[resul] = mCloseCamera(handle)`, which are called in `acqui_CreateFcn()` and in `acqui_DeleteFcn()`, what is at the moment of opening the GUI and at the moment of closing it. Exactly the same procedure follows for the API functions `AT_Open` and `AT_Close`, which provide and release a camera handle respectively (represented by the data type `AT_H`). Right after opening the camera the cooling mechanism is activated to ensure a low noise level in the images.

Every API function returns an error code (integer) when called. Each return code that could possibly be returned is listed in the atcore.h and documented in the mentioned reference for the SDK. It is recommended that a user check every return code before moving on to the next statement. Within our MEX-files, return codes have sometimes be checked, but not always. A possible improvement of the application may include checking every return code and following the proper procedure for each case, which could vary from aborting the current operation to printing information related about the error in console.

The SDK3 API can be divided into several sets of functions, each controlling a particular aspect of camera control. There are sections in the API for opening a handle to a camera, for buffer management and for accessing the features that every camera exposes. Each feature that a camera exposes to the user has a particular type that represents how that feature is controlled. The feature types are: *Integer*, *Floating Point*, *Boolean*, *Enumerated*, *Command* and *String*. Each of them having its own set of functions to manage the feature, i. e. get or set its value, check maximum and minimum valid values, maximum length, etc. Examples of these functions are shown here:

```
int AT_GetIntMax(AT_H Hndl, AT_WC* Feature, AT_64* MaxValue); // integer
int AT_SetFloat(AT_H Hndl, AT_WC* Feature, double Value); // For ...
    Floating Point
int AT_GetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL* Value); // Boolean
int AT_GetEnumIndex(AT_H Hndl, AT_WC* Feature, int* Value); // Enumerated
int AT_Command(AT_H Hndl, AT_WC* Feature); // For Command
int AT_GetStringMaxLength(AT_H Hndl, AT_WC* Feature, int* ...
    MaxStringLength); // For String
```

There are some general functions that can be used to get information about any feature, they might have been used during the development process to, for example, check if a certain feature was or not implemented, these are:

```
int AT_IsImplemented(AT_H Hndl, AT_WC* Feature, AT_BOOL* Implemented);
int AT_IsReadOnly(AT_H Hndl, AT_WC* Feature, AT_BOOL* ReadOnly);
int AT_IsReadable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Readable);
int AT_IsWritable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Writable);
```

More relevant functions for us are those which allow to manage buffers in which the acquired images are stored. SDK maintains two queues, which are used to manage the transfer of image data to the application. Both queues operate in a First-in-First-out (FIFO) basis and are used to store the addresses of blocks of memory allocated by the application. First, the `AT_QueueBuffer()` API function call lets SDK knows what memory to use for the upcoming acquisition. Therefore it manages the input queue. Multiple buffers can be queued to the SDK before an acquisition starts if a sequence of frames is being acquired, doing this is as simple as calling the function multiple times with different buffers. The second queue (output queue) is used to store the application defined buffers after they have had images copied into them. In this case the SDK adds buffers to this queue which can then be retrieved by the application using `AT_WaitBuffer()`. This function will retrieve the next buffer from the output queue and return the address in the second parameter; the size of the buffer will also be returned in the `PtrSize` parameter. Prototypes of these functions and `AT_Flush()` which allows to retrieve both queues at the same time, are shown below:

```
int AT_QueueBuffer(AT_H Hndl, AT_U8 *Ptr, int PtrSize);
int AT_WaitBuffer(AT_H Hndl, AT_U8 **Ptr, int* PtrSize, unsigned int ...
    Timeout);
```

```
int AT_Flush(AT_H Hndl);
```

The MEX-file mGetHSISoftware, shown in A.2.3, is an illustrative example about how to work with MEX-files and control the camera. It requires reading data from several input parameters. Among the tasks performed there, allocating memory for three different multidimensional output arrays, managing buffers to store images or using camera commands, are included. Also, *timestamp* information attached with the frames is extracted.

Appendix C

Starting point: RF driver communication

The code presented in this Appendix is thought to be a starting point of the software to control the RF driver which should be written in C. A version in Matlab is available and provided here, as well as the starting point of its *translation* to C.

C.1 Matlab AOTF

```
%Code available to control AOTF in matlab

% Full control AOTF
freq = (sqrt(sin(200./wavelengths)+1)).^(10.55)*20.91950466; %frequency ...
    in (MHz)
Nlam = length(wavelengths);

% Initialize
[ser]=initAOTF(mPow);

% Set frequency
liveflag = 0; channel = 6;
[chFRhex] = GetChRF(liveflag, channel, freq(1), 0); %%% Amplitude 0 so ...
    f is not important
[out] = sendHex2RF(ser,chFRhex);
```

```

% Example of HSI loop
liveflag = 0; channel = 6;
for i = 1:Nlam
    %set frequency
    [chFRhex] = GetChRF(liveflag, channel, freq(i), mPow); %%% MPOW ...
        SHOULD BE AMPLITUDE. CHECK
    [out] = sendHex2RF(ser,chFRhex);
    %get image
    trigger(vid); %acqisition
    wait(vid,pauseLen,'logging');
    %time(i)=now;
end


%End aotf
[chFRhex] = GetChRF(0,6, freq(1), 0);
[out] = sendHex2RF(ser,chFRhex);
[mastRFHex] = SetMasterRF(0,0);
[out] = sendHex2RF(ser,mastRFHex);
fclose(ser)
delete(ser)
clear ser;
```

```
% initAOTF


function [aotf] = initAOTF(mPow)
%HSI acquisition function for AOTF G&H - modified version because of frame
%problems
% INPUT:
%   mPow - master channel power (0-100)

% OUTPUT:
%   ser - AOTF file controller
```

```
%

%init AOTF
[info, N] = FindDevice;
serNum = info(end); %obtained from FindDevice
serName = ['COM',num2str(serNum)];
aotf = ...
    serial(serName,'BaudRate',9600,'DataBits',8,'Parity','none','StopBits',1,'FlowContro
fopen(aotf);
[mastRFHex] = SetMasterRF(0,mPow);
[out] = sendHex2RF(aotf,mastRFHex);

path = 'C:\Users\hyspex\Documents\MATLAB\Variables\aotf.mat';
save(path, 'aotf');

end
```

```
function [chFRhex] = GetChRF(liveflag,ch, freq, amp)
%Returnes G&H 16 channel hex for channel configuration
% INPUT:
%    liveflag  - 0=live update, 1=profiles upload
%    ch        - channel number from 0 to 15
%    freq      - frequency in MHz
%    amp       - amplitude in % max
% OUTPUT:
%    chFRhex   - structure with corresponding hexes

if(liveflag==0)
    fb = '05'; %command hex byte for immediate execution
else
    fb = '06'; %command hex byte for profile load
end

% set channel
chHex = getChHex(ch);
```

```
chFRhex(1) = {[fb,chHex,'000000']};

% set frequency
chFHex = getChFHex(ch);
f14 = dec2hex(int32(2^32*freq/500));
if(freq==0)
    f14='00000000';
end
chFRhex(2) = {[fb,chFHex,f14]};

% set phase - first two bytes sets the phase, here set to 0
chPHex = getChPHex(ch);
chFRhex(3) = {[fb,chPHex,'00000000']};

% set amplitude
chAHex = getChAHex(ch);
coeff = 10.23;
if(ch == 1 || ch == 5 || ch == 9 || ch == 13)
    coeff = 2.55;
end
a12 = dec2hex(int16((amp*coeff)+4096));%multiplication by coeff might ...
    be wrong
chFRhex(4) = {[fb,chAHex,'00',a12,'00']};

%change to hexMatrix
s = chFRhex;
chFRhex = [];
for i=1:4
    is = s{i};
    a=[];
    for j=1:6
        a = [a;is(2*j-1:2*j)];
    end
    chFRhex{i}=a;
end
end
```

```

%function returns Hex for a channel
function chHex = getChHex(ch)
    switch(ch)
        case 0
            chHex = '0016';
        case 1
            chHex = '0026';
        case 2
            chHex = '0046';
        case 3
            chHex = '0086';
        case 4
            chHex = '2016';
        case 5
            chHex = '2026';
        case 6
            chHex = '2046';
        case 7
            chHex = '2086';
        case 8
            chHex = '4016';
        case 9
            chHex = '4026';
        case 10
            chHex = '4046';
        case 11
            chHex = '4086';
        case 12
            chHex = '6016';
        case 13
            chHex = '6026';
        case 14
            chHex = '6046';
        case 15
            chHex = '6086';
    end
end
% frequency Hex
function chFHex = getChFHex(ch)
    if(ch<4)
        chFHex = '04';
    elseif(ch>3 && ch<8)
        chFHex = '24';
    elseif(ch>7 && ch<12)
        chFHex = '44';
    elseif(ch>11)
        chFHex = '64';
    end
end

% phase Hex
function chPHex = getChPHex(ch)
    if(ch<4)
        chPHex = '05';
    elseif(ch>3 && ch<8)
        chPHex = '25';
    elseif(ch>7 && ch<12)
        chPHex = '45';
    elseif(ch>11)
        chPHex = '65';
    end
end

% amplitude Hex
function chAHex = getChAHex(ch)
    if(ch<4)
        chAHex = '06';
    elseif(ch>3 && ch<8)
        chAHex = '26';
    elseif(ch>7 && ch<12)
        chAHex = '46';
    elseif(ch>11)
        chAHex = '66';
    end
end
```

```
 function [out] = sendHex2RF(ser,inst)
%Sends Hex instruction to G&H 16channel RF
% INPUT:
%   ser  - serial port ID
%   inst - a 6 bytes long Hex instruction as [hex1;...;hex6] structure

%{
serNum = 3; %obtained from FindDevice
serName = ['COM',num2str(serNum)];
ser = ...
    serial(serName,'BaudRate',9600,'DataBits',8,'Parity','none','StopBits',1,'FlowContro
fopen(ser);

fclose(ser)
delete(ser)
clear ser;
%}

s.Timeout = 1;

%find inst structure length
N = size(inst,2);

for i = 1:N
    initI = inst{i};
    %sending command ended with CR = 13
    fwrite(ser, [uint8(hex2dec(initI));uint8(13)]);
end

out = 0;
```

```
end
```

```
function [mastRFHex] = SetMasterRF(liveflag,PWlevel)
%UNTITLED5 Summary of this function goes here
%   Detailed explanation goes here

if(liveflag==0)
    fb = '05'; %command hex byte for immediate execution
else
    fb = '06'; %command hex byte for profile load
end

%set master gain to desired level
pw = dec2hex(uint8(PWlevel/100*255));
if(size(pw,2)==1)
    pw = ['0',pw];
end
s = [fb;'81';pw;'00';'00';'00'];
mastRFHex{1} = s;
end
```

C.2 C source files

```
/* In this file we initialize the AOTF driver*/


#include <iostream>
#include <stdlib.h>
#include <windows.h>
#include <stdio.h>
#include <time.h>
#include <string>
```

```

using namespace std;

int SetMasterRF(char *mastRFHex, int liveflag,float PWlevel);

int initAOTF(int mPow){

    DCB dcb={0};
    HANDLE hCom;
    BOOL fSuccess;
    char *pcCommPort = "COM1";


    /***************************************CommTimeouts********************************
    COMMTIMEOUTS timeouts={0};
    timeouts.ReadIntervalTimeout=50;
    timeouts.ReadTotalTimeoutConstant=50;
    timeouts.ReadTotalTimeoutMultiplier=10;
    timeouts.WriteTotalTimeoutConstant=50;
    timeouts.WriteTotalTimeoutMultiplier=10;


    /*******************************************Handle***********************************
    hCom = CreateFile( pcCommPort,
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ,    // must be opened with exclusive-access
        NULL, // no security attributes
        OPEN_EXISTING, // must use OPEN_EXISTING
        FILE_ATTRIBUTE_NORMAL,    // not overlapped I/O
        NULL  // hTemplate must be NULL for comm devices
        );



    /***************************************SET*UP*COM*PORT*****************************
    if (hCom == INVALID_HANDLE_VALUE)
    {
        printf ("CreateFile failed with error %d.\n", GetLastError());
        //return (1);
    }

    if(!SetCommTimeouts(hCom, &timeouts))
    {
        /*Well, then an error occurred*/
    }

    fSuccess = GetCommState(hCom, &dcb);

    if (!fSuccess)
    {
        /*More Error Handling*/
        printf ("GetCommState failed with error %d.\n", GetLastError());
        //return (2);
    }


    dcb.BaudRate = 9600;     // set the baud rate
    dcb.ByteSize = 8;             // data size, xmit, and rcv
    dcb.Parity = NOPARITY;        // no parity bit
    dcb.StopBits = ONESTOPBIT;    // one stop bit
    fSuccess = SetCommState(hCom, &dcb);

    if (!fSuccess)
    {
        printf ("SetCommState failed. Error: %d.\n", GetLastError());
        //return (3);
    }

    printf ("Serial port %s successfully configured.\n", pcCommPort);

    /************************************************************************************

    int i_retCode;
    char mastRFHex[12]="\0";

    i_retCode = SetMasterRF(mastRFHex, 0, mPow);




    /*
    [mastRFHex] = SetMasterRF(0,mPow);
    [out] = sendHex2RF(aotf,mastRFHex);*/


    return 0;

}


int SetMasterRF(char *mastRFHex, int liveflag,float PWlevel){

    int i_retCode=1;
    char fb[3];

    if(liveflag == 0)
        strcpy(fb,"05");
    else
        strcpy(fb,"06");


    char hexPower[3]="00";
    if( PWlevel < 100 ){

        int power;
        float prod;
        prod = PWlevel*255/100;
        prod = round(prod);

        power = static_cast<int>(prod);

        int aux,i=1;
        while(power!=0 && i>0){
            aux = power % 16;

            //To convert integer into character
            if( aux < 10)
                aux =aux + 48;
            else
                aux = aux + 55;

            hexPower[i]= aux;
            i--;
            power = power / 16;
        }
    }
    else
        i_retCode = 0;

    strcat(mastRFHex, fb);
    strcat(mastRFHex, "81");
    strcat(mastRFHex, hexPower);
    strcat(mastRFHex, "000000");

    return i_retCode;
}
```

Appendix D

Matlab code for testing

The code presented in this Appendix was used to perform test of the Gooch&Housego hyper-spectral imaging system.

D.1 Grid

```
%Load data
info = readHyperHeader('Grey450-808.hdr');
dark = multibandread('Dark450-808.bip', ...
    [info.lines,info.samples,info.bands], info.data_type, 0, ...
    info.interleave, 'ieee-le');
gray = multibandread('Grey450-808.bip', ...
    [info.lines,info.samples,info.bands], info.data_type, 0, ...
    info.interleave, 'ieee-le');
img = multibandread('Grid450-808.bip', ...
    [info.lines,info.samples,info.bands], info.data_type, 0, ...
    info.interleave, 'ieee-le');
[imgD] = SubDark(img,dark);
[grayD] = SubDark(gray,dark);

%Stract wavelengths vector
lam = zeros(1,info.bands);
```

```
a=1;
for i=2:8:1439
    lam(a)=str2double(info.band_names(i:i+2));
    a = a+1;
end

%Reflectance image
imgR = img2R(imgD,grayD, lam);

%Data set that will be modified
ModifiedGrid = zeros(1000, 1000, 180);
for i=1:180;
    ModifiedGrid(:,:,i) = imgR(i,:,:);
end

% Grid draw: maximum value
range = [−6 : 6];
columna = [30 91 122 248 375 502 661 789 881 974];
verticalDistortion = zeros(180, length(columna));
for l=1:length(columna)
    for k = 1:180;
        for i=50:870
            ref = imgR(k, i, columna(l)+range);
            [¬, I] = min(ref);
            offset = range(I);
            verticalDistortion(k, l) = verticalDistortion(k, l) + ...
                abs(offset)^2;
            ModifiedGrid(i, columna(l)+offset, k)=0;
        end
    end
end

row = [108 203 329 424 457 520 615 710 804];
horizontalDistortion = zeros(180, length(row));
for l=1:length(row)
    for k = 1:180;
```

```
          for i=1:1000
              ref = imgR(k, row(l)+range, i);
              [¬, I] = min(ref);
              offset = range(I);
              horizontalDistortion(k, l) = horizontalDistortion(k, l) + ...
                  abs(offset)^2;
              ModifiedGrid(row(l)+offset, i, k)=0;
          end
      end
  end

  % Grid draw: straight lines
  ModifiedGrid(:, columna, :) = 0;
  ModifiedGrid(row, :, :) = 0;

  % Normalize vertical and horizontal distortion: 1 means no distortion
  Normalization = min([min(min(verticalDistortion)) ...
      min(min(horizontalDistortion))]);
      verticalDistortion = verticalDistortion ./ Normalization;
      horizontalDistortion = horizontalDistortion ./ Normalization;

  % Plot distortion for different wavelengths
  color = [0 0 255; 102 178 255;...
           0 255 0; 102 255 178;...
           255 0 0; 255 178 102]/255;
  t=1;
  s=1;
  for h = 10:30:180
  plot(columna, verticalDistortion(h, :), 'Color', color(t,:));
  strings{s} = [num2str(lam(h)) 'nm'];
  hold on;
  t = t+1; s = s+1;
  end
  plot(columna, ones(length(columna)), 'Color', [153 153 ...
      255]/255,'LineStyle', '--')
  hold on;
```

```
plot([499.99:0.01:500.01],[0 20 0],  'Color', [255 153 ...
    204]/255,'LineStyle', ':')
hold on;
xlabel('Pixel');
ylabel('Vertical distortion');
strings{s}='No distortion';
strings{s+1}='Center';
legend(strings)
axis([0 1000 0 20])

figure(2);
t=1;
s=1;
for h = 10:30:180
plot(row, horizontalDistortion(h, :), 'Color', color(t,:));
strings{s} = [num2str(lam(h)) 'nm'];
t = t+1;s = s+1;
hold on;
end
plot(row, ones(length(row)), 'Color', [153 153 255]/255,'LineStyle', '--')
hold on;
plot([456.99:0.01:457.01],[0 20 0],  'Color', [255 153 ...
    204]/255,'LineStyle', ':')
hold on;
xlabel('Pixel');
ylabel('Horizontal distortion');
strings{s}='No distortion';
strings{s+1}='Center';
legend(strings)
axis([0 1000 0 20])

% Calculate total distortion and plot it
averageVert = zeros(1, 180);
for i=1:180
averageVert(i) = ...
    sum(verticalDistortion(i,:))/length(verticalDistortion(1,:));
```

```
end
averageHor = zeros(1, 180);
for i=1:180
averageHor(i) = ...
    sum(horizontalDistortion(i,:))/length(horizontalDistortion(1,:));
end

smoothedVert = averageVert;
smoothedHor = averageHor;
for i=3:177
    smoothedVert(i) = 1/5*(averageVert(i-2) + averageVert(i-1) + ...
        averageVert(i) +...
         averageVert(i+2) + averageVert(i+1));
    smoothedHor(i) = 1/5*(averageHor(i-2) + averageHor(i-1) + ...
        averageHor(i) +...
         averageHor(i+2) + averageHor(i+1));
end
figure(3);
plot(lam, smoothedVert);
hold on;
plot(lam, smoothedHor);
xlabel('Wavelength');
ylabel('Distortion');
strings = {'Vertical', 'Horizontal'};
legend(strings);

figure(4);
columna6 = verticalDistortion(:, 6)
N = 3;
n = floor(N/2)
n_vect = [-n:n];
for i=1+n:180-n
    columna6(i) = 1/3*(sum(columna6(i+n_vect)))
end
plot(lam, columna6)
xlabel('Wavelength');
```

```
ylabel('Distortion central column');
```

D.2 Color chart

```
%TestColor

%Load spectral data
letters = 'FGHI';
for j=1:length(letters)
    for i=1:6
        strings{(6*(j-1))+i}=[letters(j) num2str(i) '.ProcSpec'];
    end
end

R_spec = zeros(length(strings), 2048);
slam = zeros(1, 2048);
for i = 1:length(strings)
[R_data(i,:), slam(:)] = Proc2R(['Spectrometer\' strings{i}]);
end
R = R_data;
R(:,1) = R(:,2);

%Load HSI data
info = readHyperHeader('Grey450-808.hdr');
dark = multibandread('Dark450-808.bip', ...
    [info.lines,info.samples,info.bands], info.data_type, 0, ...
    info.interleave, 'ieee-le');
gray = multibandread('Grey450-808.bip', ...
    [info.lines,info.samples,info.bands], info.data_type, 0, ...
    info.interleave, 'ieee-le');
img = multibandread('Color450-808.bip', ...
    [info.lines,info.samples,info.bands], info.data_type, 0, ...
    info.interleave, 'ieee-le');
```

```
[imgD] = SubDark(img,dark);
[grayD] = SubDark(gray,dark);

%Stract wavelengths vector
ilam = zeros(1,info.bands);
a=1;
for i=2:8:1439
    ilam(a)=str2double(info.band_names(i:i+2));
    a = a+1;
end

%Reflectance image
imgR = img2R(imgD,grayD, ilam);
for k=1:length(imgR(:,1,1))
    iColor(:,:,k) = imgR(k,:,:);
end

% Select area of each of the squares that will be analyzed
squares{1}='red';
coordinates(1,:) = floor([500.208528892673 410.362660250495 ...
    136.780142916271 131.973173037425]);
imgColorRed = ...
    imgR(:,coordinates(1,2):coordinates(1,2)+coordinates(1,4),coordinates(1,1):coordinate
squares{2}='blue';
coordinates(2,:) = floor([656.130567873183 569.072442739442 ...
    132.337000249194 127.686179154175]);
imgColorBlue = ...
    imgR(:,coordinates(2,2):coordinates(2,2)+coordinates(2,4),coordinates(2,1):coordinate
squares{3}='yellow';
coordinates(3,:) = floor([45.9356653244576 570.255557252473 ...
    121.787639168145 117.507562392268]);
imgColorYellow= ...
    imgR(:,coordinates(3,2):coordinates(3,2)+coordinates(3,4),coordinates(3,1):coordinate

for i=1:116 ...
for j=1:122 ...
```

```
imgColorRed(:,i,j) = imgColorRed(:,i,j)/norm(imgColorRed(:,i,j));
imgColorBlue(:,i,j) = imgColorBlue(:,i,j)/norm(imgColorBlue(:,i,j));
imgColorYellow(:,i,j) = imgColorYellow(:,i,j)/norm(imgColorYellow(:,i,j));
end;end

%Get spectrum
specRed = mean(mean(imgColorRed,3),2);
specBlue = mean(mean(imgColorBlue,3),2);
specYellow = mean(mean(imgColorYellow,3),2);

%plot
n = 25;
Xred = R(10, n:802);
Yred = 2*interp1(ilam, specRed, slam(n:802));
Hred = Yred./Xred;
plot(slam(n:802),Xred)
hold on
plot(slam(n:802),Yred')
hold on
plot(slam(n:802),Hred','--')
axis([500 800 0.5 1.5])
xlabel('Wavelength')
ylabel('Reflectance')
title('Spectrum red square')
legend('H response')

figure(2);
n = 25;
Xblue = R(17, n:802);
Yblue = 2*interp1(ilam, specBlue, slam(n:802));
Hblue = Yblue./Xblue;
plot(slam(n:802),Xblue)
hold on
plot(slam(n:802),Yblue')
hold on
plot(slam(n:802),Hblue','--')
```

```
axis([500 800 0.5 1.5])
xlabel('Wavelength')
ylabel('Reflectance')
title('Spectrum blue square')
legend('H response')
figure(3);
n = 25;
Xyellow = R(13, n:802);
Yyellow = 8*interp1(ilam, specYellow, slam(n:802));
Hyellow = Yyellow./Xyellow;
plot(slam(n:802),Xyellow)
hold on
plot(slam(n:802),Yyellow')
hold on
plot(slam(n:802),Hyellow','--')
axis([500 800 0.5 1.5])
xlabel('Wavelength')
ylabel('Reflectance')
title('Spectrum yellow square')
legend('H response')
```

Bibliography

Afromowitz, M. A. (1988). Multispectral imaging of burn wounds: a new clinical instrument for evaluating burn depth. *IEEE Trans. Biomed. Eng. 35(10).*

Andor®. *Software Development Kit 3 v3.9.* ANDOR.

Andor®(2014a). *Zyla sCMOS Hardware Guide 1.5.* Andor Technology.

Andor®(2014b). *Zyla sCMOS Specifications.* Andor Technology.

B. S. Sorg, O. D. and Cao, Y. (2005). Hyperspectral imaging of hemoglobin saturation in tumor microvasculature and tumor hypoxia development. *Biomedical Optics 10(4).*

Bahaa E. A. Saleh, M. C. T. (1991). *Fundamentals of Photonics.* John Wiley & Sons, Inc.

C Stedham, M Draper, J. W. E. W.-C. P. (2008). A novel acousto-optic tunable filter for use in hyperspectral imaging systems. *Physics and Simulation of Optoelectronic Devices XVI.*

Conrady, A. E. (1919). Decentred lens-systems. *Monthly notices of the Royal Astronomical Society 79.*

Dennis R. Suhre, L. J. D. and Gupta, N. (2004). Telecentric confocal optics for aberration correction of acousto-optic tunable filters. *APPLIED OPTICS Vol. 43.*

Dr. Colin Coates, Dr. Boyd Fowler, D. G. H. (2009). scmos white paper: Scientific cmos technology, a high-performance imaging breakthrough. *www.scmos.com.*

Gat, N. (2000). Imaging spectroscopy using tunable filters: a review. *Proc. SPIE 4056.*

Gooch&Housego (2014a). *Acousto-Optic Tunable Filter TF625-350-2-11-BR1A.* Gooch & Housego ®.

Gooch&Housego (2014b). *Driver for Acousto-Optic Tunable Filter MSD0XX-YYY-10UC-16x1.* Gooch & Housego ®.

Gowen, A. A. (2007). Hyperspectral imaging, an emerging process analytical tool for food quality and safety control. *Trends Food Sci. Technol. 18(12).*

Ingvaldsen, A. T. R. (2012). An imaging spectrometer using an acousto-optic tunable filter. Master's thesis, Norwegian University of Science and Technology.

Jaka Katrasnik, F. P. and Likar, B. (2013). A method for characterizing ilumination systems for hyperspectral imaging. *OPTICS EXPRESS.*

Joan Vila-Frances, Emilio Ribes-Gomez, C. I.-L. L. G.-C. J. M.-M. J. A.-L. J. C.-M. (2006). Configurable-bandwidth imaging spectrometer based on an acousto optic tunable filter. *Proc. of SPIE.*

Joan Vila-Frances, Javier Calpe-Maravilla, L. G.-C. J. A.-L. (2010). Improving the performance of acousto-optic tunable filters in imaging applications. *Journal of Electronic Imaging.*

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language.*

M. B. Sinclair, D. M. Haaland, J. A. T. and Jones, H. D. T. (2006). *Appl. Optics 45(24).*

M. E. Martin, M. B.Wabuyele, K. C. P. K. M. P.-M. P. B. O. G. C.-D. W. R. C. D. and T. Vo-Dinh, A. (2006). *Biomedical Engineering 34(6).*

M. Govender, K. C. and Bulcock, H. (2007). A review of hyperspectral remote sensing and its application in vegetation and water resource studies. *Water SA 33(2).*

Mathworks (2015).

P. De Beule, D. M. Owen, H. B. M. C. B. T. J. R.-I. C. D.-J. M. R. K. P. B. D. S. E. I. M. M. J. L. P. A. M. A. N. and French, P. M. W. (2007). *Microscopic Research Techniques 70(5).*

QIOPTIQ (2010). *LINOS Machine Vision Lenses.* QIOPTIQ Photonics for Innovation, www.qioptiq.com.

Sellar, R. G. and Boreman, G. D. (2005). Classification of imaging spectrometers for remote sensing applications. *Optical Engineering 44(1).*

Thorlabs (2014). *WP25M-UB-AutoCAD.* Thorlabs, www.thorlabs.com.

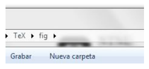
V. B. Voloshinov, K. B. Y. and Yukhnevich, T. V. (2012). Compensation for chromatic aberrations in acousto-optic systems used in spectral analysis of images. *Moscow Unviersity Physics Bulletin Vol.67.*

W. F. J. Vermaas, J. A. Timlin, H. D. T. J. M. B. S.-L. T. N.-S. W. H. D. K. M. and D. M. Haaland, P. N. (2008). *Acad. Sci. 105(10).*

web site EMCCD. What is emccd? http://www.emccd.com/what_is_emccd/. Andor®.

Curriculum Vitae

Name:	**Your Name**
Gender:	Female
Date of birth:	1. January 1995
Address:	Nordre gate 1, N–7005 Trondheim
Home address:	King’s road 1, 4590 Vladivostok, Senegal
Nationality:	English
Email (1):	your.name@stud.ntnu.no
Email (2):	yourname@gmail.com
Telephone:	+47 12345678



Your picture

Language Skills

Describe which languages you speak and/or write. Specify your skills in each language.

Education

- School 1
- School 2
- School 3

Computer Skills

- Program 1

- Program 2
- Program 3

Experience

- Job 1
- Job 2
- Job 3

Hobbies and Other Activities