UNIVERSITAT POLITÈCNICA DE CATALUNYA

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONA**TECH**

Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

STUDY: ANALYSIS OF PLANETARY SPACECRAFT IMAGES FOR ATMOSPHERE DYNAMICS STUDIES USING CROSSCORRELATION TOOLS AND PLANETARY IMAGE NAVIGATION SOFTWARE

REPORT

A thesis submitted by Martí Sierra Salvadó for the Bachelor's Degree in Aerospace Vehicle Engineering

June 10, 2019

Directed by:
Enrique García Melendo

Co-director:
Manel Soria Guerrero

Acknowledgements

"I would like to thank Enrique García and Manel Soria for their unconditional support throughout this project, and make a special mention to Roger Sala, without whom it would have been impossible to achieve the objectives set. Also dedicate the work done to my family and friends..."

Abstract

Atmospheric science is the study of the Earth's atmosphere, its processes and the interactions with other atmospheres. However, it has been extended to the field of planetary science and the study of atmospheres of the planets of the solar system. In the same way that the Earth's atmosphere, the planetary atmospheres are affected by other atmospheres and by varying degrees of energy, leading to the formation of dynamic weather systems, such as the anticyclonic storm on Jupiter, called the *Great Red Spot* [53]. In this way, the objective of the scientists is to know the evolution of this storms by performing simulations.

This study aims to process planetary images taken by interplanetary probes with the objective of obtaining the longitudes and latitudes of the pixels, so that the locations of the meteorological phenomena are known and can serve as validation of simulations [2]. For the cases studied, the images are in RAW format, i.e. as captured by the spacecraft camera, and its image file format, called VICAR, was developed by the NASA's JPL to transport images from space missions. The VICAR format predates the standard image formats, such as PNG, TIF, FITS, among others, so it has been necessary to develop a decoder capable of reading and converting the images to the aforementioned conventional formats.

Thus, starting from specialized software in this field, such as PLIA [46, 9], the objective has been to understand the algorithms that it implements and develop an own program capable of carrying them out, based on the the library SPICE developed by the NASA's JPL. In this sense, until obtain the longitudes and latitudes of the images, an initial program has been developed capable of reading the original images (RAW) taken by the interplanetary probes, optical distortions have been calibrated and corrected by specialized software (CISSCAL), and a solver able to implement the navigation algorithms involved in the PLIA code has been developed, which also estimates the navigation error for an automatic adjustment.

At the end of this study, the objectives have been achieved and the results have been compared with the specialized software, obtaining satisfactory results. Looking ahead, more space missions can be tested, more specifically, the next objective is to focus on the Juno probe, which is presented as a challenge due to its operationality.

Contents

I General introduction 13

1 Overview 14

1.1 Aim 14

1.2 Scope 15

1.3 Requirements 15

1.4 Justification 15

1.5 Collaboration 16

II Planetary Images Navigation 17

2 Introduction to the spacecrafts 18

2.1 Introduction to Cassini-Huygens 18

2.1.1 Mission Overview 18

2.1.2 Cassini Orbiter Instruments 19

2.2 Introduction to Voyager 21

2.2.1 Mission Overview 21

2.2.2 Voyager Instruments 22

2.3 Imaging Science Subsystem 24

3 ISS Image Reading 25

3.1 PDS ISS Data Archive 25

3.2 ISS Vicar Image Format 26

3.2.1 Overview 26

3.2.2 Labels 27
3.2.3 Image area 35
3.3 ISS Image Reading Software 35
3.3.1 Planetary Virtual Observatory and Laboratory (PVOL) 36
3.3.2 *Vicarread.m* 41

4 ISS Image Calibration: CISSCAL 44
4.1 Introduction 44
4.2 Setting Up the Environment 44
4.3 Starting CISSCAL 45
4.4 Default Options File 47
4.5 CISSCAL User Manual 48

5 Image Navigation 50
5.1 Planetary Laboratory for Image Analysis (PLIA) 50
5.1.1 Setting Up the Environment 50
5.1.2 PLIA User Manual 51
5.2 Developed Software 53
5.2.1 MATLAB Prototype 54
5.2.2 C Solver 60

6 Results and Validation 83

7 Environmental impact 96

8 Conclusions 97

List of Figures

1.1 Frame of the simulation [2] 16

2.1 Cassini Spacecraft Diagram [38] 21
2.2 Voyager Spacecraft Diagram [21] 23

3.1 Basic structure of a VICAR file [6] 27
3.2 VICAR file organization types [6] 30
3.3 VICAR file image area [6] 35
3.4 "File" Menu [7] 37
3.5 File slection form [7] 37
3.6 Filter options form [7] 38
3.7 List of images of the Cassini probe loaded [7] 39
3.8 Image selection [7] 40
3.9 Route selection form for the dump [7] 40

4.1 CISSCAL GUI [3, 44] 47
4.2 Batch Mode dialog [3, 44] 49

5.1 Planetocentric (θ) and planetographic (ϕ) coordinates [58] 52
5.2 Navigated image of Jupiter through MATLAB 57
5.3 Image projection of Jupiter 59
5.4 Illumination angles [49] 73

6.1 Navigated image of Jupiter through C solver 83
6.2 Illuminated area from real image 84

6.3 Illuminated area from real image + Moon correction 85
6.4 Navigation error 85
6.5 Navigation image corrected (Limits method) 86
6.6 Navigation image corrected (Centroid method) 87
6.7 Navigation error corrected (Limits method) 87
6.8 Navigation error corrected (Centroid method) 88
6.9 Image projection (Limits method) 88
6.10 Image projection (Centroid method) 89
6.11 Image projection (PLIA) 89
6.12 Image projection (PLIA) 90
6.13 Image projection reducing illumination value (Limits method) 90
6.14 Image projection reducing illumination value (Centroid method) 91
6.15 Navigated image of entire Jupiter 92
6.16 Image projection (Limits method) 92
6.17 Image projection (Centroid method) 93
6.18 Image projection (PLIA) 93
6.19 Image projection (PLIA) 94
6.20 Image projection in RGB format 95

List of Algorithms

1 *Vicarread.m algorithm* . . . 41
2 *Vicarlabels.m algorithm* . . . 42
3 *Convert_images_all_folders.m algorithm* . . . 43
4 *Convert_images_one_folder.m algorithm* . . . 43
5 *Navega_cassini_pA.m algorithm* Part 1 . . . 54
6 *Navega_cassini_pA.m algorithm* Part 2 . . . 55
7 *initSPICEd.m algorithm* . . . 57
8 *Navega_cassini_pB.m algorithm* Part 1 . . . 58
9 *Navega_cassini_pB.m algorithm* Part 2 . . . 59
10 *Navega_cassini.c algorithm* Part 1 . . . 63
11 *Navega_cassini.c algorithm* Part 2 . . . 64
12 *Navega_cassini.c algorithm* Part 3 . . . 65
13 *set_home_kernels algorithm* . . . 65
14 *only_download algorithm* . . . 66
15 *home_images algorithm* . . . 67
16 *read_lbl algorithm* . . . 67
17 *read_param algorithm* . . . 68
18 *furnsh_d_all algorithm* . . . 69
19 *furnsh_d algorithm* Part 1 . . . 69
20 *furnsh_d algorithm* Part 2 . . . 70
21 *furnsh_d algorithm* Part 3 . . . 71
22 *furnsh_t algorithm* . . . 71
23 *read_utctime algorithm* . . . 72
24 *generappm algorithm* . . . 72
25 *generalonlat algorithm* . . . 74
26 *navegatots.py algorithm* . . . 75
27 *worksplit algorithm* . . . 75
28 *Navega_cassini_pB_C.m algorithm* Part 1 . . . 76
29 *Navega_cassini_pB_C.m algorithm* Part 2 . . . 77
30 *Navega_correction.m algorithm* Part 1 . . . 78
31 *Navega_correction.m algorithm* Part 2 . . . 79

32 ***Navega_ correction.m algorithm*** Part 3 80
33 ***Moon_ correction.m algorithm*** Part 1 80
34 ***Moon_ correction.m algorithm*** Part 2 81
35 ***Limits_ correction.m algorithm*** 81

Acronyms

ASCII American Standard Code for Information Interchange.

ASI Italian Space Agency.

CAPS Cassini Plasma Spectrometer.

CCD Charged-Coupled Device.

CDA Cosmic Dust Analyzer.

CICLOPS Cassini Imaging Central Laboratory for Operations.

CIRS Composite Infrared Spectrometer.

CISSCAL Cassini Imaging Science Subsystem Calibration.

CNMC National Commission of Markets and Competition.

CRS Cosmic Ray Subsystem.

EOL End of File Label.

ESA European Space Agency.

FOV Field of View.

GCP Grupo de Ciencias Planetarias.

IDL Interactive Data Language.

INMS Ion and Neutral Mass Spectrometer.

IRIS Infrared Interferometer Spectrometer and Radiometer.

ISS Imaging Science Subsystem.

JPL Jet Propulsion Laboratory.

LECP Low-Energy Charged Particles.

LEMPA Low-Energy Magnetospheric Particle Analyser.

LEPT Low-Energy Particle Telescope.

MAG Magnetometer.

MIMI Magnetospheric Imaging Instrument.

MPI Message Passing Interface.

NAC Narrow-Angle Camera.

NASA National Aeronautics and Space Administration.

PDS Planetary Data System.

PLIA Planetary Laboratory for Image Analysis.

PLS Plasma Science.

PPM Portable Pixmap.

PPS Photopolarimeter Subsystem.

PRA Planetary Radio Astronomy.

PVOL Planetary Virtual Observatory and Laboratory.

PWS Plasma Wave Subsystem.

RADAR Radio Detection and Ranging.

RGB Red, Green, Blue.

RPWS Radio and Plasma Wave Spectrometer.

RSP Remote Sensing Palette.

RSS Radio Science Subsystem.

SCLK Spacecraft Clock Count.

TCS Telecommunications Subsystem.

UPV/EHU University of the Basque Country.

UVIS Ultraviolet Imaging Spectrograph.

UVS Ultraviolet Spectrometer.

VIMS Visual and Infrared Mapping Spectrometer.

WAC Wide-Angle Camera.

Part I

General introduction

Chapter 1

Overview

In planetary science, it is important to dispose of images of planets, processed so that the latitude and longitude of each pixel are known. Notably, for the study of the atmosphere of giant planets, such as Jupiter and Saturn, it is wanted to know the evolution of storms from images taken by interplanetary probes, like Cassini and Voyager, and also from terrestrial or space telescopes like the Hubble, and be able to compare them with simulations [2].

The process from the RAW image obtained by the spacecraft cameras to the images projected to a longitude/latitude plane involves several complex computations. Research groups specialized in planetary science have developed tools to do so. Among them, the Planetary Laboratory for Image Analysis (PLIA), developed at the *Grupo de Ciencias Planetarias (GCP)* in the University of the Basque Country (UPV/EHU) [46, 9].

The main goal of this TFG has been to understand the basic algorithms involved in a code such as PLIA and to implement them in MATLAB, using the library SPICE developed by the Jet Propulsion Laboratory (JPL).

The results obtained with the software developed have been compared with those obtained using PLIA.

1.1 Aim

In this way, the aim of this study is to process and navigate images of giant planets taken during exploration missions of the outer solar system, carried out by the main interplanetary spacecrafts, such as the Cassini-Huygens and the Voyager programs, with the purpose of carrying out a subsequent image analysis using planetary image navigation software.

1.2 Scope

Throughout this project, it is intended to understand the main algorithms implemented by PLIA and to program them in MATLAB, using the NASA's JPL SPICE library, an information system developed to assist NASA scientists in planning and interpreting scientific observations from space-borne instruments, and to assist NASA engineers involved in modeling, planning and executing activities needed to conduct planetary exploration missions.

Moreover, some functions of specialized software (PVOL, CISSCAL) for the selection, calibration and navigation of the images have also been implemented as MATLAB codes.

The design and implementation of graphic user interfaces for the algorithms implemented has been considered out of the scope of the TFG.

1.3 Requirements

Beyond this project, the added purpose of this study is that the developed solvers can be used by professionals working in the analysis of planetary images. Therefore, the source code must be clear and understandable, it must use good programming techniques and must be accompanied by the relevant documentation.

In order to create a proper solver, first prototype programs should be developed through MATLAB and final solvers should be written in C and PYTHON to ensure that the results are computed and delivered as fast as possible.

1.4 Justification

As already mentioned, in the realization of simulations it is very important to have real data with which to compare the results. As an example, in Figure 1.1 a numerical result of the simulation of [2] can be seen.

The analysis and navigation of real planetary images is of great importance when it comes to obtaining relevant data and real images of storms on planets, such as the red spot of Jupiter, so that the simulations performed on the behavior of these storms can be contrasted with the reality.

Furthermore, the existing specialized software for the processing and navigation of interplanetary images are written in programming languages not frequently used. Thus, this study intends to carry out an alternative software that allows to easily read, convert and display this type of files, for a better use and later analysis, and to program and build im-

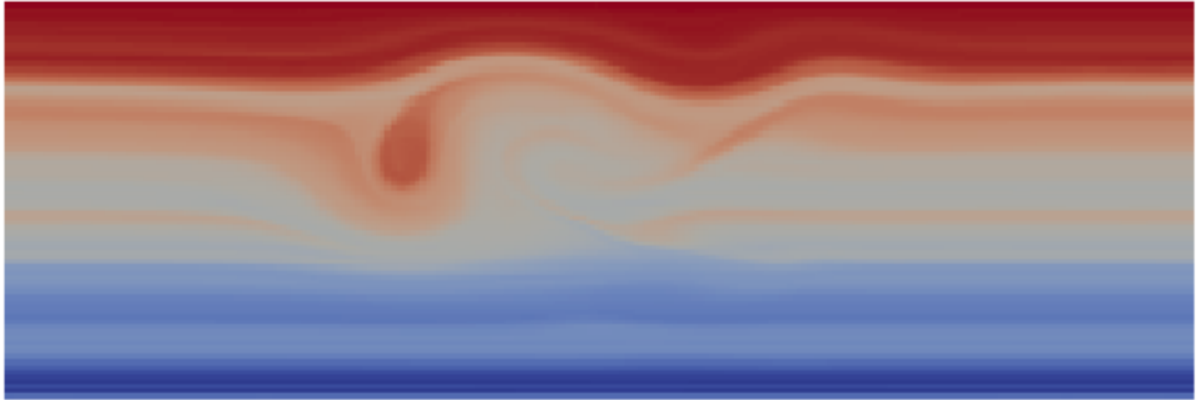

Figure 1.1: Frame of the simulation [2]

age processing and navigation solvers with languages understandable by anyone, through the use of more optimal, sophisticated and easier-to-use software.

1.5 Collaboration

This program is based on a NASA's information system called SPICE, which is capable of interpreting science information from space-borne instruments. Although the SPICE system is an important part in the development of this study, this document does not provide a detailed explanation of its operation, that can be found in the TFG of Roger Sala Marco.

Part II

Planetary Images Navigation

Chapter 2

Introduction to the spacecrafts

In this chapter, a brief introduction to the main spacecrafts launched on missions to the outer solar system is made. As said before, the spacecrafts on which this study is focused are the Cassini-Huygens and the Voyager programs.

Thus, for each spacecraft, an overview of the mission carried out is made, as well as a description of the its science sensors, focusing on the imaging systems.

2.1 Introduction to Cassini-Huygens

In this section, an overview of the mission carried out by the Cassini probe is made, as well as a description of its science sensors.

2.1.1 Mission Overview

Cassini-Huygens was a collaboration between the National Aeronautics and Space Administration (NASA), the European Space Agency (ESA) and the Italian Space Agency (ASI). It was an unmanned space mission whose objective was to study the planet Saturn and its systems, including its rings and natural satellites. The spacecraft consisted of two main elements: the Cassini probe and the Huygens descent module [54, 39].

The launch took place on October 15, 1997, with a Titan IVB/Centaur rocket of two stages, and after a seven-year voyage that included four gravity-assist maneuvers, one of them in Jupiter (December 2000), Cassini entered the orbit of Saturn on July 1, 2004. It then began a four-year mission that included more than 70 orbits around the ringed planet and its moons. Cassini completed its initial four-year mission in June 2008 [40].

After the primary mission, NASA announced a two-year extension, through September

2010, named the Cassini Equinox Mission, coinciding with the Saturn's equinox, which occurred in August 2009, when the sun shone directly on the equator and then began to illuminate the northern hemisphere and the rings' northern face. Another extension was announced on September 28, 2010, through the Saturnian summer solstice in May 2017, named the Cassini Solstice Mission[54, 40] .

For the final destination of the Cassini probe, it was decided to send it to an orbit of very high eccentricity in which, after performing 20 orbits, made its final approach to the giant planet, burning in its atmosphere on September 15, 2017. This last phase was named by NASA as Grand Finale [54, 40, 39].

2.1.2 Cassini Orbiter Instruments

Optical Remote Sensing: These instruments studied Saturn and its rings and moons in the electromagnetic spectrum [37].

- **CIRS – Composite Infrared Spectrometer:** Captured infrared light and split the light into its component wavelengths (or colours). Analysing an object's light, it can be determined its temperature, but also its composition [26].
- **ISS – Imaging Science Subsystem:** Consisting of a wide-angle and a narrow-angle digital camera, they were sensitive to visible wavelengths, and each of them had several filters to select the wavelengths to be sampled in each image [28].
- **UVIS – Ultraviolet Imaging Spectrograph:** Created pictures by observing ultraviolet light. Some gases become visible in ultraviolet lengths, so the spectrograph could determine the composition of those gases by splitting the light into its component wavelengths (similar to CIRS) [35].
- **VIMS – Visual and Infrared Mapping Spectrometer:** Collected both visible and infrared wavelengths. Scientists could learn about the composition of materials from which the light is reflected or emitted [36].

Fields, Particles and Waves: These instruments studied the dust, plasma and fields around Saturn [37].

- **CAPS – Cassini Plasma Spectrometer:** An in-situ instrument, detecting and analysing plasma in the vicinity of the spacecraft. The data measured was used to learn about the composition, density, flow, velocity and temperature of ions and electrons in Saturn's magnetosphere, helping to determine the sources of plasma [25].

- **CDA – Cosmic Dust Analyzer:** Detected dust particles, determining their charge, speed, size and in which direction they were going. When those particles smash into the instrument's detectors, it can be determined their composition [27].
- **INMS – Ion and Neutral Mass Spectrometer:** Determined the chemical, elemental and isotopic composition of the gaseous and volatile components of the neutral particles and the low energy ions in the atmosphere and ionosphere of Titan, magnetosphere of Saturn, and the ring environment [29].
- **MAG – Magnetometer:** Recorded the direction and strength of magnetic fields around the spacecraft [30].
- **MIMI – Magnetospheric Imaging Instrument:** Had three sensors that worked together to detect energetic charged particles in the excited gas, or plasma, around Saturn, and atoms without an electric charge (neutral atoms) [31].
- **RPWS – Radio and Plasma Wave Spectrometer:** Detected radio and plasma waves, as well as the plasma medium through which Cassini passed, using a suite of antennas and sensors [33].

Microwave Remote Sensing: These instruments used radio waves to map atmospheres, determine the mass of moons, collect data of ring particle size, and unveil the surface of Titan [37].

- **RADAR – Radio Detection and Ranging:** Sent radio waves at surfaces and, by recording slight differences in the signal's arrival time and wavelength back at the spacecraft, the instrument created pictures of the landscapes upon which it reflected [32].
- **RSS – Radio Science Subsystem:** Sent radio signals through, near or even bouncing off of objects in the Saturn system, helping scientists learn about the objects with which the radio waves interact [34].

SPICE – Navigation and Pointing information: Information system to assist NASA scientists in planning and interpreting scientific observations from space-borne instruments, and to assist NASA engineers involved in modelling, planning and executing activities needed to conduct planetary exploration missions [51].

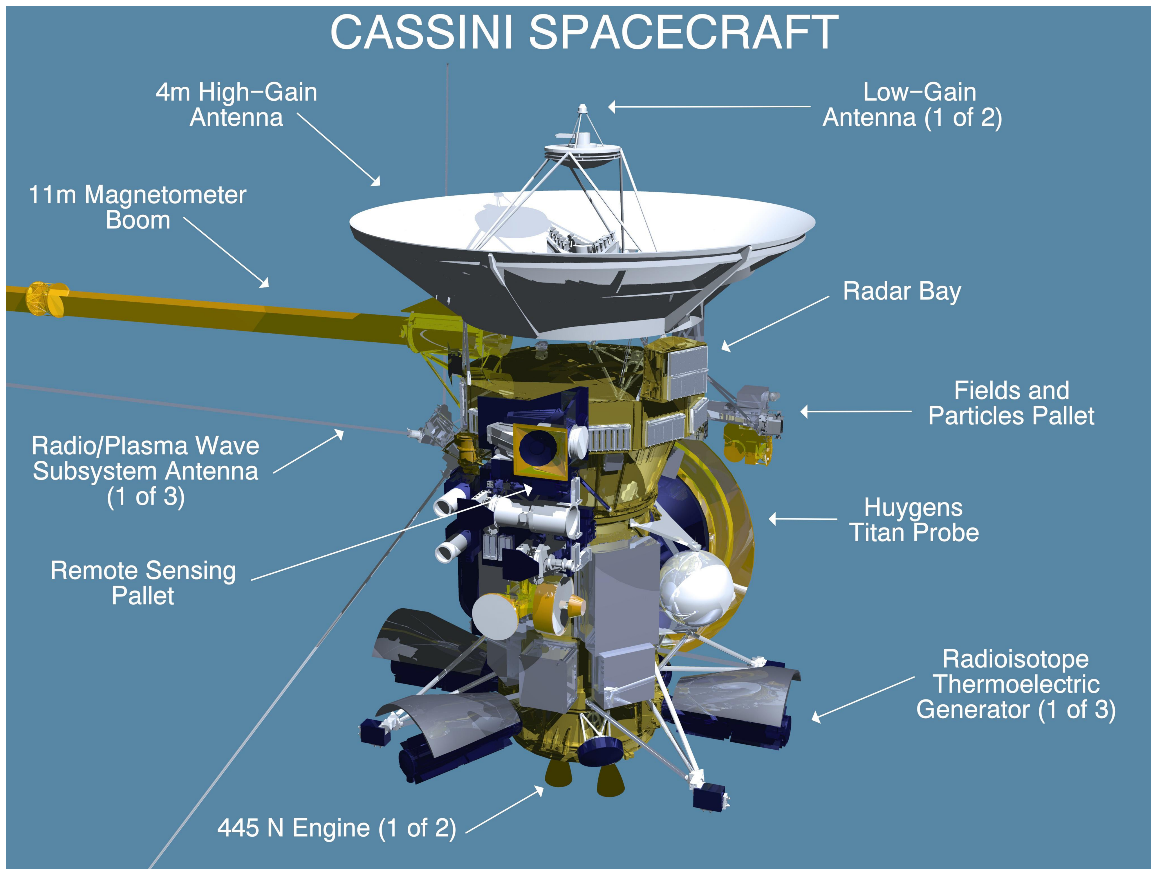

Figure 2.1: Cassini Spacecraft Diagram [38]

2.2 Introduction to Voyager

In this section, an overview of the mission carried out by the Voyager program is made, as well as a description of its science sensors.

2.2.1 Mission Overview

The Voyager program is an American scientific program that employs two robotic probes, Voyager 1 and Voyager 2, to study the outer Solar System. The twin probes were launched by NASA in separate months in the summer of 1977 to take advantage of a favourable alignment of Jupiter, Saturn, Uranus and Neptune [57].

As originally designed, the Voyagers were to conduct closeup studies of Jupiter and Saturn, the rings of Saturn, and the larger moons of the two planets. To accomplish their two-planet mission, the spacecraft were built to last five years. But as the mission went on,

and with the successful achievement of all its objectives, the additional flybys of the two outermost giant planets, Uranus and Neptune, proved possible, so their two-planet mission became four and their five-year lifetimes stretched to 12 and is now near thirty-seven years [11].

2.2.2 Voyager Instruments

The prime mission science payload consisted of 10 instruments (11 investigations including radio science) [22]. Target body or remote sensing instruments included:

- **ISS – Imaging Science Subsystem:** Utilized a two-camera system (narrow-angle/wide-angle) to provide imagery of Jupiter, Saturn and other objects along the trajectory [13, 57].
- **PPS – Photopolarimeter Subsystem:** Utilized a 0.2 m telescope fitted with filters and polarization analysers, covering eight wavelengths in the region between 235 and 750 nm, to gather information on surface texture and composition of Jupiter, Saturn, Uranus and Neptune and information on atmospheric scattering properties and density for these planets [17, 57].
- **IRIS – Infrared Interferometer Spectrometer and Radiometer:** Acting as three separate instruments, it can determine the temperature of a body or substance, the composition of an atmosphere or a surface and the amount of sunlight reflected by a body [14, 57].
- **UVS – Ultraviolet Spectrometer:** Sensitive to ultraviolet light (wavelength range of 40 to 180 nm), it was designed to measure the scattering properties of the lower planetary atmospheres and to look for certain elements and compounds whose emission colour in the ultraviolet light spectrum is known [20, 57].

Fields, waves and particles sensors included:

- **PLS – Plasma Science:** Measured the low energy ions and electrons of the plasma, determining its flow speed and direction, its density and its temperature [19, 57].
- **LECP – Low-Energy Charged Particles:** Utilized two subsystems, the Low-Energy Particle Telescope (LEPT) and the Low-Energy Magnetospheric Particle Analyser (LEMPA), to determine the concentration of particles (particles of higher energy than PLS) in the solar wind and measure their velocity and direction [15, 57].
- **CRS – Cosmic Ray Subsystem:** Looking only for very energetic particles in plasma, it provides information on the energy content, origin, accelerations process, life history, and dynamics of cosmic rays in the galaxy [12, 57].

- **MAG – Magnetometer:** Designed to measure and represent the planetary magnetic fields of Jupiter, Saturn, Uranus and Neptune, investigate the interactions between the solar wind and the magnetospheres of these planets, as well as the interactions of the satellites of these planets with their magnetosphere/solar wind environments [16, 57].
- **PWS – Plasma Wave Subsystem:** Provided measurements of the electron-density profiles at Jupiter and Saturn, as well as information on local wave-particle interaction, covering a frequency range of 10 Hz to 56 kHz [18, 57].
- **PRA – Planetary Radio Astronomy:** Utilized a sweep-frequency radio receiver to study the radio-emission signals from Jupiter and Saturn, covering two frequency bands, from 20.4 to 1300 kHz and from 2.3 to 40.5 MHz [18, 57].

The **Radio Science Subsystem (RSS)** investigation utilized the on-board and ground elements of the Telecommunications Subsystem (TCS) to determine the physical properties of planets and satellites (masses, gravity fields, densities. . .) and the amount and size distribution of material in the Saturn rings and the ring dimensions [57].

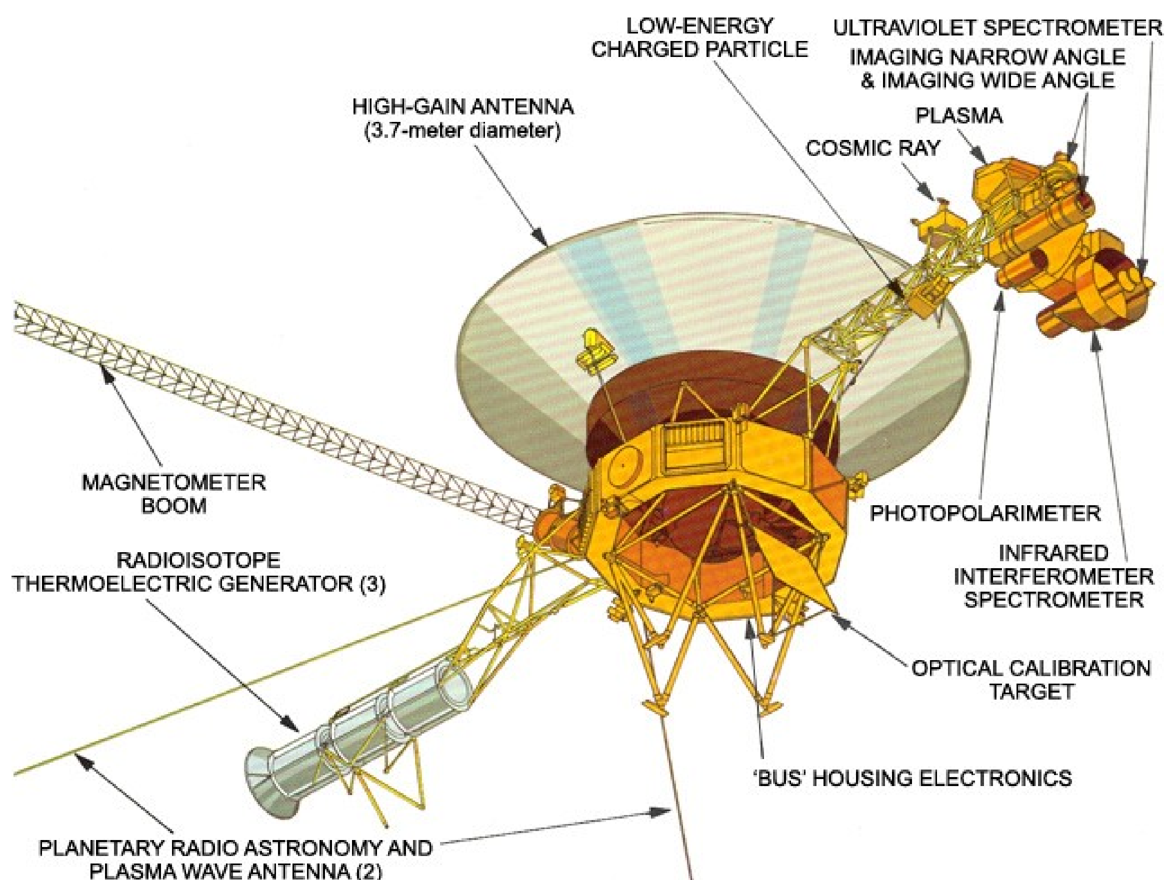

Figure 2.2: Voyager Spacecraft Diagram [21]

Unlike Cassini, when Voyager program began its missions, there was no SPICE system yet. Therefore, the existing SPICE files referring to the Voyager program missions were made later, once this system was implemented. This has resulted in a series of problems that have prevented the navigation of the images of said planetary program.

2.3 Imaging Science Subsystem

In this chapter, an overview of the Imaging Science Subsystem is presented. Since both Cassini and Voyager present a similar imaging tool, and finally the images of the Voyager have not been able to be navigated, only the Cassini ISS is explained.

The Cassini ISS consists of two fixed focal length telescopes, a Narrow-Angle Camera (NAC) and a Wide-Angle Camera (WAC). The NAC is 95 cm long and 40 cm x 33 cm wide, and has a focal length of 2002.70 +/- 0.07 mm in the clear filter, while the WAC is 55 cm long and 35 cm x 33 cm wide, and has a focal length of 200.77 +/- 0.02 mm in the clear filter [41, 42, 28, 3].

The two cameras together have a mass of 57.83 kg, and sit on the Remote Sensing Palette (RSP), fixed to the body of the Cassini Orbiter, between the Visual and Infrared Mapping Spectrometer (VIMS) and the Composite Infrared Spectrometer (CIRS), and above the Ultraviolet Imaging Spectrograph (UVIS) [3].

The NAC is an f/10.5 reflecting telescope with an image scale of 6 µrad/pixel, a 0.35 deg x 0.35 deg Field of View (FOV), and a spectral range from 200 nm - 1100 nm. Its filter wheel subassembly carries 24 spectral filters: 12 filters on each of two wheels [41, 28, 3].

The optical train of the WAC is an f/3.5 refractor with a 60 microrad/pixel image scale, a 3.5 deg x 3.5 deg FOV, and a spectral range from 380 nm - 1050 nm. Its filter wheel subassembly carries 18 spectral filters: 9 filters on each of two wheels [42, 28, 3].

Due to its long focal length, which makes it particularly susceptible to temperature effects, the NAC is thermally isolated from the RSP in order to minimize the effects of RSP thermal transients on the NAC image quality. The WAC has less stringent image quality requirements, and so its temperature is maintained by the RSP to which it is attached [41, 42, 3].

Chapter 3

ISS Image Reading

In this chapter, the different processes that can be used for reading and displaying the VICAR files, as well as the particularities of the latter, are explained.

So, this chapter is divided into 3 sections:

- **PDS ISS Data Archive:** Summary of how the ISS Data is organized.
- **ISS VICAR Image Format:** Description of the ISS VICAR files.
- **Reading Software:** Presentation of the Planetary Virtual Observatory and Laboratory (PVOL) and the developed MATLAB code, *Vicarread.m*.

3.1 PDS ISS Data Archive

The Planetary Data System (PDS) is a long-term archive of digital data products returned from NASA's planetary missions and research programs. It is a federation of "nodes" supporting research into specific disciplines, and the "Imaging Node" of the PDS is the Cartography and Imaging Sciences Discipline Node, which stores NASA's primary digital image collections from past, present and future planetary missions [45, 43].

For each archive volume containing the images there is a VOLUME_ID. The following naming conventions are followed [23]:

VOLUME_ID = <spacecraft><instrument>_<number>

Where:

spacecraft = 2-character spacecraft identifier, e.g. CO ("Cassini Orbiter").
instrument = instrument identifier, e.g. ISS.

number = 4-digit value, where the first value is 1 for Jupiter, 2 for Saturn, 3 for cartographic maps, and 0 for calibration, and where the next 3 values is the sequential numbering of the volume starting with 001 [3].

Each volume of these contains a series of folders and files, the most important of which is the one called "DATA", which contains all data files, ordered by time or spacecraft clock count (SCLK) [23, 3].

The images contained in the "DATA" folder are of VICAR format and are named by the spacecraft clock count. The following naming conventions are followed [23, 3]:

Image file = <camera><SCLK time>_<version>.IMG

Where:

camera = 1-character instrument identifier (N=NAC, W=WAC).
SCLK time = 10-digit value of spacecraft clock at time of shutter close.
version = version number of the file.

So, for example, an image file named W1832898283_4.IMG would indicate the fourth version of a Wide Angle Camera image taken at SCLK time 1832898283. Each image file has its corresponding detached label file which follows the same naming convention as above except with ".LBL" as the filename extension. Example: W1832898283_4.LBL.

3.2 ISS Vicar Image Format

VICAR is a set of computer programs and procedures designed to facilitate the acquisition, processing and handling of digital image data. The VICAR image processing language was defined by the NASA's Jet Propulsion Laboratory (JPL) and implemented in 1966 to process image data produced by the planetary exploration program [10].

The following is an overview of the basic structure of VICAR files, as well as a description of their content. Any VICAR files written out must include all the system label items defined below.

3.2.1 Overview

The basic structure of a VICAR file is shown below.

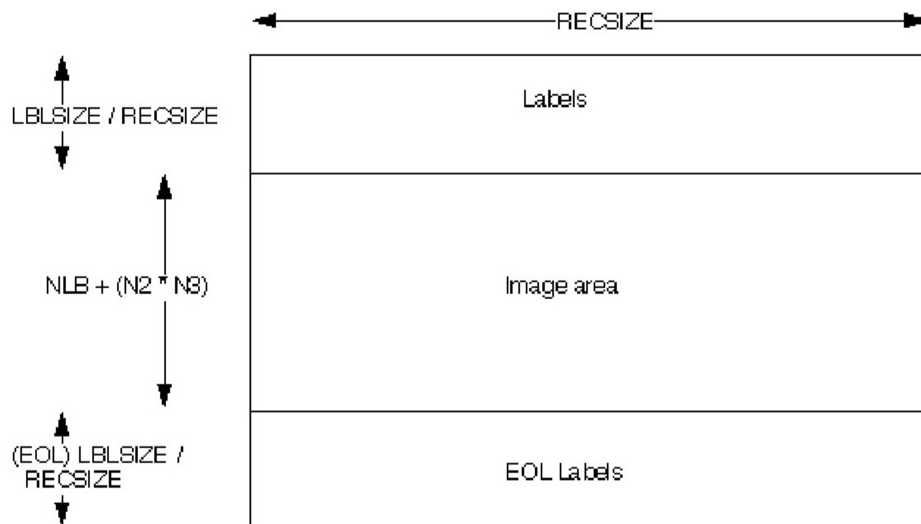

Figure 3.1: Basic structure of a VICAR file [6]

A VICAR file consists of two major parts: the labels, which describe what the file is, and the image area, which contains the actual image. The labels are potentially split into two parts, one at the beginning of the file, and one at the end. Normally, only the labels at the front of the file are present. However, if the End of File Label (EOL) keyword in the system label (described below) is equal to 1, then the EOL labels are present. This happens if the labels expand beyond the space allocated for them [6].

The VICAR file is treated as a series of fixed-length records, of size RECSIZE (see below). The image area always starts at a record boundary, so there may be unused space at the end of the label, before the actual image data starts [6].

3.2.2 Labels

The label consists of a sequence of "keyword=value" pairs that describe the image, and is made up entirely of ASCII characters. Each keyword-value pair is separated by spaces. The keyword is a text keyword (string), up to 32 characters in length, that identifies the label item, and the value is the information portion of the label item; may be of type string, integer, real, or double, and may be multi-valued. Spaces may appear on either side of the equals character (=), but are not normally present [6].

The first keyword is always LBLSIZE, which specifies the size of the label area in bytes. LBLSIZE is always a multiple of RECSIZE, even if the labels don't fill up the record. If the labels end before LBLSIZE is reached (the normal case), then a 0 byte terminates the label string. If the labels are exactly LBLSIZE bytes long, a null terminator is not necessarily

present. The size of the label string is determined by the occurrence of the first 0 byte, or LBLSIZE bytes, whichever is smaller [6].

As said before, if the system keyword EOL has the value 1, then EOL labels exist at the end of the image area (see above). The EOL labels, if present, start with another LBLSIZE keyword, which is treated exactly the same as the main LBLSIZE keyword. Note that the main LBLSIZE does not include the size of the EOL labels [6].

The label is divided into three logical parts: System labels, Property labels, and History labels, in that order [6]. These parts are described later in this section.

Label values

The label values may be of three types: integer, real, or string [6].

- Integer: A sequence of digits (0-9), with an optional sign (+/−). There must be no embedded blanks in the integer, including between the sign and the number [6].
- Real: A sequence of digits (0-9) including a decimal point (.), an optional sign (+/−), and an optional exponent (one of the letters EeDd followed by a base-10 exponent in integer format). The letter E is greatly preferred for indicating the exponent. There must be no embedded blanks in the real number. The number must contain either a decimal point or an exponent, or else it is considered an integer. The number of significant digits is variable, so the number may be read as either single or double precision [6].
- String: A string is a sequence of ASCII characters enclosed in single quotes ('). A single quote may be included in the string by doubling it (e.g. 'can''t') [6].

A keyword may have more than one value by enclosing the values in parentheses and separating the values with commas. The collection of values is treated like an array for that keyword. All values in a multivalued label item must be of the same type. Spaces may exist around the parentheses or the commas, but are not normally present [6].

Examples:

LBLSIZE=1024; FORMAT='BYTE'; LATITUDE=45.3;
COORDS=(5.7,-3.2E+2); COMMENTS=('Wow, this is a comment!', 'This can''t be real');
EXTRA_SPACES = (1, 2,3, 4 , -5); ...

System labels

System labels describe the format of the image and how to access it. They are always the first labels in the file. The system labels extend from the beginning of the file until the

first PROPERTY or TASK keyword, or until the end of the label (if there are no property or history labels) [6].

Some system label items are mandatory, while others are optional. The mandatory ones are mentioned below. However, when writing a new file, all system label items should normally be included [6].

The currently defined system label items are listed below. They generally appear in the order listed, but the items order is no guaranteed, except that LBLSIZE must always be first. So, any program that reads the label must be able to handle any order of label items [6].

- LBLSIZE, integer, mandatory: The size of the label storage area, in bytes. It is always the first thing in the file. This label appears twice if EOL labels are present; once at the beginning of the file and once at the beginning of the EOL labels. The size specified applies only to the section (main or EOL) that the LBLSIZE item is in [6].
- FORMAT, string, mandatory: The data type of the pixels in the image. Valid values are [6]:
    - BYTE: one byte unsigned integer, range 0 to 255.
    - HALF: two byte signed integer, range −32768 to 32767.
    - FULL: four byte signed integer, range −2147483648 to 2147483647.
    - REAL: single-precision floating point number.
    - DOUB: double-precision floating point number.
    - COMP: complex number, composed of two REALs in the order (real, imaginary).

  The following values are obsolete, but may appear in some older images:

    - WORD: same as HALF.
    - LONG: same as FULL.
    - COMPLEX: same as COMP.
- TYPE, string: The kind of file this is. TYPE defaults to IMAGE (standard VICAR image file) [6].
- BUFSIZ, integer, mandatory: This label item is obsolete, but it still must be present for historical reasons. In new files, it is set equal to RECSIZE.
- DIM, integer: The number of dimensions in the file, which is always equal to 3. Some older images may have a DIM of 2, in which case some labels are present [6].

- EOL, integer: A flag indicating the existence of EOL labels. If EOL=1, the labels are present. If EOL=0 (or is absent), no EOL labels are present, and the entire label string is at the front of the file [6].
- RECSIZE, integer, mandatory: The size in bytes of each record in the VICAR file. It may be calculated with the formula NBB+N1*pixel_size (see Figure 3.3) [6].
- ORG, string: The organization of the file. While N1 is always the fastest-varying dimension, and N3 is the slowest, the terms Samples, Lines, and Bands may be interpreted in different ways. ORG specifies which interpretation to use, and defaults to BSQ. The valid values are [6]:
  - BSQ: Band SeQuential. The file is a sequence of bands. Each band is made up of lines, which are in turn made up of samples. So, N1=Samples, N2=Lines, and N3=Bands. This is the most common case.
  - BIL: Band Interleaved by Line. The file is a sequence of lines. Each line is made up of bands, which are in turn made up of samples. So, N1=Samples, N2=Bands, and N3=Lines.
  - BIP: Band Interleaved by Pixel. The file is a sequence of lines. Each line is made up of samples, which are in turn made up of bands. So, N1=Bands, N2=Samples, and N3=Lines.

  The three organizations are depicted graphically below.

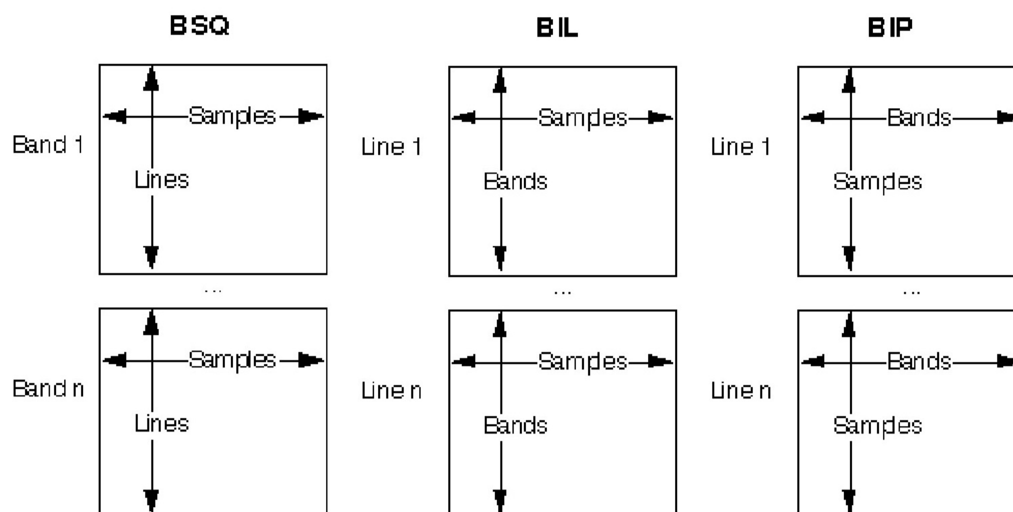

Figure 3.2: VICAR file organization types [6]

- NL, integer, mandatory: The number of lines in the image (same as N2 for BSQ or N3 for BIL and BIP) [6].

- NS, integer, mandatory: The number of samples in the image (same as N1 for BSQ and BIL or N2 for BIP) [6].
- NB, integer, mandatory: The number of bands in the image (same as N3 for BSQ, N2 for BIL, or N1 for BIP) [6].
- N1, integer: The size (in pixels) of the first (fastest-varying) dimension. If not present, it defaults to NS or NB, as appropriate [6].
- N2, integer: The size of the second dimension. If not present, it defaults to NL, NS, or NB, as appropriate [6].
- N3, integer: The size of the third (slowest-varying) dimension. If not present, it defaults to NL or NB, as appropriate [6].
- N4, integer: This item was to have been used for four-dimensional files, but this has not yet been implemented. It defaults to 0 [6].
- NBB, integer: The number of bytes of binary prefix before each record. Each and every record consists of the pixels of the fastest-varying dimension, optionally preceded by a binary prefix. The size (in bytes, not pixels) of this binary prefix is given by NBB, which defaults to 0 [6].
- NLB, integer: The number of lines (records) of binary header at the top of the file. The optional binary header occurs once in the file, between the main labels and the image data. The size of the binary header in bytes is given by NLB*RECSIZE, since NLB is a line count. NLB defaults to 0. Note that the binary header also includes space reserved for the binary prefix (NBB), since NBB goes into RECSIZE. The binary header and the binary prefix together make up the binary label (see Section 3.2.3) [6].
- HOST, string: The type of computer used to generate the image. It is used only for documentation. HOST defaults to VAX-VMS [6].
- INTFMT, string: The format used to represent integer pixels (BYTE, HALF, and FULL) in the file. If INTFMT is not present, it defaults to LOW. Note that INTFMT should be present even if the pixels are a floating-point type. The valid values are [6]:
  - HIGH: High byte first, big endian.
  - LOW: Low byte first, little endian.

  For INTFMT=HIGH, the high-order byte is first for HALF and FULL, while for INTFMT=LOW the low-order byte is first and all the bytes are swapped (i.e. 4321 instead of 1234). The representations for BYTE are identical in HIGH and LOW.

- REALFMT, string: The format used to represent floating-point pixels (REAL, DOUB, and COMP) in the file. If REALFMT is not present, it defaults to VAX. Note that REALFMT should be present even if the pixels are an integral type. The valid values are [6]:
  - IEEE: IEEE 754 format, with the high-order bytes (containing the exponent) first.
  - RIEEE: Reverse IEEE format. Just like IEEE, except the bytes are reversed, with the exponent last.
  - VAX: VAX format. Single precision is in VAX F format, double precision is in VAX D format.
- BHOST, string: The type of computer used to generate the binary label. It can take the same values with the same meanings as HOST. The reason BHOST is separate is that the data in the binary label may be in a different host representation than the pixels [6].
- BINTFMT, string: The format used to represent integers in the binary label. It can take the same values with the same meanings as INTFMT. The reason BINTFMT is separate is that the data in the binary label may be in a different host representation than the pixels [6].
- BREALFMT, string: The format used to represent floating-point data in the binary label. It can take the same values with the same meanings as REALFMT. The reason BREALFMT is separate is that the data in the binary label may be in a different host representation than the pixels [6].
- BLTYPE, string: The type of the binary label. This is not a data type, but is a string identifying the kind of binary label in the file. It is used for documentation [6].

Example:

The system label for a typical file is shown below. Although carriage returns have been inserted for clarity, none actually exist in the file.

LBLSIZE=1024 FORMAT='BYTE' TYPE='IMAGE' BUFSIZ=20480 DIM=3 EOL=0
RECSIZE=512 ORG='BSQ' NL=512 NS=512 NB=1 N1=512 N2=512 N3=1 N4=0
NBB=0 NLB=0 HOST='VAX-VMS' INTFMT='LOW' REALFMT='VAX'
BHOST='VAX-VMS' BINTFMT='LOW' BREALFMT='VAX' BLTYPE=''

Property Labels

Property labels describe properties of the image in the image domain. They contain other current information about the file, such as the map projection used, a lookup table, or latitude/longitude information for the image [6].

Property labels are divided into named sets called properties. Each property is made up of zero or more label items that contain the actual property information. The name space for each property is independent, so the same label item keyword may be used in more than one property. Only one property of a given name may exist [6].

Property labels are located between the system and the history labels. They start with the first occurrence of the keyword PROPERTY, and end with the first occurrence of the keyword TASK or the end of the labels (if there are no history labels). It is quite possible that no property labels exist in a file, in which case there would be no PROPERTY keywords [6].

Each property begins with a PROPERTY keyword, which has a string value. This value is the name of the property set. The PROPERTY keyword is followed by the label items that make up the property. The set continues until the next PROPERTY keyword, or the end of the property labels [6].

Label items within a property must not use the keywords DAT_TIM, LBLSIZE, PROPERTY, TASK, or USER. A simple display program could ignore the property labels completely [6].

Example:

Below is an example of what a property label with two properties might look like. Also, carriage returns have been inserted for clarity, and do not exist in the label.

PROPERTY='MAP' PROJECTION='mercator' LAT=34.2 LON=177.221
PROPERTY='LUT' RED=(1,2,3,4,5,6,7,8) GREEN=(8,7,6,5,4,3,2,1)
BLUE=(1,1,1,3,5,7,8,8)

History Labels

History labels describe the processing history of the image. Each processing step has an entry (called a TASK) in the history label. Each task can optionally have label items further describing the task (such as parameters to the program). They should contain only historical information; however, they often contain current state information that should be in a property label, since property labels are new and not yet well utilized [6].

History labels are divided into sets called tasks. Each task is made up of three mandatory label items, and zero or more label items that contain additional history information. The

name space for each task is independent, so the same label item keyword may be used in more than one task. Each task has a task name associated with it, which is the name of the program that created that part of the history label. However, the task names are not unique. Several tasks may have the same name. Each occurrence of the task name is called an instance, so the task name and the instance combine to uniquely identify the task set [6].

History labels are located after the system and the property labels. They start with the first occurrence of the keyword TASK, and end with the end of the labels. It is possible that no history labels exist in a file, in which case there would be no TASK keywords [6].

Each history task begins with a TASK keyword, which has a string value. This value is the name of the task. The instance is derived by counting the number of previous TASK keywords with the same task name; it is not stored explicitly in the label. The TASK keyword is followed by a USER and a DAT_TIM keyword [6]:

- USER: String specifying the username of the account that ran the program.
- DAT_TIM: String specifying the date the program was run, in the format Www Mmm dd hh:mm:ss yyyy, where Www is the three-letter day of the week, Mmm is the three-letter month, and the rest are digits.

Following the USER and DAT_TIM keywords are the optional label items with further history information. The task set continues until the next TASK keyword, or until the end of the labels.

Label items within a task must not use the keywords DAT_TIM, LBLSIZE, PROPERTY, TASK, or USER. A simple display program could ignore the history labels completely [6].

Example:

Below is an example of a typical history label with several tasks in it. Although carriage returns have been inserted for clarity, none actually exist in the file.

TASK='GEN' USER='RGD059' DAT_TIM='Thu Sep 24 17:31:50 1992' IVAL=0.0 SINC=1.0 LINC=1.0 BINC=1.0 MODULO=0.0 TASK='COPY' USER='RGD059' DAT_TIM='Thu Sep 24 17:31:54 1992' TASK='LABEL' USER='RGD059' DAT_TIM='Thu Sep 24 17:32:54 1992' TASK='F2' USER='RGD059' DAT_TIM='Thu Sep 24 17:33:07 1992' FUNCTION='in1+10' TASK='STRETCH' USER='RGD059' DAT_TIM='Thu Sep 24 17:33:55 1992' PARMS='AUTO-STRETCH: 0 to 0 and 138 to 255'

3.2.3 Image area

Following the labels (or between the label parts if there are EOL labels) is the image area. The structure and content of the image area are described in this section [6].

The image area is made up of records RECSIZE in length. Each record contains one line of data (for BSQ), i.e. one set of N1 pixels, plus the binary prefix, if any. If NBB=0, the binary prefix does not exist. A set of N2 records comprises a band (for BSQ), and a set of N3 bands makes up the image. The image is optionally preceded by NLB records of binary header. If NLB=0, the binary header does not exist [6].

The structure of the image area is shown below.

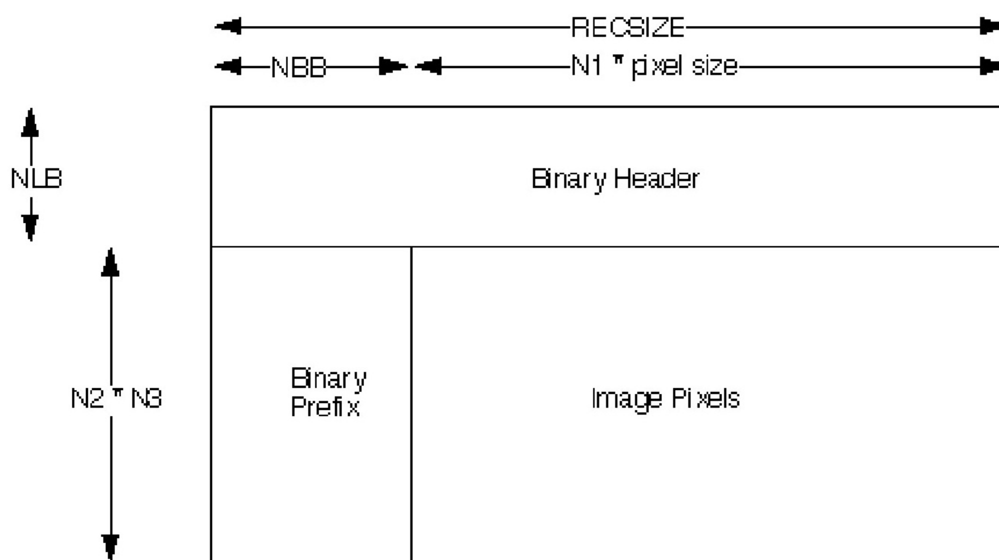

Figure 3.3: VICAR file image area [6]

Binary labels are the least well-defined part of the VICAR file format. Binary labels consist of two parts: binary headers, which occur once at the top of the file, and binary prefixes, which occur before every image record. For most purposes, especially for simple display programs, binary labels can be ignored. Most of the time, they are not even present [6].

3.3 ISS Image Reading Software

In this section, first the existing software to view and select Cassini images is presented, called Planetary Virtual Observatory and Laboratory (PVOL), and then the developed

MATLAB program is presented, called *Vicarread.m*, which allows reading and displaying the images.

3.3.1 Planetary Virtual Observatory and Laboratory (PVOL)

Introduction

PVOL stands for Planetary Virtual Observatory and Laboratory and is a searchable database of ground-based observations of solar system planets. It is an image management system, so images in a certain collection can be added/edited/deleted/searched/filtered [47, 48].

It is a very useful tool, since most of the software are focused on the navigation of images, as is the case of PLIA, explained in Section 5.1, and not so much in the management actions of these images, since it is not easy to manage collections with thousands of files, such as the one used in this project (see Section 3.1).

The PVOL system is based on two elements that can work independently: a WEB system (PVOLweb) and a Windows application (PVOL++). Throughout this study, only the PVOL++ application is used, since all the databases and images are already downloaded [1].

Thus, the PVOL++ browser facilitates the selection and storage of the desired images, serving at the same time as a viewer of said images, which are difficult to visualize through the programs included in a conventional computer, due to its format (VICAR).

Next, the requirements for the installation and use of the PVOL++ application and the necessary files for the selection and visualization of the images are specified, and the basic order of operations for the selection and storage process of the images is explained step by step.

Set Up the Environment

For the installation of PVOL++, the only requirement is that the computer on which PVOL++ is going to run has Windows as the operating system [1].

In addition, PVOL is a database developed at the *Grupo de Ciencias Planetarias* (GCP) in the University of the Basque country (UPV/EHU) [47], so some contact with this association is needed in order to obtain the PVOL++ browser.

Regarding this study, the tutor Enrique García Melendo participated in the development of said browser, so there have been no problems in obtaining it.

PVOL++ User Manual

Below is a brief description of the main actions to be performed in the PVOL++ browser in order to load the images in the system and view, select and store them in the specified destination folder.

- Load list of Cassini images (local database): The loading process of the Cassini database is already indicated in the software, as "default configuration". The actions to be carried out can be summarized in the following [7]:
  - Request the local database load: It is done by selecting "Open local database" in the "File" menu.

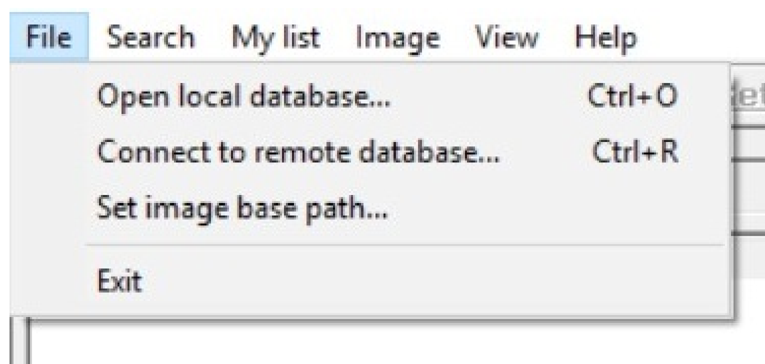

Figure 3.4: "File" Menu [7]

  - If the default configuration was not used, the program would ask the user to indicate the place where the configuration and database files are located.

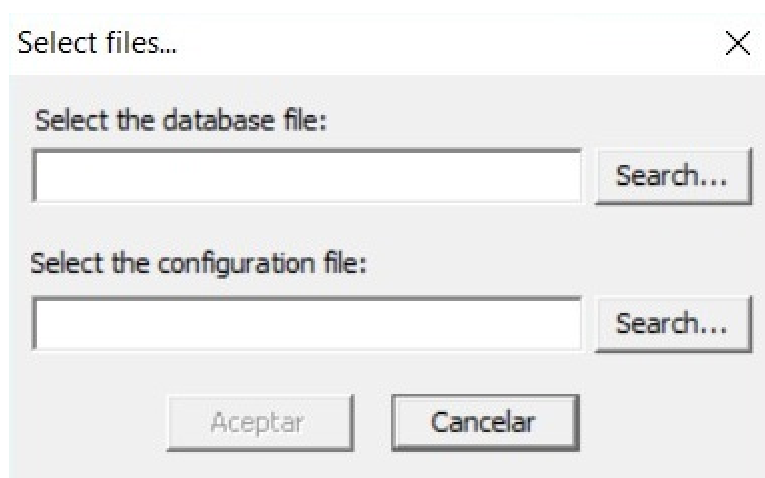

Figure 3.5: File slection form [7]

◦ Set the search parameters: It is done by filling out the form that is presented to the user.

Figure 3.6: Filter options form [7]

As it can be seen in Figure 3.6, the form presents the following filter options [7]:

– Target: Target body of the images, e.g. Jupiter.
– Volume: Volume of images to be displayed, e.g. COISS_1001.
– Date: The images can be filtered according to the specified date range.
– Camera: The images can be filtered according to the camera with which they were taken, e.g. NAC.

– Filters: The images taken usually present two different types of optical filters, so the images can be filtered according to the filters with which they were taken.

In the process of image filtering, it must be taken into account that not all the objectives of the images are present in all the volumes, so empty lists may appear [7]. Once the search parameters are set, the following result is obtained is:

Figure 3.7: List of images of the Cassini probe loaded [7]

At this point, if the user is working with an indicated database, a message may appear indicating that the image has not been found, instead of the image of the selected planet, as seen in Figure 3.7. It can be for several reasons [7]:

– Perhaps the corresponding volume of images (COISS_1001 in Figure 3.6) is not inserted in the equipment.

– Perhaps the base search path for images is not established. Select the "Set image base path" option in the "File" menu and enter the path to the corresponding directory.

- The selection of determined images is done using the buttons with the "+" and "−" symbols in the toolbar, or through the "Add selected item" and "Remove selected item" options in the "MyList" menú. The selected images are marked with a red dot in the list [7].

File Search My list Image View Help

Get

C:\TFG\cassini_ISS\COISS_1004\\\\\data\1355536275_1355692706\N1355543387_1.IMG

CASSINI_ID	FILE_SPE...	VOLUME_ID	Date	Time	FILTER_N...	FILTER_N...	CENTER_L...	CENTER_L...
2202	data/1355...	COISS_10...	2000-12-15	03:30:01	BL1	CL2	32.538539	62.735813
2203	data/1355...	COISS_10...	2000-12-15	03:30:36	IRP0	CB3	32.361647	63.328896
2204	data/1355...	COISS_10...	2000-12-15	03:30:55	CL1	CB3	32.79007	63.655681
2205	data/1355...	COISS_10...	2000-12-15	03:31:13	CL1	MT2	32.847736	63.054193
2206	data/1355...	COISS_10...	2000-12-15	03:31:30	CL1	CB2	32.844674	63.286426
2207	data/1355...	COISS_10...	2000-12-15	03:32:04	IRP0	MT3	32.251106	64.41599
2208	data/1355...	COISS_10...	2000-12-15	03:32:21	CL1	MT3	32.444695	63.986457
2209	data/1355...	COISS_10...	2000-12-15	03:33:24	UV1	CL2	32.532401	64.996278
2210	data/1355...	COISS_10...	2000-12-15	03:38:18	BL1	CL2	-28.313313	67.807923
2211	data/1355...	COISS_10...	2000-12-15	03:38:53	IRP0	CB3	-27.780458	68.224989
2212	data/1355...	COISS_10...	2000-12-15	03:39:12	CL1	CB3	-27.845986	68.613594
2213	data/1355...	COISS_10...	2000-12-15	03:39:30	CL1	MT2	-27.219281	68.457253
2214	data/1355...	COISS_10...	2000-12-15	03:39:47	CL1	CB2	-27.428872	68.65027
2215	data/1355...	COISS_10...	2000-12-15	03:40:21	IRP0	MT3	-27.419952	69.33232
2216	data/1355...	COISS_10...	2000-12-15	03:40:38	CL1	MT3	-27.458895	69.115549
2217	data/1355...	COISS_10...	2000-12-15	03:41:41	UV1	CL2	-27.137506	69.339256
2218	data/1355...	COISS_10...	2000-12-15	04:30:01	BL1	CL2	32.054799	98.918456
2219	data/1355...	COISS_10...	2000-12-15	04:30:34	CL1	CB1	32.268844	99.369426
2220	data/1355...	COISS_10...	2000-12-15	04:31:08	CL1	MT1	32.292819	99.207715
2221	data/1355...	COISS_10...	2000-12-15	04:31:41	CL1	CB2	32.538524	99.667108
2222	data/1355...	COISS_10...	2000-12-15	04:32:14	CL1	MT2	32.606169	100.14764
2223	data/1355...	COISS_10...	2000-12-15	04:32:47	CL1	CB3	32.470862	100.65489
2224	data/1355...	COISS_10...	2000-12-15	04:33:20	CL1	MT3	32.452946	100.90848
2225	data/1355...	COISS_10...	2000-12-15	04:34:00	UV1	CL2	32.528113	100.77491
2226	data/1355...	COISS_10...	2000-12-15	04:38:18	BL1	CL2	-27.690905	103.95287
2227	data/1355...	COISS_10...	2000-12-15	04:38:51	CL1	CB1	-27.062576	104.36735
2228	data/1355...	COISS_10...	2000-12-15	04:39:25	CL1	MT1	-27.057408	104.64263
2229	data/1355...	COISS_10...	2000-12-15	04:39:58	CL1	CB2	-27.088881	104.99513
2230	data/1355...	COISS_10...	2000-12-15	04:40:31	CL1	MT2	-26.817528	105.28458
2231	data/1355...	COISS_10...	2000-12-15	04:41:04	CL1	CB3	-27.278159	105.767
2232	data/1355...	COISS_10...	2000-12-15	04:41:37	CL1	MT3	-27.289063	105.86275
2233	data/1355...	COISS_10...	2000-12-15	04:42:17	UV1	CL2	-27.181077	106.86317
2234	data/1355...	COISS_10...	2000-12-15	05:30:01	BL1	CL2	32.386855	134.55244
2235	data/1355...	COISS_10...	2000-12-15	05:30:34	UV2	CL2	32.542909	134.81812
2236	data/1355...	COISS_10...	2000-12-15	05:30:52	CL1	CB2	32.322848	135.56164
2237	data/1355...	COISS_10...	2000-12-15	05:31:28	CL1	MT2	32.484071	135.90149
2238	data/1355...	COISS_10...	2000-12-15	05:32:01	CL1	MT3	32.062539	136.20136
2239	data/1355...	COISS_10...	2000-12-15	05:33:03	UV1	CL2	32.4767	136.71526
2240	data/1355...	COISS_10...	2000-12-15	05:38:18	BL1	CL2	-27.9841	140.28106

Figure 3.8: Image selection [7]

If the list of images is saved in a text file (with the option "Save to file" in the "MyList" menu), the points turn green [7].

If the user wants to dump all the selected images to a certain directory of the user's filesystem, it must be requested using the "Get" button in the toolbar or with the option "Get files!" from the "MyList" menu. A form appears requesting the path of the directory in which the user wants to save the files, as presented below [7]:

Destination directory...

Insert the path for the destination directory:

OK

Cancel

Figure 3.9: Route selection form for the dump [7]

Once the route is specified, the program copies all the image files to the destination directory, as well as the associated "Label" or navigation files. In addition, a file of

type '.XML' is created, which contains the labels associated with the selected image so that it can be opened using the PLIA software [7].

3.3.2 *Vicarread.m*

Alternatively to the PVOL++ browser, a MATLAB code has been developed in order to read and display any VICAR image, as well as to save the information stored in them. So, the pseudocode of the algorithm is as follows:

Vicarread.m algorithm

```
Input data: image name (path of the file may be needed);
vicar ← Structure with the labels of the image file. Call Vicarlabels.m function;
Read the image file using the fopen function;
Seek for the position of the first image pixel using the fseek function;
num_records ← Number of records = Number of rows of the image;
n ← Depends on the image format;
switch vicar.FORMAT do
    case 'BYTE'
        n ← 1 (one byte unsigned-integer);
    case 'HALF'
        n ← 2 (two byte signed-integer);
    case 'REAL'
        n ← 3 (single-precision floating point);
end switch
endianness ← Depends on the endianness of the image;
switch vicar.INTFMT do
    case 'HIGH'
        endianness ← 'b' (big-endian);
    case 'LOW'
        endianness ← 'l' (little-endian);
end switch
for i ← 1:num_records do
    First reads the binary prefix of the record (or row), if needed;
    Switch case n: reads the image record and saves pixels data using the fread function;
    pixels ← Array keeping the data of all the records;
end for
pixels ← Switch case n: converts pixels data into displayable data;
Close the image file using the fclose function;
return pixels, vicar;
```

As presented in Section 3.2, the VICAR files consist of two major parts: the labels and

the image area. So, in order to know where the image area starts and read it properly, is also important to read the labels that the file contains, from which relevant parameters can be obtained, such as the dimensions of the image, in what format the image is, among others.

This task is carried out by the function called *Vicarlabels.m*, which can be seen in the diagram above. So, the pseudocode of the algorithm is as follows:

Vicarlabels.m algorithm

Input data: image name (path of the file may be needed);
Read the image file using the *fopen* function;
Get the labels of the file as a single string using the *fgetl* function;
$l \leftarrow$ String containing the labels of the file;
Split each label using the *strsplit* function;
$s \leftarrow$ Cell array containing the labels;
metadata_ dict $\leftarrow$ Structure that will be filled with the labels;
for i $\leftarrow$ 1:*length(s)* **do**
    Each label consists of a "tag=value" pair. To fill the *metadata_ dict* structure, the tag-value pair must be divided into two single strings, using the *strsplit* function;
    $C \leftarrow$ Cell array containing the pair tag-value;
    tag $\leftarrow$ *C{1}*; *value* $\leftarrow$ *C{2}*;
    metadata_ dict $\leftarrow$ Assign value to structure field using the *setfield* function;
    Loop until *tag*='TASK' (History labels are irrelevant for the purpose of the code);
end for
Close the image file using the *fclose* function;
return *metadata_ dict*;

Unlike the PVOL++ browser, which has its own image viewer, to be able to choose the images that are interesting among all the available ones, these have to be converted to a format displayable by the basic programs of any computer.

Therefore, once the *Vicarread.m* function has been developed, a small code has been created which reads all the VICAR files of a data volume and stores them in the specified format. This code consists of two functions, called *Convert_ images_ all_ folders.m* and *Convert_ images_ one_ folder.m*, respectively. Next, the pseudocode of the algorithm of each of them is shown:

Convert_ images_ all_ folders.m algorithm

Input data: path of the folders with the images and format to which the images will be converted;
List the files and folders in the current folder using the *dir* function;
files ← Structure with the list of files and folders;
For each folder in the *files* structure, call *Convert_ images_ one_folder.m* function;

Convert_ images_ one_ folder.m algorithm

Input data: folder with images to be converted and format to which the images will be converted;
List the VICAR files (.IMG extension) in the current folder using the *dir* function;
files ← Structure with the list of VICAR files;
For each file in the *files* structure, call *Vicarread.m* function to read it and save it in the specified format using the *imwrite* function;

Once all the available images have been converted, they can be displayed without problems and choose the ones that are most interesting. When all the chosen images are already taken, they are calibrated for later navigation.

As it is presented in the next chapter, the calibration of the images is carried out by means of a function called "Batch Mode" available in the CISSCAL software, which allows to calibrate a large number of images at the same time.

Thus, all the images could have been calibrated before choosing the ones of most interest. However, the calibration process is not as fast as just converting them, so it is only important to calibrate the necessary images.

Chapter 4

ISS Image Calibration: CISSCAL

4.1 Introduction

CISSCAL is the Cassini Imaging Science Subsystem Calibration (CISSCAL) software and it was developed by the Cassini Imaging Central Laboratory for Operations (CICLOPS). It performs standard CCD calibration steps such as bias and dark subtraction and flatfield correction, as well as ISS-specific calibrations such as bitweight correction and removal of 2-Hz noise. CISSCAL only reads and writes images in VICAR format [3, 4, 44].

Many of the calibration options included in said software are difficult to implement and would involve excessive effort for the purpose marked in this study, so, unlike the software for reading and navigating images, in which alternative programs have been developed to those already existing, the calibration process has been carried out using CISSCAL.

Thus, in this chapter, the calibration process is explained in detail and the CISSCAL software is examined minutely.

4.2 Setting Up the Environment

CISSCAL is written in the Interactive Data Language (IDL), and thus requires IDL (version 5.5 or later) to be installed on the computer on which CISSCAL is going to be run, and its executable placed in the user's PATH [3, 44]. Throughout this project, the user uses version 6.1 of IDL to run version 3.6 of CISSCAL.

Once IDL has been installed, the first step is to set the preferences of the system, indicating the main directories and paths to look for. It is done by selecting "Preferences" in the "File" menu. Among the tabs that appear, only two of them must be modified [3, 44]:

- *Startup*: Set the "IDL Main directory" and the "Working Directory" with the corresponding paths.
- *Path*: Set the main directories in which to search for files.

Finally, the user needs only to edit the *cisscal.pro* IDL file to define the CISSCAL-specific environment variables: *CisscalDir*, *CalibrationBaseDir*, and *ImageBaseDir*. Assuming a Windows environment, that the user has installed CISSCAL in the home directory (e.g. C:/) in a subdirectory named "cisscal3.6", and that the calibration support directory "calib" has been downloaded, the following lines should be filled in the *cisscal.pro* IDL file as follows [3, 44]:

- *CisscalDir* = ‘C:/cisscal3.6/’
- *CalibrationBaseDir* = ‘C:/calib/’
- *ImageBaseDir* = ‘C:/images/’

Specifically, CisscalDir specifies the location of the CISSCAL software files, CalibrationBaseDir specifies the location of the calibration support directories, and ImageBaseDir is the default directory where CISSCAL looks for images to be calibrated [3, 44].

4.3 Starting CISSCAL

CISSCAL is launched by typing "@cisscal" at the IDL prompt, or "idl cisscal" from the terminal prompt. Doing so, the main graphics widget is launched [3, 44].

The CISSCAL menu bar contains several pull-down menus. The menu buttons are labeled "File", "Image", "Tools", "Help", "Log Options" and "Batch Mode". Below the menu bar is a text window which logs to the GUI any messages generated by CISSCAL. Various logging options can be adjusted by the user from the "Log Options" menu item, as discussed below [3, 44].

To the right of the log window is a list of calibration options to be set by the user. These options are executed in the order listed, and can be toggled on and off using the buttons in the "On/Off" column. The calibration options are [3, 44]:

- LUT conversion: A reverse-lookup table is applied to the image to convert the pixel values from an 8-bit range (0 to 255) to a 12-bit range (0 to 4096). This option only applies to images with a DATA_CONVERSION_TYPE of "TABLE".
- Bitweight correction: Corrects for uneven bit-weighting. It is not applied to LOSSY-compressed or TABLE-encoded images.

- Subtract bias: The "bias" is the zero-exposure DN level of the CCD chip.
- Remove 2-Hz noise: Cassini ISS images suffer from a particularly bothersome type of coherent noise that results in a horizontal banding pattern across the image. This noise is introduced during image readout, and has been found to have two peaks in its power spectrum near 2 Hz.
- Subtract dark: Remove the dark current and the residual bulk image (RBI) effect.
- A-B pixel pairs: Remove the bright/dark pixel pair artifacts created by the anti-blooming mode, only produced when anti-blooming mode is ON.
- Linearize: Corrects for non-linearity of the CCD response.
- Flatfield: Dustring removal and flatfield removal.
- Convert DN to flux: Multiplies image by appropriate gain to yield electrons, divides by exposure time, divides by optics area and solid angle, and divides by the system transmission integrated over wavelength for the given filter combination (called "efficiency").
- Correction factors: Filter-specific correction factors to force the observation and theory to match.
- Geometric correction: Performs a 2-D distortion transformation on the image array, to account for the geometric distortion introduced by the optical system.

With the exception of the geometric correction, the default value for each of these options is "ON" [3, 44]. The default options are explained in Section 4.4.

To the immediate left of the calibration option names is another column of buttons which toggles the "Option Parameters" field for that option. These parameters appear in the lower right of the GUI. Some calibration options does not have any user-definable parameters, while others have many, and can be controlled quite specifically (see Figure 4.1) [3, 44].

When an image is read into CISSCAL, another field pops up at the bottom of the GUI which gives the keyword values pulled out of the VICAR label. This table is not editable, and is for information purposes only [3, 44].

The general order of operations in CISSCAL is as follows [3, 44]:

1. Read in an image file by clicking on the "Open" button in the "File" menu.
2. Select the desired calibration options.

3. Select the desired parameters for each calibration option. With the "Save calibration options file" and the "Load calibration options file" buttons in the "File" menu, the current calibration options can be saved and a previously-saved calibration options file can be loaded, respectively.
4. Go to "Calibrate Image" under the "Image" menu to execute desired calibration steps in the order listed.
5. Save output to a real-format VICAR image file by clicking the "Save Image" button in the "File" menu.

This is the general order of operations for a single image file, but the procedure to be followed for the calibration of a set of images is shown in Section 4.5.

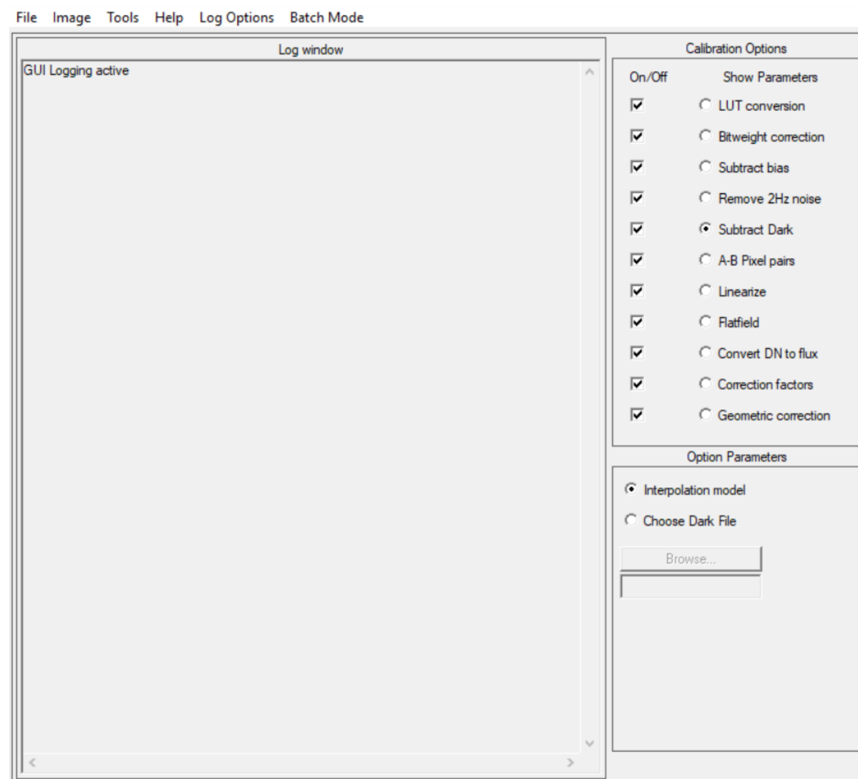

Figure 4.1: CISSCAL GUI [3, 44]

4.4 Default Options File

The CISSCAL default calibration options are specified in the *cisscal_default_options.txt* file included with the CISSCAL distribution. This file is user-editable, though care should be taken not to corrupt the formatting [3, 44].

The default options file consists of a list of 23 keywords and their values, separated by a colon. Unlike the first 22 keywords, which are calibration option, the 23rd is the default output filename suffix, which replaces the input filename suffix when the output VICAR file is written. This is initially set to '.IMG.cal' [3, 44].

4.5 CISSCAL User Manual

This project intends to use CISSCAL with the objective of calibrating a large number of images at the same time, so the user needs only to know how the "Batch Mode" option works. In this mode the "Open" button is not used at all, and images are read in sequentially and individually.

In batch mode, the user selects the calibration options and option parameters he or she desires, as usual, but then clicks on the "Calibrate Batch..." button in the "Batch Mode" menu item instead of "Calibrate Image." Doing so, a separate dialog is brought up, from which the user sets the following batch options (see Figure 4.2) [3, 44]:

- Input Directory: Directory containing images to be calibrated.
- Input Filter/File List: See description below.
- Output Directory: Where calibrated images are written.
- Output Filename Extension: File suffix to replace that of the input image filename ('.IMG.cal' set in the default options file).
- Dark Subtraction Options: Allows user to specify a list of dark files corresponding to each image (only used instead of the interpolation dark model). In this project, the "Interpolation model" option is always used.

If the "Input Filter" toggle is selected, the user can enter a regular expression to designate which files in the directory are to be calibrated. The default is *.IMG, or all image files. Another example would be N* or W*, i.e. all NACs or WACs [3, 44].

Alternatively, the user can specify use of a file list. This is just an ASCII text file containing a single-columned list of the names of images to be processed. If this file is located in the Input Directory, then full image path names are not necessary.

This last option is not used because the input filename suffix of all the images to be calibrated is '.IMG', so the "Input Filter" option is always used [3, 44].

In addition, in the batch mode the images are saved automatically in the indicated Output Directory, so the "Save image" button in the "Image" menu is not used. Likewise,

the images are saved in VICAR format, but this output VICAR file is not identical to the input file. The primary differences are [3, 44]:

- Output image is in real (floating-point) format instead of integer.
- Output image lacks binary header (including overclocked and extended pixels).
- VICAR label has PROCESSING HISTORY TEXT keyword appended, with record of calibration tasks performed.
- VICAR label has CALIBRATION STAGE keyword appended, which allows CISSCAL to automatically resume calibration of a partially-calibrated image at the same stage that it left off.

In addition to the calibrated VICAR file, an additional file of type '.SAV' is created, which contains the data of the calibrated image so that it can be opened using the PLIA software, presented below [3, 44].

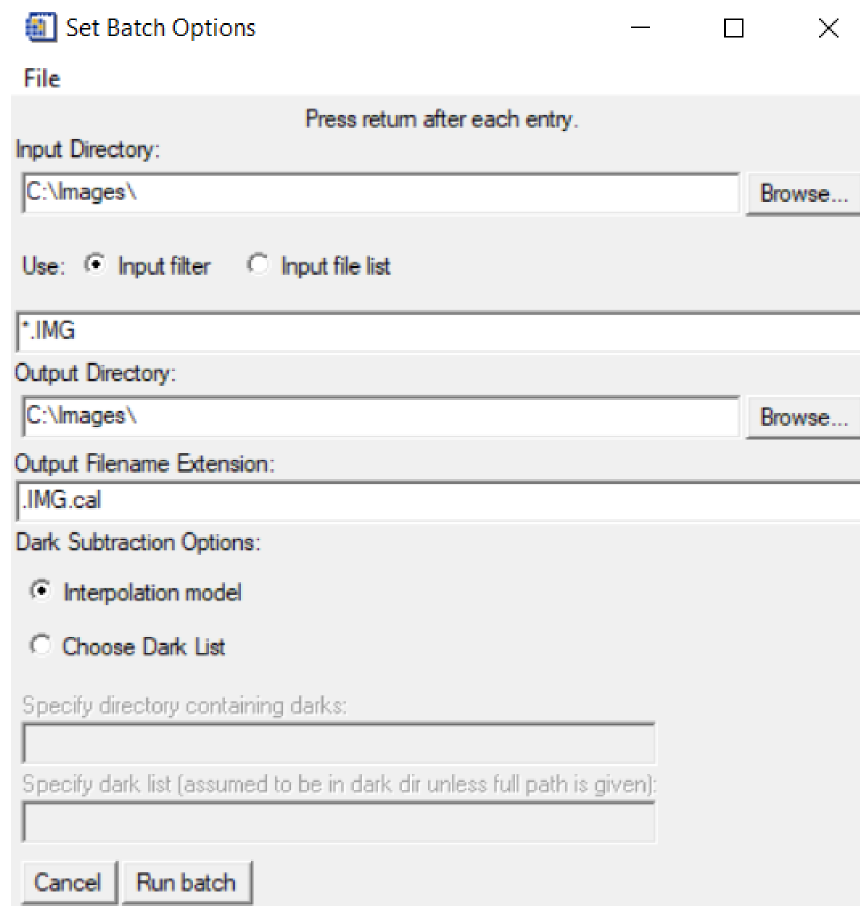

Figure 4.2: Batch Mode dialog [3, 44]

Chapter 5

Image Navigation

5.1 Planetary Laboratory for Image Analysis (PLIA)

PLIA is the Planetary Laboratory for Image Analysis software and it was developed at the *Grupo de Ciencias Planetarias* (GCP) in the University of the Basque country (UPV/EHU). It is an IDL based software, like CISSCAL, for planetary image processing and navigation [9, 8].

Up to now, this software has been used to analyze images obtained by spacecrafts such as Cassini or Galileo, so in this project it serves as validation of the results obtained with the own developed software.

5.1.1 Setting Up the Environment

Similar to CISSCAL, PLIA is written in the Interactive Data Language (IDL), and thus requires IDL to be installed on the computer on which PLIA is going to be run, and its executable placed in the user's PATH [7]. Throughout this project, the user uses version 7.1.1 of IDL to run version 3.3 of PLIA.

Once IDL has been installed, the first step is to set the preferences of the system, indicating the main directories and paths to look for. It is done by selecting "Preferences" in the "Window" menu. Among the tabs that appear, only the "Route" option in the "IDL" tab must be modified. The path of the PLIA executable must be added [7].

The user also needs to edit two files: the *plia.ini* configuration file and the *plia.pro* IDL file [9].

- The *plia.ini* file is readed by *plia.pro* and configures basic settings for PLIA. It must be placed in the PLIA directory [7]. In this file, one parameter must be modified:

  - *Image_Directory*: Directory where to look for the images to navigate. Example: 'C:/images/'.

  During the navigation with PLIA, the *kernel* (see Section 5.2) files are not used, so the parameters referring to them can be left by default.

- The *plia.pro* file is the PLIA executable and some parameters of its code must be modified [7]:
  - *Installation_path*: Path of the PLIA excutable. Example: 'C:/PLIA/'
  - *Directory*: Path of the PLIA excutable. Example: 'C:/PLIA/'

5.1.2 PLIA User Manual

Once PLIA is launched, the main graphics widget appears. The PLIA menu bar contains several pull-down menus, but as in this project this software has been used as validation, only the following buttons have been used: "File" and "Analysis".

- The "File" menu presents the basic function to open and save files. This main functions are [7]:
  - "Open" button: To open the desired image file, depending on the planetary mission (Cassini, Venus Express, Mars Express, among others). For the PLIA validation, only calibrated Cassini images have been tested, so they must be previously calibrated by CISSCAL ('.SAV' file, see Section 4.5)[1].
  - "Write Image" button: To save the processed images in the specified format (png, jpg, fits, among others).
  - "IDL Sessions" button: To save the actual PLIA session or read and continue with a previous one.
- The "Analysis" menu presents the main navigation options. The most relevant ones are [7]:
  - "Measure Image" button: To calculate the range of longitudes and latitudes associated with the planet or satellite of the image and create a grid with that range that surrounds the object in the image.

    It is worth mentioning the importance of the coordinates with which the longitudes and latitudes of a planet or satellite are measured. Thus, the two most relevant coordinate systems are:

[1]Also a '.XML' file is needed, so the calibrated image must be previously selected by the PVOL++ browser (see Section 3.3.1).

– *Planeotcentric (or Geocentric) coordinates*: The latitude is referred to the equatorial plane and the polar axis, and the longitude is measured eastwards (i.e. positive in the sense of rotation) from the prime meridian [50].
– *Planeotgraphic (or Geodetic) coordinates*: The latitude is defined as the angle between equatorial plane and a vector through the point of interest that is normal to the biaxial ellipsoid reference surface of the body. The longitude of the sub-observation point increases with time, i.e. to the west for prograde rotators and to the east for retrograde rotators [50].

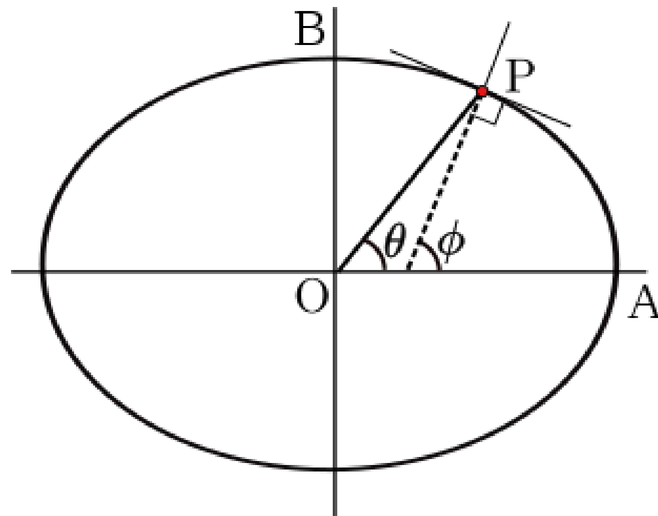

Figure 5.1: Planetocentric (θ) and planetographic (ϕ) coordinates [58]

The PLIA software computes longitudes on the so-called System III [8], that is, planetocentric coordinates, with a range of longitudes and latitudes of [0, 2π] and [-$\pi/2$, $\pi/2$] respectively, but with the longitudes measured westwards (i.e. positive in the opposite sense of rotation). So, the developed software must be adapted to work with this system as well, in order to ensure a properly validation.

- "Edit navigation" button: To manually correct the position of the grid in the image.
- "Grid options" button: To specify whether or not the grid is shown, or to define the precision of such grid.
- "Geometrical projections" button: Once the image has been measured, this option allows to create different geometrical projections of a specified range of longitudes and latitudes. In this study, only the planisphere projection has been tested.

5.2 Developed Software

In this section, the developed software for the navigation of planetary images is presented. First, an initial prototype in MATLAB has been developed, and then the same has been built in C, presenting improvements and with the possibility of starting up in parallel.

As mentioned at the beginning of the study, these developed solvers are based on the NASA's SPICE library, and the required information for the navigation of the planetary images is contained in the so-called *kernel* files[2].

Among the different types of existing *kernels*, the most important for the realization of this project is the so-called *"Camera matrix" kernel* (*C-Kernel* or *CK*), because a *CK* file holds orientation data for a spacecraft or a moving structure on the spacecraft [51]. Hence, without this *kernel*, it would be impossible to know the orientation of the camera of the spacecraft at all times, so it would be impossible to perform the navigation of images presented in this study.

This aspect has made it impossible to navigate images taken during the Voyager program, since the primary Voyager mission occurred in advance of the existence of SPICE, so the existing *CK* files are reconstructions made years later, with varying degrees of success.

Continuing with the developed software, before going into more detail, there are a series of parameters that are mandatory for the navigation of any planetary image. These parameters are:

- *Image time*: Date when the image to navigate was taken.
- *Target body*: Name of the target body seen on the image, e.g. 'JUPITER'.
- *Body-fixed frame*: Name of a body-fixed reference frame centered on the target body, e.g. 'IAU-JUPITER'.
- *Observing body*: Name of the observing body, e.g. 'CASSINI'.
- *Instrument*: Instrument name of the observing body with which the image was taken.
- *Aberration*: Aberration corrections to be applied.
- *Computation method*: Computation method to be used, e.g. 'ELLIPSOID'.

Both programs are focused on the navigation of planetary images taken by the Cassini probe. Nevertheless, more space missions have been tested with more or less success, such as Voyager.

[2]It is recommended to read the project carried out for the author's colleague Roger Sala Marco in order to expand the knowledge about the SPICE library and about the *kernel* files.

5.2.1 MATLAB Prototype

This first MATLAB prototype is divided into two parts:

- Part A: MATLAB code called *Navega_cassini_pA.m* which computes the range of longitudes and latitudes of all the pixels of a given image, as well as the illumination angles of each of them.
- Part B: MATLAB function called *Navega_cassini_pB.m* which displays a projection image of a specified range of longitudes and latitudes contained in the navigated image.

Part A

The part A of the MATLAB prototype is a set of MATLAB and SPICE functions, so first, a pseudocode of the global algorithm is presented, and later the algorithms of each MATLAB function are shown. In Appendix A.1, each SPICE function is described, and the inputs and outputs involved are discussed.

Thus, the pseudocode of the global algorithm is as follows:

Navega_cassini_pA.m algorithm Part 1

Call the *initSPICEd.m* function to load all *kernel* files;

Call the *Vicarread.m* function to read the VICAR image and return the labels structure;
img ← Array with the values of the image pixels;
utctim ← Image time from the labels structure (string);

et ← Call the *cspice_str2et* function to convert the *utctim* into Ephemeris Time;

radii ← Call the *cspice_bodvrd* function to get the radii of the target body;

Call the *cspice_gipool* function to get the geometry parameters;
NPX/NPY ← Number of image samples(columns)/lines(rows);

code ← Call the *cspice_bodn2c* function to get the instrument ID code;

Call the *cspice_getfov* function to return the field-of-view (FOV) parameters;
bounds ← Set of vectors pointing to the corners of the instrument FOV;

img3(NPY,NPX,3) ← Original image in RGB format (3-flats double precision matrix, where each of them corresponds to a colour channel: 1=Red, 2=Green, 3=Blue);

Navega_cassini_pA.m algorithm Part 2

$S \leftarrow$ Structure with the data necessary for the image projection;
$S.img \leftarrow img$;
$S.lonmat \leftarrow$ Array keeping the longitudes of the image pixels;
$S.latmat \leftarrow$ Array keeping the latitudes of the image pixels;
for i $\leftarrow$ 1:NPX **do**
  for j $\leftarrow$ 1:NPY **do**
    dvec $\leftarrow$ Pointing vector emanating from the observer to the corresponding pixel (see explanation later);

    For each image pixel, call the *cspice_sincpt* function to determine if it belongs to the target body;

    found $\leftarrow$ Logical indicating whether or not the pixel belongs to the target (if so, *found*=1);

    spoint $\leftarrow$ 3-vector (body-fixed frame) defining the surface intercept point on the target body, if exists;

    if found = 1 **then**
      Call the *cspice_reclat* function to convert the *spoint* vector to latitudinal coordinates;
      lat $\leftarrow$ Planetocentric latitude measured in radians;
      lon $\leftarrow$ Planetocentric longitude measured in radians;
      Longitude correction (see explanation later): $lon \leftarrow -lon$;
      if lon < 0 **then**
        $lon \leftarrow lon+2\pi$;
      end if

      Call the *cspice_ilumin* function to compute the illumination angles at *spoint*;
      img3(j,i,1) $\leftarrow$ 1 (range [0,1]);
      S.lonmat(j,i) $\leftarrow$ *lon*;
      S.latmat(j,i) $\leftarrow$ *lat*;
    else
      img3(j,i,3) $\leftarrow$ 1 (range [0,1]);
    end if
  end for
end for
$SDAT \leftarrow$ Table shortcut where the S structure is saved;
Unload the *kernels* and clean the *kernel pool* using the *cspice_kclear* function.

In order to compute the longitude and latitude of each pixel, a 3-vector defining the pointing vector emanating from the observer to the pixel is needed, called *dvec*. As the vectors pointing to the corner pixels are already set in the *bounds* parameter, the *dvec* vector can

be obtained as follows:

$$dvec = [\ c1(1) * (1 - lx(i)) + c2(1) * lx(i),\quad c1(2) * (1 - ly(j)) + c3(2) * ly(j),\quad 1\ ] .$$

Where:

c1(1) = *bounds(1,1)* is the x-component of the vector pointing to the upper-left pixel
c2(1) = *bounds(1,2)* is the x-component of the vector pointing to the upper-right pixel
c1(2) = *bounds(2,1)* is the y-component of the vector pointing to the upper-left pixel
c3(2) = *bounds(2,3)* is the y-component of the vector pointing to the lower-right pixel
lx is an array from 0 to 1 with NPX increments
ly is an array from 0 to 1 with NPY increments

As said in the previous pseudocode, the longitudes and latitudes obtained through the *cspice_reclat* function are planetocentric, measured eastwards, but the range of longitudes is $[-\pi, \pi]$, so they must be corrected in order to ensure the same coordinate system as in the PLIA software.

To obtain increasing westwards longitudes instead of eastwards ones, the only thing that must be done is to multiply all the longitudes by -1 (opposite longitudes); to obtain a range of longitudes of $[0, 2\pi]$, 2π must be added to the values in the range $[-\pi,0]$.

Finally, in order to verify which pixels belong to the target body, the original image has been generated in RGB format, modifying the values of each pixel according to whether or not they belong to the target body, according to the *cspice_sincpt* function. Thus, if the pixel belongs to the target body, its value changes to 1 in the first flat of the *img3* matrix, providing more red colour, but if not, its value changes to 1 in the third flat, providing more blue colour.

The result is the original image with the pixels coloured red or blue according to whether or not they belong to the target body. Below is a sample of the image obtained from navigating a Jupiter image taken by the Cassini probe:

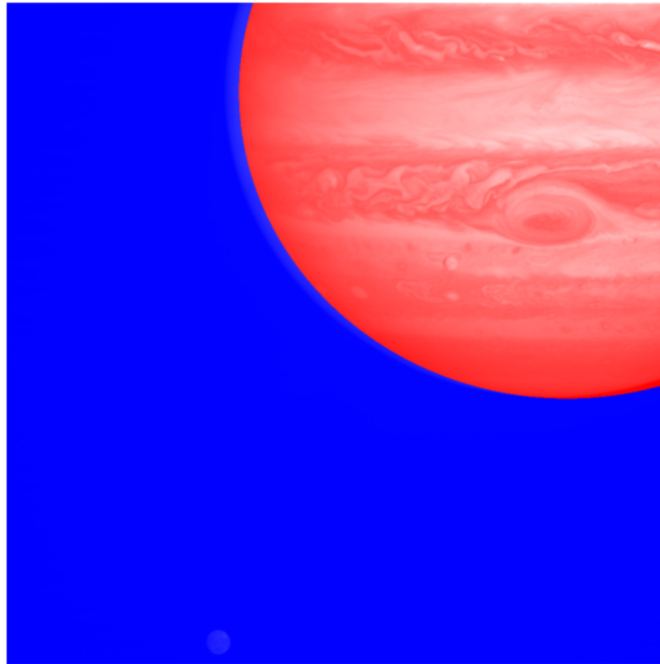

Figure 5.2: Navigated image of Jupiter through MATLAB

The navigated image seems to be correct, but may have some wrong pixels that must be corrected. The correction process is explained in Section 5.2.2.

Next, the pseudocode of the algorithm of the *initSPICEd.m* function is presented[3]:

initSPICEd.m algorithm

Input data: *METAKR* ← Cell structure of two columns: the second column contains the names of the *kernels*, and the first the URLs where to download them;

Specify the working paths:
HOMESPICE ← Path of the folder with the SPICE archives;
IMGFOLDER ← Path of the folder with the image to navigate;

for i ← 1:*length(METAKR)* **do**
  if i = Odd number **then**
    url ← *METAKR{i}*;
    continue (jump to the next i value);
  end if
  Check if the *kernel* is downloaded. If not, download the *kernel* from *url* using the *websave* function;
  Load the *kernel* using the *cspice_furnsh* function;
end for

[3]The algorithm of the *Vicarread.m* function has already been presented in Section 3.3.2

There is an additional MATLAB function called *osi.m* which exchanges the symbols '\' and '/' if needed.

Part B

As said before, this part B consists on obtaining an image shred of the original one by specifying the corresponding range of longitudes and latitudes.

In order to avoid running the image navigation every time a image projection is wanted, the code has been divided in the parts A and B already mentioned, so that with part B all the desired projections can be made without running part A again. Therefore, as seen in its pseudocode, part A saves a table shortcut (.mat file) with the structure containing all the relevant information for the realization of the image projections.

Thus, the pseudocode of the algorithm is as follows:

Navega_ cassini_ pB.m algorithm Part 1

Load the *SDAT.mat* table shortcut using the *load* function;

Create an interpolating function (interpolant) using the *scatteredInterpolant* function; *Fimg* ← *scatteredInterpolant(S.lonmat(:), S.latmat(:), double(S.img(:)))*. The interpolant fits a surface of the form *S.img(:) = Fimg(S.lonmat(:), S.latmat(:))*;

Specify the range of longitudes and latitudes:
    llon ← Left limit longitude;
    rlon ← Right limit longitude;
    dlon ← Longitude increments;
    ulat ← Upper limit latitude;
    llat ← Lower limit latitude;
    dlat ← Latitude increments;

if llon > rlon **then**
    lon1 ← *llon*;
    lon2 ← *rlon*;
else
    lon1 ← *rlon*;
    lon2 ← *llon*;
end if

if ulat > llat **then**
    lat1 ← *ulat*;
    lat2 ← *llat*;
else
    lat1 ← *llat*;

Navega_ cassini_ pB.m algorithm Part 2

lat2 ← *ulat*;
end if

lons ← *lon1:dlon:lon2*;
lats ← *lat1:dlat:lat2*;

Create 2-D grid coordinates using the *meshgrid* function;
[*lonv, latv*] ← *meshgrid(lons, lats)*. Returns a 2-D grid coordinates based on the coordinates contained in vectors *lons* and *lats*;

Get the values of the image projection by evaluating the 2-D grid coordinates with the *Fimg* interpolation function;
img2 ← *Fimg(lonv, latv)*;

As established in part A of the MATLAB prototype, the longitudes decrease eastwards and the latitudes southwards, so in an image projection the longitudes should decrease rightwards and the latitudes downwards.

Thus, in the previous pseudocode, *llon* should be higher than *rlon*, and *ulat* should be higher than *llat*. However, if for some reason it were the other way around, nothing would happen because the highest latitude and longitude are assigned to *lon1* and *lat1*, respectively, so it is always true that *lon1>lon2* and *lat1>lat2*.

Below is the image projection of the *Great Red Spot* of Jupiter shown in Figure 5.2:

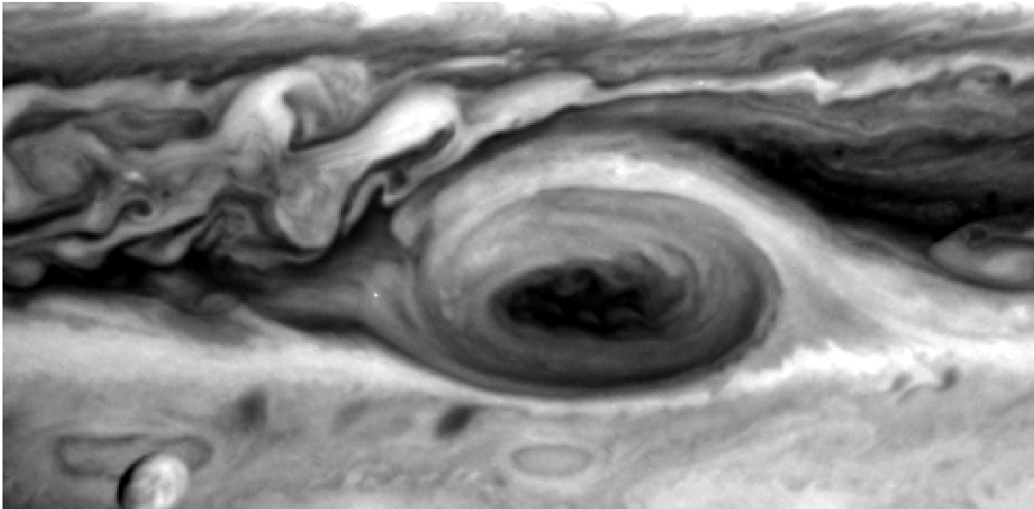

Figure 5.3: Image projection of Jupiter

It should be mentioned that the *scatteredInterpolant* MATLAB function does not exist in C language and that it is very difficult to implement manually. Therefore, only part A

has been translated into C language, while part B has been modified to be able to read the files generated by the C solver.

5.2.2 C Solver

The developed MATLAB prototype only allows one image to be navigated at a time. Even so, the execution time is too high, so it has been decided to translate it into C language to improve it and get the results as fast as possible. Thus, this C solver corresponds to the improved version of the part A of the MATLAB prototype, so the algorithms are quite similar. The most notable improvements are:

- A set of images can be navigated at the same time, not only one.
- The solver loads only the necessary *kernels*.
- The solver can be executed in parallel using a *Python* routine (see later).

As in the algorithm of part A of the MATLAB code, in this section a first global algorithm of the solver is presented, and later the functions that have appeared are discussed. Furthermore, in Appendix A.2, each SPICE function is described, although most are already explained in Appendix A.1.

In this way, this solver is divided into two C files:

- *Navega_cassini.c*: Main code of the solver.
- *kernel_d.c*: Code with most functions. They are loaded as an own library called *kernel_d.h*.

Thus, the global algorithm of the *Navega_cassini.c* code is presented first, and later the rest of the functions that intervene.

Main code

This solver has been developed in an Ubuntu 18.04 environment, based on Linux, so the C codes are compiled by the command window. Since the solver is divided into two codes, both must be run at the same time, so an executable file called *scompile* has been created. It contains the path of the SPICE archives, called *HOMESPICE*, and the command to compile the solver, which is:

gcc Navega_cassini.c kernel_d.c -I $HOMESPICE/include $HOMESPICE/lib/.a -lm*

The command above generates a file called *a.out* with the compiled program, ready to be executed. Its name can be changed, for example, to *navega* by adding the command *-o navega* at the end of the previous command.

To make the *scompile* file executable, the following command is used: *chmod +x scompile*; and to execute it: *./scompile*. The *scompile* file must be executed from its destination folder, where the C files must also be found.

Before focusing on the algorithms, there is a set of text files essential for the proper functioning of the solver. These are:

- *parameters.txt*: File with the mandatory parameters presented at the beginning of Section 5.2. 2-column file: the first column contains the label of the parameter, and the second the corresponding data. The parameters must have the following labels, in the specified order:
  - *obsrvr* = e.g. 'CASSINI'
  - *instnm* = e.g. 'CASSINI_ISS_NAC'
  - *target* = e.g. 'JUPITER'
  - *fixref* = e.g. 'IAU-JUPITER'
  - *method* = e.g. 'ELLIPSOID'
  - *abcorr* = e.g. 'NONE'
- *lbl_images.txt*: File with the names of the images to be navigated. They must be ordered by date (same order as by name).
- *genericker_cassini.txt*: File with the URLs and names of all the generic *kernels* of the Cassini missions[4]. 2-column file: the second column contains the names of the *kernels*, and the first the web's URLs where to download them.
- *ck_cassini.txt*: File with the URLs and names of all the CK *kernels* of the Cassini missions. Same structure as *genericker_cassini.txt* file.
- *spk_cassini.txt*: File with the URLs and names of all the SPK *kernels* of the Cassini missions. Same structure as *genericker_cassini.txt* file[5].

These text files are recommended to be in the same destination folder as the *scompile* and the C files.

Once the solver is compiled, the following command is used to execute it:

[4]Generic *kernels* are those that do not depend on time (FK, IK, LSK, PCK and SCLK). For a deeper knowledge about *kernel* files, it is recommended to read the project made by Roger Sala Marco.

[5]The CK and SPK *kernels* are divided into two separate files because they require different functions to read the information they contain.

./navega mode ./kernels ./Images/ lbl_ images.txt parameters.txt genericker_ cassini.txt ck_ cassini.txt spk_ cassini.txt

Where:

./navega is the compiled solver.
mode is the execution mode. The developed solver presents two execution modes: 0, in which the program downloads all the *kernels*, and 1, in which, the program navigates the images.
./kernels is the path of the *kernels* folder.
./Images is the path of the folder with the images.
lbl_ images.txt is the file with the name of the images to navigate.
parameters.txt is the file with the input parameters.
genericker_ cassini.txt is the file with the URLs and names of the generic *kernels*.
ck_ cassini.txt is the file with the URLs and names of the CK *kernels*.
spk_ cassini.txt is the file with the URLs and names of the SPK *kernels*.

Once the previous command is executed, the program receives the following variables:

argc: Integer that contains the number of arguments entered in the command.
argv: Double pointer to an array of characters that contains the parameters passed in the same order in which they were written.

In this case, *argc* contains the value 9, and *argv[0]* contains *./navega*, *argv[1]* contains *mode*, this way until *argv[8]*[6], which contains *spk_ cassini.txt*.

There are also a list of libraries that must be included on the code for its proper functioning:

#include <stdio.h> → Required for performing input and output declarations.
#include <stdlib.h> → Required for performing general functions.
#include <unistd.h> → Provides access to the POSIX operating system API.
#include <string.h> → Required for manipulating arrays of characters.
#include "SpiceUsr.h" → Performs SPICE user interface declarations.
#include "kernel_d.h" → Own library with most of the developed functions.

Next, the pseudocode of the global algorithm of the *Navega_ cassini.c* code is shown:

[6]Note that, unlike MATLAB, in which the indexes start at 1, in C language they begin at 0, so *argv* is an array with 9 positions, with indexes from 0 to 8.

Navega_ cassini.c algorithm Part 1

if argc $\neq 9$ **then**
 An error message is printed on screen, indicating the correct command to execute the solver. The current execution is interrupted;
end if
int *mode* ← *argv[1]*. The execution mode, read it as character, is assigned to the variable *mode*. First it is converted to an integer using the *atoi* C routine;
if $mode < 0 \,||\, mode > 0$ **then**
 An error message is printed on screen and the current execution is interrupted. The *mode* value must be 0 or 1;
end if
Check if the *kernels* folder exists using the *set_ home_ kernels* function. If not, create it;
if $mode == 0$ **then**
 Download all the *kernels* using the *only_ download* function;
else
 Check if the *Images* folder exists using the *home_ images* function. If not, interrupt the execution;

 char ***images* ← Double pointer to an array of characters that contains the names of the images to navigate. The allocation is done using the *read_ lbl* function, which also returns the number of images to navigate (*length*);

 char ***param* ← Double pointer to an array of characters that contains the mandatory input parameters. The allocation is done using the *read_ param* function;

 ets[length] ← Initialize array of Ephemeris Times;
 Load the generic *kernels* (required for the *str2et_ c* function below) using the *furnsh_ d_ all* function (*cas*=1, see *furnsh_ d_ all* algorithm later);
 for i ← *0:length−1* **do**
  char **utctim* ← Pointer to an array of characters that contains the dates when the images were taken. The allocation is done using the *read_ utctime* function;
  Convert each date in **utctim* to Ephemeris Time using the *str2et_ c* function;
  ets[i] ← Ephemeris Times of the corresponding image;
 end for
 Load the CK *kernels* using the *furnsh_ d_ all* function (*cas*=2).
 Load the SPK *kernels* using the *furnsh_ d_ all* function (*cas*=3).
 radii ← Call the *bodvrd_ c* function to get the radii of the target body;
 code ← Call the *bodn2c_ c* function to get the instrument ID code;
 Call the *gipool_ c* function to get the geometry parameters;
 NX/NY ← Number of image samples(columns)/lines(rows);

Navega_cassini.c algorithm Part 2

Call the *getfov_c* function to return the field-of-view (FOV) parameters;
bounds ← Set of vectors pointing to the corners of the instrument FOV;
for k ← *0:length−1* **do**
  Navigation process for each image:
  m ← Pointer to an array of integers that allocates 0s and 1s depending on whether the corresponding pixel belongs to the target body or not;
  lonmat ← Pointer to an array of doubles that allocates the longitude of each pixel;
  latmat ← Pointer to an array of doubles that allocates the latitude of each pixel;
  solar ← Pointer to an array of doubles that allocates the solar angle of each pixel;
  for j ← *0:NY−1* **do**
    for i ← *0:NX−1* **do**
      dvec ← Pointing vector emanating from the observer to the corresponding pixel;
      For each image pixel, call the *sincpt_c* function to determine if it belongs to the target body;
      found ← Logical indicating whether or not the pixel belongs to the target (if so, *found*=SPICETRUE);
      spoint ← 3-vector (body-fixed frame) defining the surface intercept point on the target body, if exists;
      if *found* **then**
        m ← 1;
        Call the *reclat_c* function to convert the *spoint* vector to latitudinal coordinates;
        lat ← Planetocentric latitude measured in radians;
        lon ← Planetocentric longitude measured in radians;
        Convert *lat* and *lon* from radians to degrees using the *convrt_c* function;
        Longitude correction: $lon \leftarrow -lon$;
        if lon < 0 **then**
          $lon \leftarrow lon+360$;
        end if
        lonmat ← *lon*
        latmat ← *lat*;
        Call the *ilumin_c* function to compute the illumination angles at *spoint*;
        incdnc ← Solar angle in radians;
        Convert *incdnc* from radians to degrees using the *convrt_c* function;
        solar ← *incdnc*;

Navega_ cassini.c algorithm Part 3

else
$m \leftarrow 0$;
$lonmat \leftarrow -1000$;
$latmat \leftarrow -1000$;
$solar \leftarrow -1000$;
end if
end for
end for
Generate a Portable PixMap (.ppm) red-blue image using the *generappm* function;
Generate a text file with the longitudes and latitudes of the image pixels using the *generalonlat* function;
end for
Unload the *kernels* and clean the *kernel pool* using the *kclear_ c* function.
end if

Relevant functions

As said before, most of the functions seen in the previous algorithm are in the *kernel_ d.c* file, but not all, since some are declared in the *Navega_ cassini.c* file.

The pseudocodes of the algorithms of the functions seen above are presented below:

set_ home_ kernels algorithm

Input data: *argv[2]* ← Path of the *kernels* folder;
homekernels ← *argv[2]*;
Check if the folder exists using the *exists_ c* function;
if *exists_ c(homekernels) == SPICEFALSE* **then**
Call the system to create a folder with the name in *homekernels* using the *system* function. The command is *mkdir -p homekernels*;
end if
return *homekernels* (available for all the functions below);

only_ download algorithm

Input data: The text files with the URLs and names of the *kernels* (*argv[6]*, *argv[7]* and *argv[8]*);

kfile ← 3-row array containing the *kernels* text files:
  kfile[0] ← *argv[6]*;
  kfile[1] ← *argv[7]*;
  kfile[2] ← *argv[8]*;

for i = *0:2* **do**
  Open *kfile[i]* in reading mode (*'r'*) using the *fopen* function;
  while true **do**
    For each line of the file:
    l ← Read the line using the *fgets* function;
    Get the length of the line using the *strlen* function;
    Check that is not a empty line:
    if *strlen(l) == 0* **then**
      continue;
    end if

    Check for comments on the line:
    for j = *0:strlen(l)* **do**
      if *l[j]* == '#' **then**
        l[j] = '/0' (force the end of line);
      end if
    end for
    Check that is not a empty line again (if the whole line was a comment, it has been converted to a blank line);

    Scan the line to get the URL and the name of the *kernel*:
      wname ← URL of the *kernel*;
      kname ← Name of the *kernel*;

    fullkname ← Path of *kname* in the *homekernels* folder;
    Check if *kname* exists in the *homekernels* folder using the *exists_ c* function;
    if *exists_ c(fullkname) == SPICEFALSE* **then**
      Call the system to download *kname* from *wname* using the *system* function. The command is *wget -P wname*;
    end if
  end while
  Close the file using the *fclose* function;
end for

home_images algorithm

Input data: *argv[3]* ← Path of the *Images* folder;
Check if the folder exists using the *exists_c* function;
if *exists_c(argv[3]) == SPICEFALSE* **then**
    An error message is printed on screen and the current execution is interrupted;
end if

read_lbl algorithm

Input data: *argv[4]* ← Text file with the names of the images to navigate;
count ← 0 (initialize a counter);
***images* ← Memory allocation using the *malloc* routine;
Open the file in reading mode (*'r'*) using the *fopen* function;
while true **do**
    For each line of the file:
    l ← Read the line using the *fgets* function;
    Get the length of the line using the *strlen* function;
    Check that is not a empty line:
    if *strlen(l) == 0* **then**
        continue;
    end if
    Check for comments on the line:
    for j = *0:strlen(l)* **do**
        if *l[j]* == '#' **then**
            l[j] = '/0' (force the end of line);
        end if
    end for
    Check that is not a empty line again (if the whole line was a comment, it has been converted to a blank line);
    Scan the line to get the name of corresponding image:
        lbl ← Name of the image;
    images[count] ← *lbl*;
    count++;
end while
length ← *count* (number of images to navigate);
Close the file using the *fclose* function;
return *images, length*;

read_param algorithm

Input data: *argv[5]* ← Text file with the mandatory input parameters;
count ← 0 (initialize a counter);
param_lbl ← Initialize array of parameters;
***param* ← Memory allocation using the *malloc* routine;
p ← Array with the labels of the parameters, allocated in a specific order;
p ← *{"obsrvr", "instnm", "target", "fixref", "method", "abcorr"}*;
Open the file in reading mode (*'r'*) using the *fopen* function;
while true **do**
  For each line of the file:
  l ← Read the line using the *fgets* function;
  Get the length of the line using the *strlen* function;
  Check that is not a empty line:
  if *strlen(l) == 0* **then**
    continue;
  end if
  Check for comments on the line:
  for j = *0:strlen(l)* **do**
    if *l[j]* == '#' **then**
      l[j] = *'/0'* (force the end of line);
    end if
  end for
  Check that is not a empty line again (if the whole line was a comment, it has been converted to a blank line);
  Scan the line to get the label and the value of the corresponding parameter:
    param_lbl[count] ← Label of the parameter;
    value ← Value of the parameter;
  Compare *param_lbl[count]* and *p[count]* using the *strcmp* function;
  if *strcmp(param_lbl[count], p[count])* ! = *0* **then**
    An error message is printed on screen and the current execution is interrupted;
  end if
  param[count] ← *value*;
  count++;
end while
Close the file using the *fclose* function;
return *param*;

furnsh_d_all algorithm

Input data: Text file with the corresponding *kernels* to load (*argv[6]*, *argv[7]* or *argv[8]*);
ets ← Array of Ephemeris Times;
length ← Number of images to navigate;
Function mode (*cas*=1, *cas*=2 or *cas*=3, respectively);

Open the file in reading mode (*'r'*) using the *fopen* function;

while true **do**

For each line of the file:
l ← Read the line using the *fgets* function;
Get the length of the line using the *strlen* function;
Check that is not a empty line:
if *strlen(l) == 0* **then**
continue;
end if

Check for comments on the line:
for j = *0:strlen(l)* **do**
if *l[j]* == '#' **then**
l[j] = '/0' (force the end of line);
end if
end for
Check that is not a empty line again (if the whole line was a comment, it has been converted to a blank line);

Scan the line to get the URL and the name of the *kernel*:
wname ← URL of the *kernel*;
kname ← Name of the *kernel*;

Call the *furnsh_d* function to load *kname*;

end while

Close the file using the *fclose* function;

furnsh_d algorithm Part 1

Input data: *kname* ← *Kernel* to load;
ets ← Array of Ephemeris Times;
length ← Number of images to navigate;
cas ← Function mode;

Check that *homekernels* folder has been set. If not, an error message is printed on screen, indicating that the *set_home_kernels* function must be called first. The current execution is interrupted;
fullkname ← Path of *kname* in the *homekernels* folder;
Check if *kname* exists in the *homekernels* folder using the *exists_c* function;

furnsh_d algorithm Part 2

if *exists_c(fullkname) == SPICEFALSE* **then**
  An error message is printed on screen, indicating that the download of all the *kernels* is needed. The current execution is interrupted;
end if
ids ← Initialize cell array of ID codes;
cover ← Initialize cell array of coverage windows;
switch *cas* **do**
  case 1
    Load *fullkname* using the *furnsh_c* function;
  case 2
    Reinitialize *ids* array using the *scard_c* function;
    ids ← Find the set of ID codes of all objects in *fullkname* using the *ckobj_c* function (CK file);
    Get the cardinality of *ids* using the *card_c* function;
    for i = *0:card_c(ids)* **do**
      obj ← *ids[i]*
      Reinitialize *cover* array using the *scard_c* function;
      ids ← Find the coverage window in *fullkname* for *obj* using the *ckcov_c* function (CK file);
      Call the *furnsh_t* function to check the coverage window and load *fullkname* if needed. Also a parameter indicating if *fullkname* has been loaded (*load*) is returned (if so, *load*=1);
      if *load* == 1 **then**
        break;
      end if
    end for
  case 3
    Reinitialize *ids* array using the *scard_c* function;
    ids ← Find the set of ID codes of all objects in *fullkname* using the *spkobj_c* function (SPK file);
    Get the cardinality of *ids* using the *card_c* function;
    for i = *0:card_c(ids)* **do**
      obj ← *ids[i]*
      Reinitialize *cover* array using the *scard_c* function;
      ids ← Find the coverage window in *fullkname* for *obj* using the *spkcov_c* function (SPK file);
      Call the *furnsh_t* function to check the coverage window and load *fullkname* if needed. Also a parameter indicating if *fullkname* has been loaded (*load*) is returned (if so, *load*=1);

furnsh_ d algorithm Part 3

```
            if load == 1 then
                break;
            end if
        end for
    default
        An error message is printed on screen, indicating that the function mode is not valid.
        The current execution is interrupted;
end switch
```

furnsh_ t algorithm

```
Input data: cover ← Coverage window;
            kname ← Number of images to navigate;
            ets ← Array of Ephemeris Times;
            length ← Number of images to navigate;
niv ← Get the cardinality of cover using the wncard_ c function;
for j = 0:niv do
    Get the left and right endpoints of the j interval in cover window;
    et1 ← Left endpoint of the j interval in cover;
    et2 ← Right endpoint of the j interval in cover;
    for i = 0:length−1 do
        Check if ets[i] is between the endpoints of the j interval in cover;
        if ets[i] >= et1 && ets[i] <= et2 then
            fullkname ← Path of kname in the homekernels folder;
            Load fullkname using the furnsh_ c function;
            load ← 1;
            break;
        end if
    end for
    if load ← 1 then
        break;
    end if
end for
return load;
```

read_ utctime algorithm

```
Input data: argv[3] ← Path of the folder with the images to navigate;
            images[i] ← Name of the corresponding image;
fullimage ← Path of images[i] in the argv[3] folder;
utctim ← Command to get the date when the image was taken. The command is grep
IMAGE_ TIME fullimage.LBL |cut -d\"=\" -f 2 |tr -d '\' |tr -d ' ';
Call the system to execute the utctim command using the popen function in reading
mode ('r');
utctim ← Get the result of the popen execution using the fgets function;
Close the popen execution using the pclose function;
return utctim;
```

generappm algorithm

```
Input data: argv[3] ← Path of the folder with the images to navigate;
            ppm ← PPM file (name of the image + .ppm extension);
            NX ← Number of image samples;
            NY ← Number of image lines;
            m ← Array of 0s and 1s;
            solar ← Solar angles of the image pixels;
path ← Path of ppm in the argv[3] folder;
Create an empty ppm file using the fopen function in writing (binary) mode ('wb');
Write the header in ppm using the fprintf function (see PPM file later);
Check for each image pixel if it belongs to the target body:
for py = 0:NY-1 do
    for px = 0:NX-1 do
        Initialize color array:
            color[0] ← 0;
            color[1] ← 0;
            color[2] ← 0;
        if m = 1 then
            if solar < 90 then
                color[0] ← 255;
            end if
        else
            color[2] ← 255;
        end if
        Write in ppm the values of color of the corresponding pixel using the fwrite
        function;
    end for
end for
```

The Portable Pixmap (PPM) format is an image file format designed to generate color RGB images. Each PPM file starts with a two-byte magic number, P3 or P6, depending on its encoding (ASCII or binary) [55]. In the *generappm* function, in which the PPM image is written in binary format, the header is as follows:

P6 ← Two-byte magic number (RGB colour image in binary)
NX NY ← Width and height of the image in pixels
255 ← Maximum value for each colour

As mentioned in Section 5.2.1, an image in RGB format consists of a 3-flats double precision matrix, where each of them corresponds to a colour channel: 1=Red, 2=Blue and 3=Green. In this way, in PPM files, a 0 value in each channel represents the colour black, and a 255 value in each channel represents the colour white.

Therefore, going back to the algorithm above, if the pixel does not belong to the target body, its colour is blue (255 value in the third channel), but if does, its colour depends on its solar angle, so that if the pixel is illuminated, its colour is red (255 value in the first channel), but if it is not, its value is black. The following figure shows the different illumination angles computed by the *ilumin_ c* function:

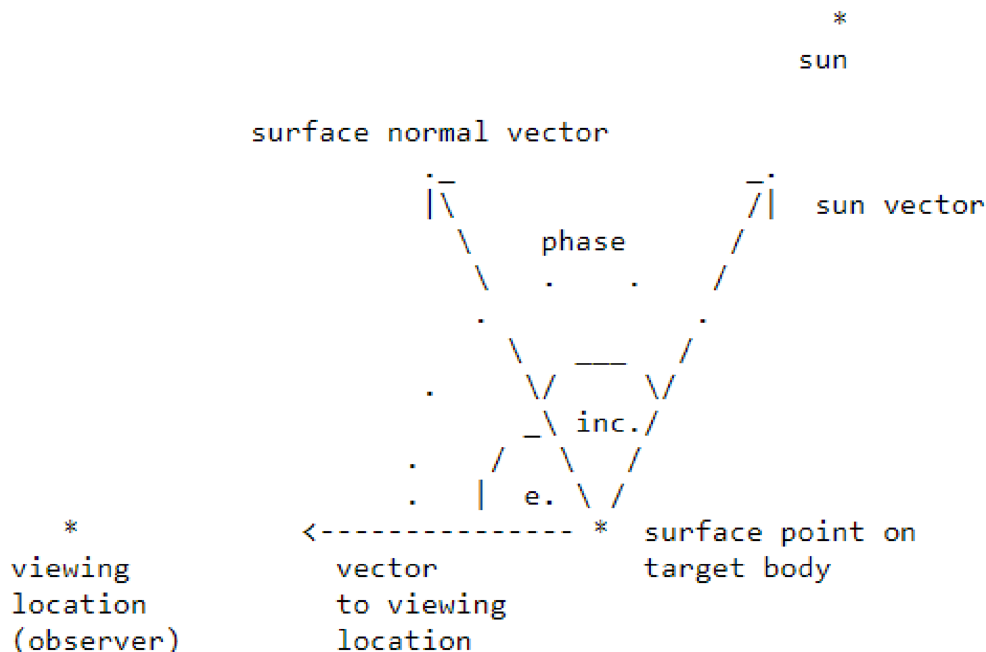

Figure 5.4: Illumination angles [49]

As it can be seen in Figure 5.4, the solar angle (inc.) is the angle between the surface normal vector at the corresponding point and the point-sun vector. Thus, the first points

with no illumination, called *terminator* or *twilight zone* [56], are those in which the point-sun vector is tangent to the surface. This corresponds to a solar angle of 90º, so the illuminated points are those in which the solar angle is less than 90º.

---

generalonlat algorithm

Input data: *argv[3]* ← Path of the folder with the images to navigate;
lonlat ← Longitudes and latitudes file (name of the image + _lonlat.txt);
NX ← Number of image samples;
NY ← Number of image lines;
m ← Array of 0s and 1s;
lonmat ← Longitudes of the image pixels;
latmat ← Latitudes of the image pixels;

path ← Path of *lonlat* in the *argv[3]* folder;
Create an empty *lonlat* file using the *fopen* function in writing (binary) mode (*'wb'*);
for py = *0:NY-1* **do**
  for px = *0:NX-1* **do**
    Write in *lonlat* the coordinates in pixels (*px* and *py*), the longitude (*lonmat*) and the latitude (*latmat*) of the corresponding pixel using the *fwrite* function;
  end for
end for

Python routine

As mentioned before, the goal of translating the MATLAB prototype to C language is to reduce the execution time and obtain the results as soon as possible. The best way to do this is to start the program in parallel, that is, distributing the work to be done between the different computer processors.

First, it was thought to run the program through Message Passing Interface (MPI), so that each processor made a part of the navigation. However, the program is designed to navigate more than one image at a time, so it is more obvious to distribute the images among the different processors. In this way, the work of each processor is independent, so programming by MPI is not very useful. So, finally a python code has been made, which equally distributes the images to be navigated between the different processors.

The Ubuntu 18.04 environment incorporates by default a python interpreter, so a python file, called *navegatots.py*, has been developed in order to run the C solver in parallel. Thus, this python file must be executed from its destination folder, where the C files must also be found. Once the solver is compiled through the *scompile* file, the following command is used to execute it: *python navegatots.py*.

The pseudocode of the python file algorithm is shown below:

navegatots.py algorithm

fname ← File with the names of the images to be navigated;
NP ← Number of processors between which images to be navigated are distributed;

Execute the solver to download all the *kernel* files using the *navega* command with *mode=0*;

Open and read *fname*;

Create an empty file of images for each processor, called *file_p.txt*, where *p* is the number of the corresponding processor;

Get the initial and final image indexes for each processor using the *worksplit* function;

Write the images to be navigate by each processor to each corresponding *file_p.txt*;

For each processor, execute the solver to navigate the images in the corresponding *file_p.txt* using the *navega* command with *mode=1* (*file_p.txt* instead of *lbl_images.txt*);

Below is the pseudocode of the *worksplit* algorithm:

worksplit algorithm

Input data: *start* ← Index of the first image to be navigated, equal to 0;
end ← Index of the last image to be navigated (number of images − 1);
me ← Number of the corresponding processor;
P ← Number of processors;
Calculate the number of images to be navigated: $N \leftarrow end - start + 1$;
mystart ← *start*;
remainder ← Get the remainder of dividing the total number of images by the number of processors (remainder of N/P);
if $P > N$ **then**
  An error message is printed on screen, indicating that there are too many processors. The current execution is interrupted;
end if
for pp = *0:P-1* **do**
  if pp < *remainder* **then**
    myshare ← $N/P + 1$ (integer number, without decimals);
  else
    myshare ← N/P (integer number, without decimals);
  end if
  if pp == *me* **then**
    break;
  end if
  mystart ← **myend** + **1**;
end for
return *mystart, myend*;

Image projection

Finally, the part B of the MATLAB prototype has been modified to be able to read the files generated by the C solver. This new code, unlike the old one, makes the projections of a set of images, so first the *lbl_images.txt* file is loaded.

Then, for each image, the corresponding text file with the longitudes and latitudes generated with the C solver (*lonlat* file) is loaded. Both files are loaded using the *importdata* function.

In addition, in the new code appears a new MATLAB function, called *Navega_correction.m*, that serves to correct the possible errors of the navigation made in C, aspect that had not been taken into account in the MATLAB prototype.

Hence, the algorithm is as follows:

Navega_cassini_pB_C.m algorithm Part 1

Load the data of *lbl_images.txt* file using the *importdata* function;
folder ← Path of the folder where to look for the navigation images and files;
files ← Cell array with the names of the images navigated;
for i = *1:length(files)* **do**
  A ← Load the data of the corresponding *lonlat* file of *files[i]*;
    A(:,1) ← X pixel (column);
    A(:,2) ← Y pixel (row);
    A(:,3) ← Longitude;
    A(:,4) ← Latitude;
  img ← Call the *Vicarread.m* function to read *files[i]* and get the values of the image pixels;

  Correct the longitudes and latitudes of the image navigation using the *Navega_correction.m* function;

  img2 ← *double(img')*;
  Create an interpolating function (interpolant) using the *scatteredInterpolant* function;
  Fimg ← *scatteredInterpolant(A(:,3), A(:,4), img2(:))*. The interpolant fits a surface of the form *img2(:) = Fimg(A(:,3), A(:,4))*;

  Specify the range of longitudes and latitudes:
    llon ← Left limit longitude;
    rlon ← Right limit longitude;
    dlon ← Longitude increments;
    ulat ← Upper limit latitude;
    llat ← Lower limit latitude;
    dlat ← Latitude increments;
  if llon > rlon **then**

Navega_ cassini_ pB_ C.m algorithm Part 2

lon1 ← *llon*;
lon2 ← *rlon*;
else
lon1 ← *rlon*;
lon2 ← *llon*;
end if

if ulat > llat **then**
lat1 ← *ulat*;
lat2 ← *llat*;
else
lat1 ← *llat*;
lat2 ← *ulat*;
end if

lons ← *lon1:dlon:lon2*;
lats ← *lat1:dlat:lat2*;

Create 2-D grid coordinates using the *meshgrid* function;
[*lonv, latv*] ← *meshgrid(lons, lats).* Returns a 2-D grid coordinates based on the coordinates contained in vectors *lons* and *lats*;

Get the values of the image projection by evaluating the 2-D grid coordinates with the *Fimg* interpolation function;
img2 ← *Fimg(lonv, latv)*;
end for

For the correction of the navigations, the function *Navega_ correction.m* presents two methods, both with the same degree of success. In both methods, first of all, a navigation of the image is generated according to its illumination, that is, according to the values of the pixels. Thus, for the correction only the pixels corresponding to the daylight side of the target body are taken into account, without having those corresponding to the night side.

In this way, a minimum value of light must be defined in order to differentiate the illuminated area from the unlighted area and from the space. This value defines the degree of success of the correction, regardless of the method used. However, in some images, moons appear orbiting the target body, so that the lighting values are usually the same. Thus, the generated navigations have to be corrected so that the possible moons do not appear illuminated.

The first method is to find the limits of the daylight side of the target body, both in the navigation generated through the illumination, which would be the correct navigation, and in the navigation generated through the C solver, which may has some errors. Then,

compare the limits and calculate the difference in pixels, to know the deviation of the C navigation and move the matrices of longitudes and latitudes according to it.

The second method is to find the centroid of the daylight side of the target body, both in the navigation generated through the illumination and in the navigation generated through the C solver. Then, compare the centroids and calculate the difference in pixels, to know the deviation of the C navigation and move the matrices of longitudes and latitudes according to it.

Next, the pseudocode of the algorithm of the *Navega_ correction.m* function is presented:

Navega_ correction.m algorithm Part 1

Input data: *A* ← Array with the navigation data of *files[i]*;
folder ← Path of the folder where to look fr the navigation images and files;
files[i] ← Name of the image whose navigation is to be corrected;
n ← Correction method;
af ← Call the *Vicarread.m* function to read *files[i]* from *folder* and get the values of the image pixels;
B ← Reshape the matrix of longitudes using the *reshape* function;
B ← *B'* (transposed matrix);
C ← Reshape the matrix of latitudes using the *reshape* function;
C ← *C'* (transposed matrix);

Generate the image navigation through the illumination values:
ilum ← Minimum illumination value of the daylight side of the target body;
bf ← Initialize the image navigation generated through the illumination values (double precision RGB image);
for i = *1:size(af,1)* **do**
  for j = *1:size(af,2)* **do**
    if *af* < *ilum* **then**
      bf(i,j,3) ← 1;
    else
      bf(i,j,1) ← 1;
    end if
  end for
end for
Correct the *bf* image in case there is a moon using the *Moon_ correction.m* function (see algorithm later);
b ← Read the corresponding *ppm* image of *files[i]* from *folder* using the *imread* function;
Correction process:
switch n **do**
  case 1

Navega_ correction.m algorithm Part 2

Get the limits of *bf* and *b* using the *Limits_ correction.m* function (see algorithm later);
$hl_pix \leftarrow$ Left limit deviation: $|xl_original - xl_navigated|$;
$hr_pix \leftarrow$ Right limit deviation: $|xr_original - xr_navigated|$;
if $hr_pix > hl_pix$ **then**
  if $xr_original > xr_navigated$ **then**
    $h_pix \leftarrow hr_pix$;
  else
    $h_pix \leftarrow -hr_pix$;
  end if
else
  if $xl_original > xl_navigated$ **then**
    $h_pix \leftarrow hl_pix$;
  else
    $h_pix \leftarrow -hl_pix$;
  end if
end if
$vu_pix \leftarrow$ Upper limit deviation: $|yu_original - yu_navigated|$;
$vl_pix \leftarrow$ Lower limit deviation: $|yl_original - yl_navigated|$;
if $vl_pix > vu_pix$ **then**
  if $yl_original > yl_navigated$ **then**
    $v_pix \leftarrow vl_pix$;
  else
    $v_pix \leftarrow -vl_pix$;
  end if
else
  if $yu_original > yu_navigated$ **then**
    $v_pix \leftarrow vu_pix$;
  else
    $v_pix \leftarrow -vu_pix$;
  end if
end if
Shift *b*, *B* and *C* using the *imtranslate* function (error due to rotation is not considered);

case 2
$s_original \leftarrow$ Get the pixel coordinates of the centroid of *bf* using the *regionprops* function ('centroid' mode);
$s_navigated \leftarrow$ Get the pixel coordinates of the centroid of *b* using the *regionprops* function ('centroid' mode);

Navega_ correction.m algorithm Part 3

h_ pix ← Horizontal deviation: *round*(|*s_ original(1)*−*s_ navigated(1)*|);
if *s_ original(1)* < *s_ navigated(1)* **then**
  h_ pix ← −*h_ pix*;
end if
if *s_ original(2)* < *s_ navigated(2)* **then**
  v_ pix ← −*v_ pix*;
end if
Shift *b*, *B* and *C* using the *imtranslate* function (error due to rotation is not considered);
end switch
B ← *B'* (transposed matrix);
B ← *B(:)* (matrix of longitudes as single column);
C ← *C'* (transposed matrix);
C ← *C(:)* (matrix of latitudes as single column);
return *B, C*;

As mentioned before, the appearance of a moon in the image is a problem for the correction process, so it must be corrected beforehand. For this, the difference in size between the moon and the target body has been used.

In the case of the target body, the number of consecutive illuminated pixels is almost always high, while in the moon the value are low. Thus, a correction factor (m) has been established, so that a consecutive series of illuminated pixels is considered part of the target body only if it has more than m values, otherwise it is considered part of the moon. Below is the pseudocode of the algorithm of the *Moon_ correction.m* function:

Moon_ correction.m algorithm Part 1

Input data: *bf* ← Image navigation generated through the illumination values;
Convert *bf* to gray scale image using the *rgb2gray* function;
b ← Generate a binary image from *bf* using the *imbinarize* function;
B ← Recognize the shapes in *b* using the *bwboundaries* function;
Get the number of pixels of the largest shape (target body shape):
max ← Initialize the maximum value of pixels of a shape;
for i = *1:size(B,1)* **do**
  m ← *size(Bi,1)*;
  if *m)* > max **then**
    max ← *m*;
  end if
end for

Moon_ correction.m algorithm Part 2

bf(:,:,1) ← Remove the shapes with less than *max* pixels using the *bwareaopen* function;
bf(:,:,3) ← Compute the complement of the *bf(:,:,1)* image using the *imcomplement* function;
return *bf*;

In the following pseudocode, the algorithm of the *Limits_ correction.m* function is shown:

Limits_ correction.m algorithm

Input data: *a* ← Image navigation whose limits are to be computed (*bf* or *b*);
xl ← 1e5 (initialize left pixel limit);
xr ← 0 (initialize right pixel limit);
for i = *1:size(a,1)* **do**
    jl ← Find the first 1 value in *a(i,:,1)* using the *find* function ('first' mode);
    jr ← Find the last 1 value in *a(i,:,1)* using the *find* function ('last' mode);
    if $jl < xl$ **then**
        $xl \leftarrow jl$;
    end if
    if $jr > xr$ **then**
        $xr \leftarrow jr$;
    end if
end for
yu ← 1e5 (initialize upper pixel limit);
yl ← 0 (initialize lower pixel limit);
for j = *1:size(a,2)* **do**
    iu ← Find the first 1 value in *a(:,j,1)* using the *find* function ('first' mode);
    il ← Find the last 1 value in *a(:,j,1)* using the *find* function ('last' mode);
    if $iu < yu$ **then**
        $yu \leftarrow iu$;
    end if
    if $il > yl$ **then**
        $yl \leftarrow il$;
    end if
end for
return *xl, xr, yu, yl*;

In addition to the two correction methods already explained, a third method has been tried based on image registering, using the *detectSURFFeatures* [24] MATLAB function, so that the characteristics of both navigations are detected, a transformation matrix is estimated and the corrected navigation is obtained. However, the transformation matrix

obtained is not applicable to the matrices of longitudes and latitudes, so this correction method does not allow to obtain the corrected longitudes and latitudes[7].

[7]Visit [24] to see correction process.

Chapter 6

Results and Validation

In this chapter, the results of navigating different interplanetary images by means of the C solver are presented, as well as the subsequent projection and correction processes by means of MATLAB. In this sense, the results obtained are compared with those of PLIA, so that they are contrasted and validated.

First of all, as seen in Section 5.2.2, once the solver is run in C, a PPM image with the colours red, blue and black is generated, depending on the area of the image. Thus, the result of navigating the same image as that of Figure 5.2 through C gives the following result:

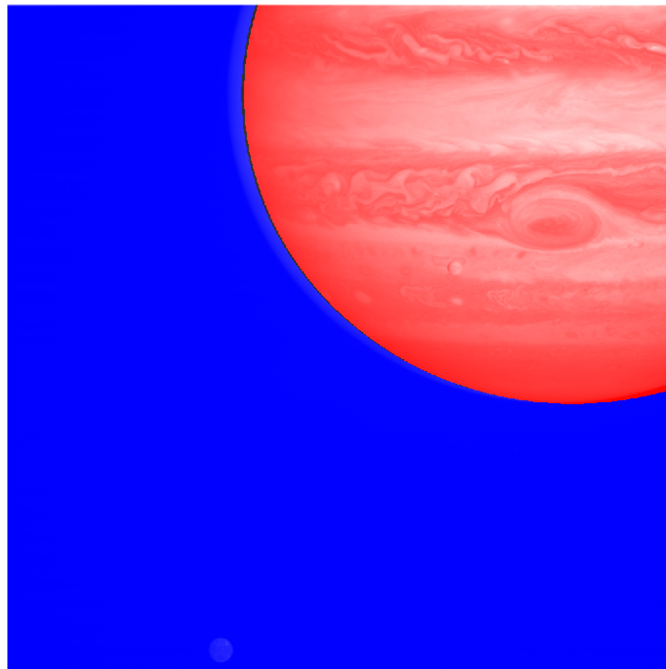

Figure 6.1: Navigated image of Jupiter through C solver

Unlike Figure 5.2, now the generated image presents a third zone of black colour, which corresponds to the area of the unlighted planet, which in this case is a small portion.

At first glance, it seems that the obtained result resembles that obtained by the MATLAB prototype. However, this image alone can not ensure that the navigation is correct, so it is necessary to check it with a correct navigation of the original RAW image, so that the errors can be seen, if there are any.

To do this, first an image like the one of Figure 5.2 has been generated, based on the values of the pixels of the original RAW image, that is, according to the illumination of each of them. Thus, as seen in the *Navega_correction.m* algorithm, a minimum value of light has been established (0.10) in order to differentiate the pixels belonging to the illuminated side of the planet of those belonging to the non-illuminated side or to space, so that the correction is carried out taking into account only the pixels belonging to the illuminated side. In this way, an image like the PPM one is obtained, which would be an approximate representation of the correct navigation:

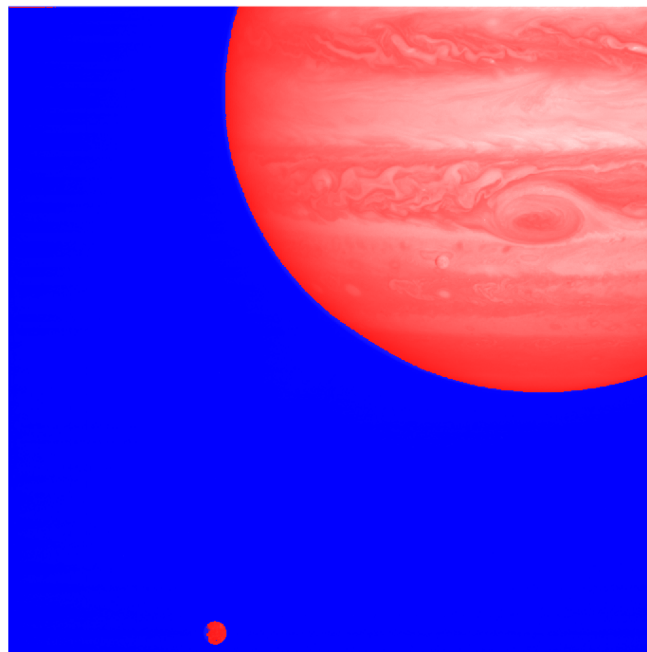

Figure 6.2: Illuminated area from real image

In Figure 6.2, the pixels belonging to the unlit side are blue, so the same in Figure 6.1 are coloured blue in order to satisfy the same conditions.

However, as can be seen, the image obtained presents problems when moons appear, since their lighting is usually similar to that of the planet. So, before proceeding with the navigation error, Figure 6.2 is corrected by the *Moon_correction.m* function, so that the moon does not appear, obtaining the following result:

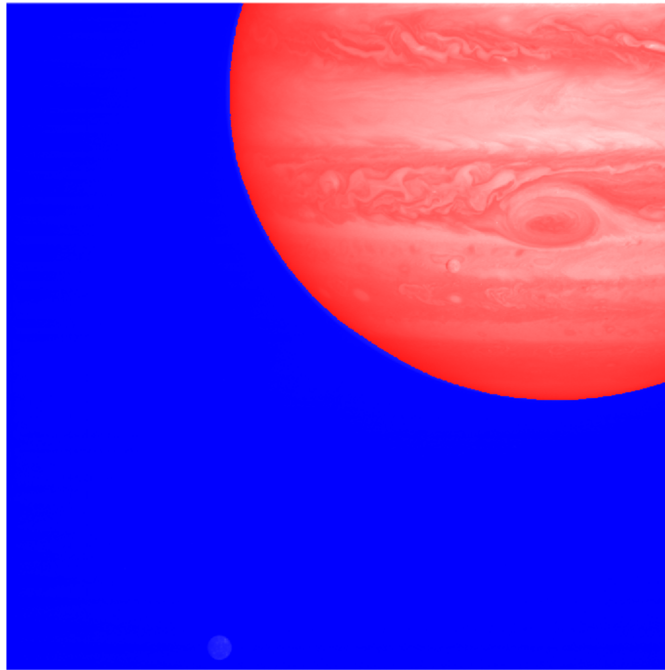

Figure 6.3: Illuminated area from real image + Moon correction

Thus, the combination of Figures 6.1 and 6.3 gives the following result:

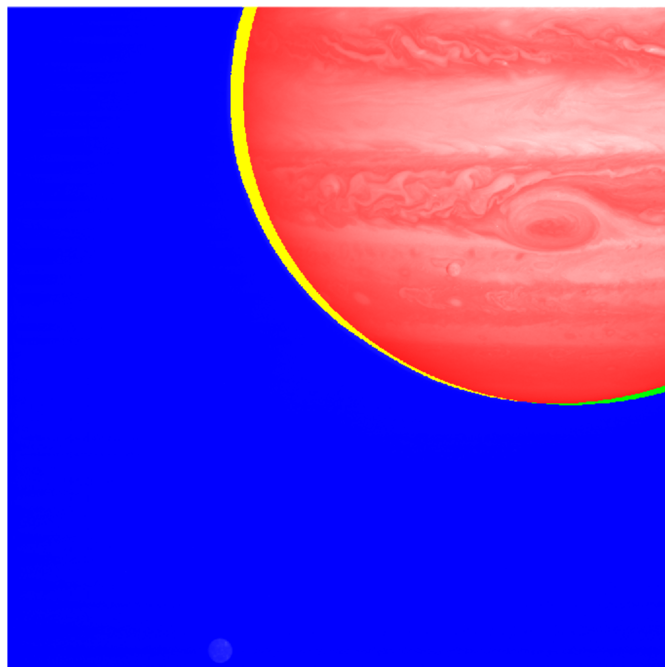

Figure 6.4: Navigation error

The error in pixels is coloured green and yellow, depending on the type of error:

- The green error corresponds to pixels that in the SPICE image are considered part of the illuminated side of the planet, but that actually are part of the unlighted side or space.
- The yellow error corresponds to pixels that are part of the illuminated side of the planet, but that in the SPICE image are considered part of the unlighted side or space.

This differentiation of colors allows to know how the pixels of the SPICE image should be moved, as well as the longitudes and latitudes, during the correction process. In this way, it can be observed that the error is considerable, so it must be corrected in order to obtain the real longitudes and latitudes of each pixel.

As seen in the *Navega_ correction.m* algorithm, the correction process can be carried out by two different methods, so the results must be compared in order to verify its correct functioning. In this way, for each method, the correction process is carried out and the image projection is generated, for a subsequently validation validated using the PLIA software. Below are the SPICE images corrected with both methods:

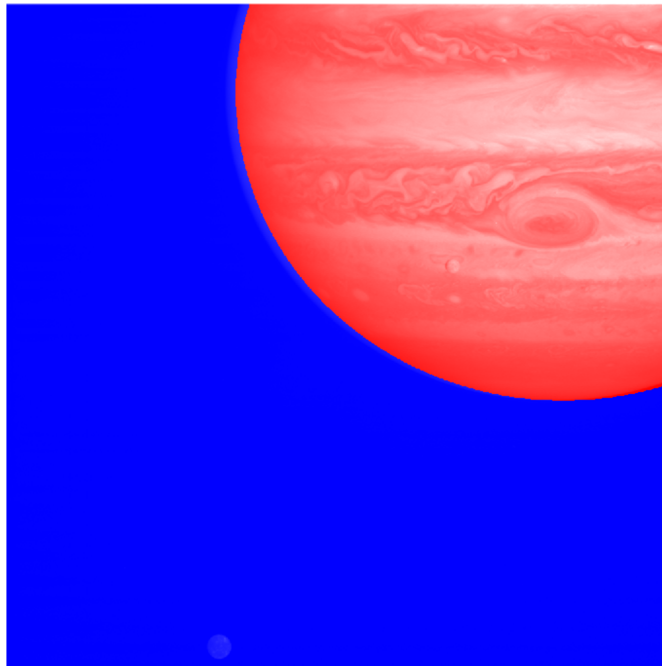

Figure 6.5: Navigation image corrected (Limits method)

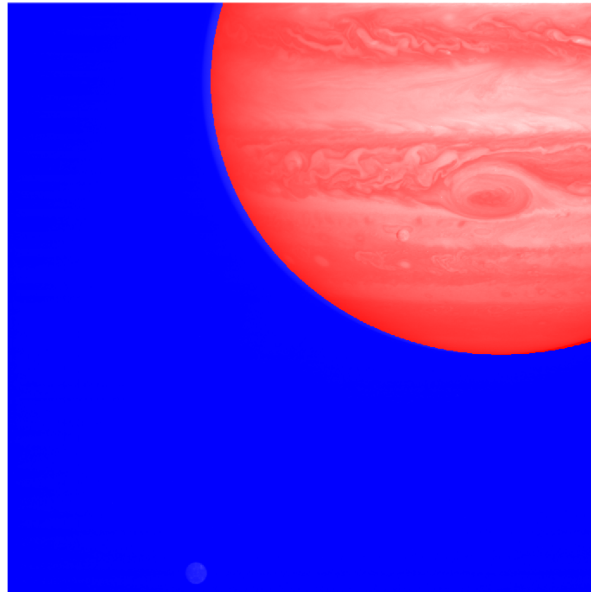

Figure 6.6: Navigation image corrected (Centroid method)

In both methods, the pixels of the navigated image are moved to the left and slightly upwards.

Once the correction process has been completed, the images are combined again as in Figure 6.4, so that the result should not have the green and yellow areas corresponding to the unlighted side and space. The following are the combination images for both methods:

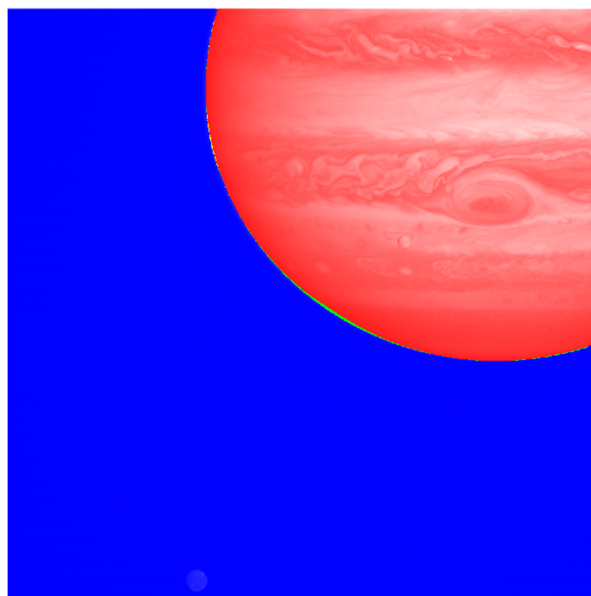

Figure 6.7: Navigation error corrected (Limits method)

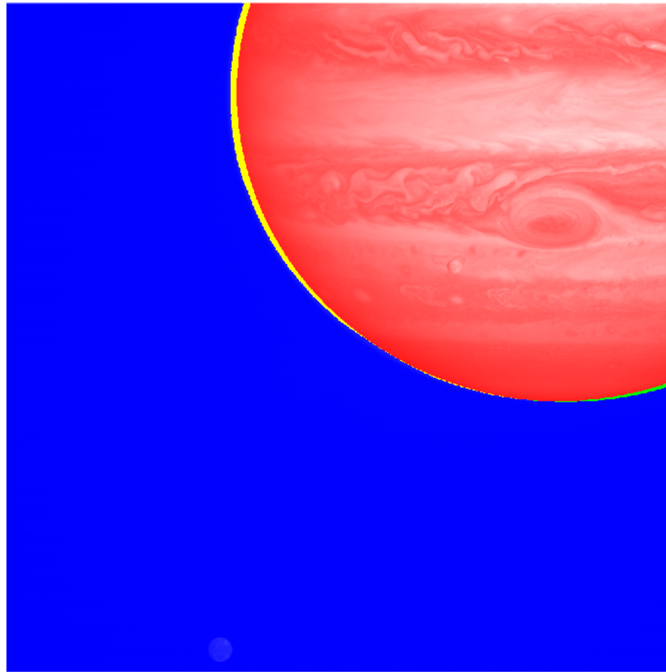

Figure 6.8: Navigation error corrected (Centroid method)

The adjustment of the navigated image seems to be better using the first method. However, the illuminated part of the planet may be larger or smaller, because it is restricted by an imposed minimum value of lighting, so it can only be confirmed by generating the projections and comparing them with those of PLIA.

Therefore, the projections of the *Great Red Spot* generated with each method and generated by the PLIA software are shown below:

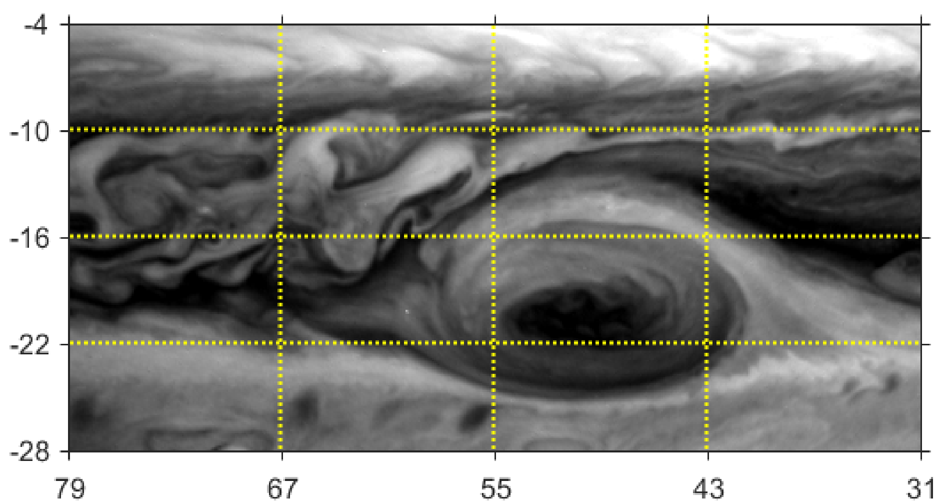

Figure 6.9: Image projection (Limits method)

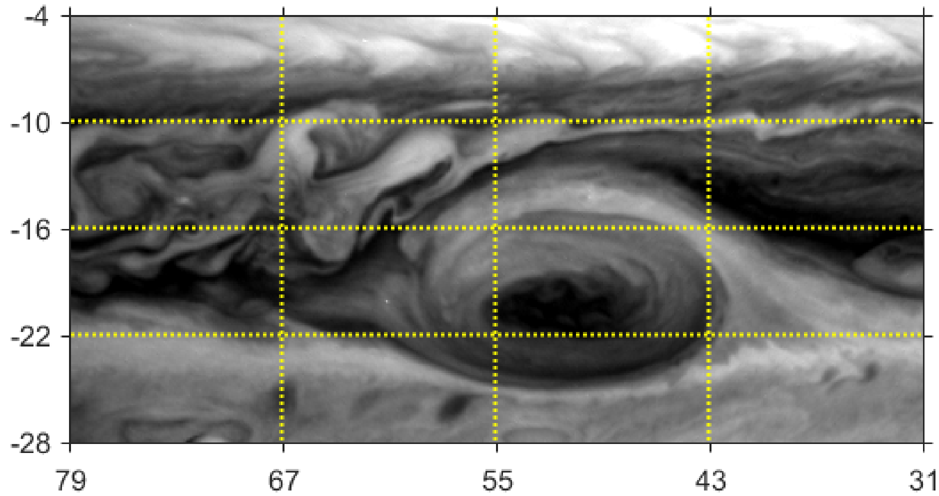

Figure 6.10: Image projection (Centroid method)

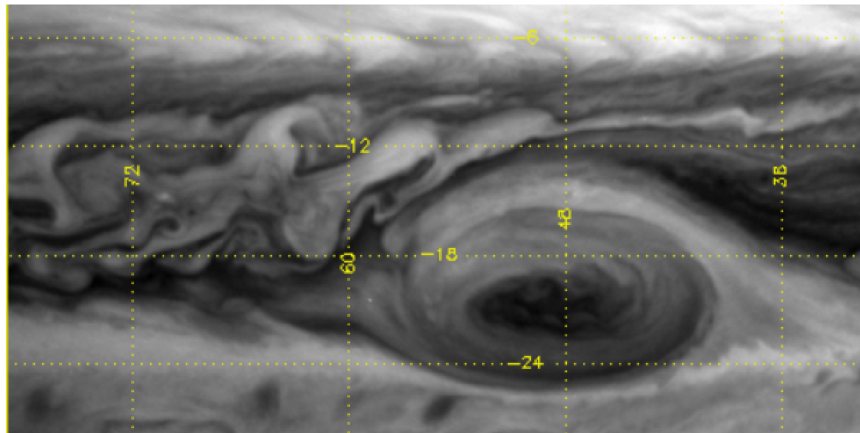

Figure 6.11: Image projection (PLIA)

The first conclusion that can be made is that the projections with the two correction methods are quite similar to the one generated by PLIA, but some deviations are appreciated:

- In the lower part of Figures 6.9 and 6.10, part of a moon that was in transit at the time of the photo is observed. In contrast, in Figure 6.11, this moon does not appear, so it can be assumed that the latitudes of Figures 6.9 and 6.10 are slightly displaced downwards.
- On the right side of Figure 6.11, a small part of what appears to be a cloud is observed. In contrast, in Figure 6.9 and 6.10, the cloud portion is larger (less pronounced in Figure 6.9), so it can be assumed that the longitudes are slightly shifted to the right.

However, the navigation made by the PLIA software is not perfect and requires a manual adjustment. Once this adjustment is made, the following projection is obtained:

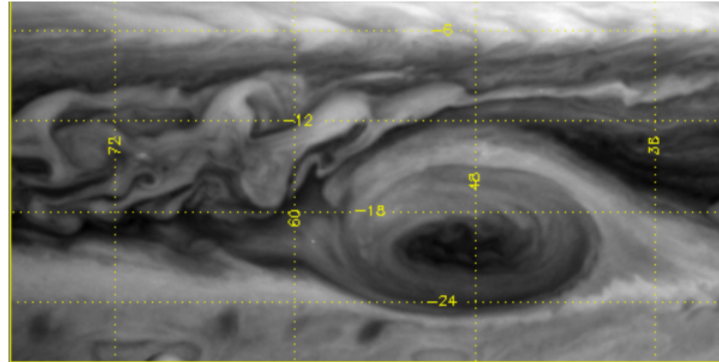

Figure 6.12: Image projection (PLIA)

Unlike Figure 6.11, the moon can now be seen in the lower part of the projection, so that the latitudes of the projections in Figures 6.9 and 6.10 are correct. Even so, the error of the longitudes remains.

Looking at Figure 6.1, it can be seen that the part of the planet corresponding to the unlit side only affects the longitudes (no black area appears on the bottom of the planet), so it is no coincidence that Figures 6.9 and 6.10 only present error due to the longitudes. This fact leads one to think that this error can be corrected by a better delimitation of the unlighted side of the planet, so that, if the longitudes are shifted to the right, it means that the unlit side of the planet should be smaller, so that the minimum value of lighting should be lower.

Applying this reasoning, if the minimum illumination value is reduced from 0.10 to 0.03, the following projections are obtained:

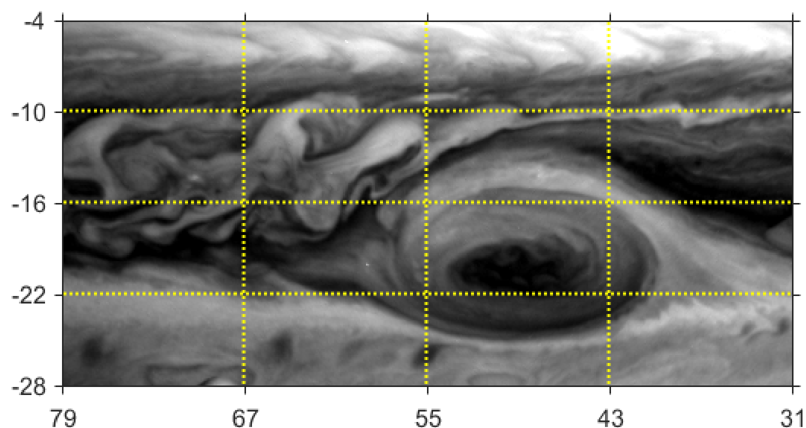

Figure 6.13: Image projection reducing illumination value (Limits method)

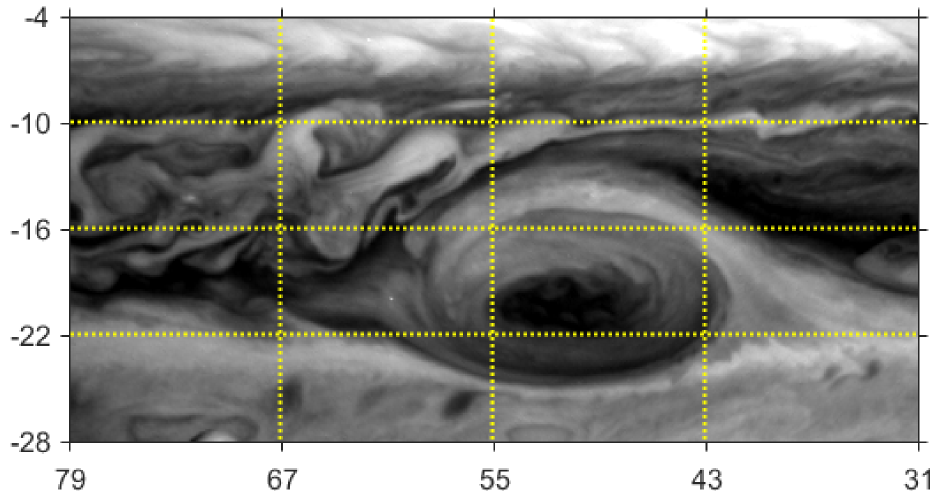

Figure 6.14: Image projection reducing illumination value (Centroid method)

In the second method, the fact of reducing the minimum value of lighting has not served and the same projection has been obtained. However, in the first method, this change has worked and a projection has been obtained practically equal to that of Figure 6.12, which leads to think that the first method is better, as expected.

The fact that it is an image in which the planet is not seen completely, since it is in a corner of the image, if the navigation error occurs towards "outside the image", the illuminated planet area is reduced, so with the second correction method the centroid may not be well calculated.

At the same time, in this image there are only two limits for the first method correction, on the left and below. If the whole planet were to appear in the image, there would be four limits, so the limits with a higher error would be used. In this way, a better adjustment through the limit with no unlit area is expected, because the illuminated planet zone is clearer due to a greater illumination gradient with respect to the space. However, if a later correction is needed, as the longitudes are corrected by the limit with no unlighted planet, the variation of the minimum illumination would not change the projection.

Thus, the entire previous procedure has been done with the following image, in which the entire planet appears:

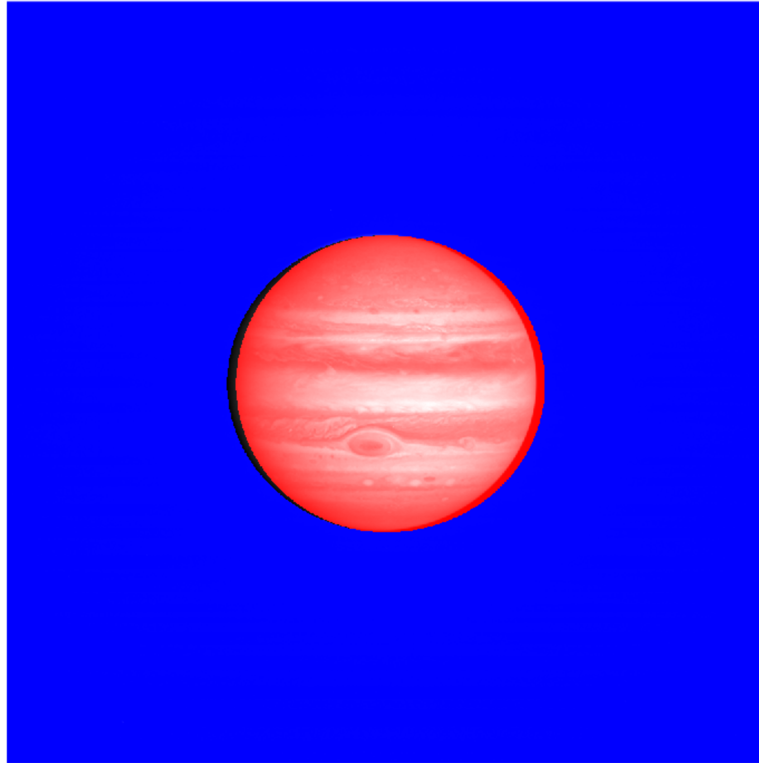

Figure 6.15: Navigated image of entire Jupiter

In this case the error can be seen without having to combine images. Therefore, the projections of the *Great Red Spot* generated with each method and generated by the PLIA software are shown below:

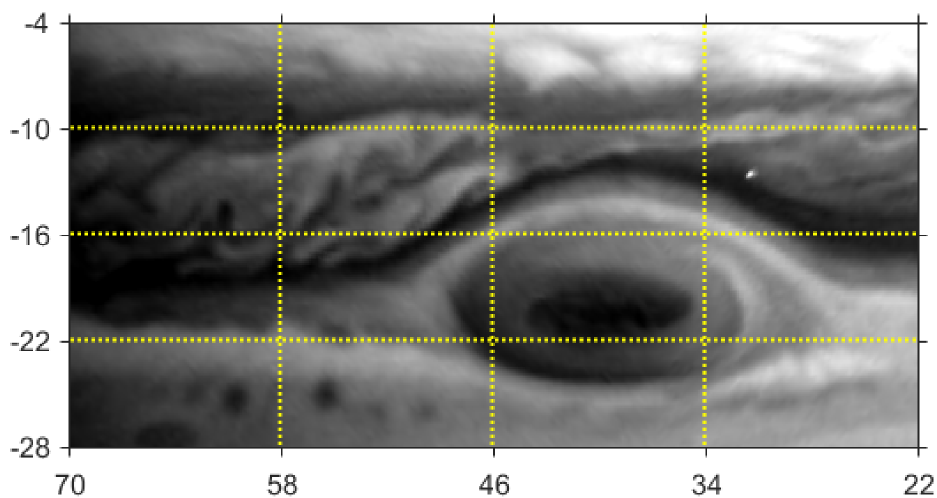

Figure 6.16: Image projection (Limits method)

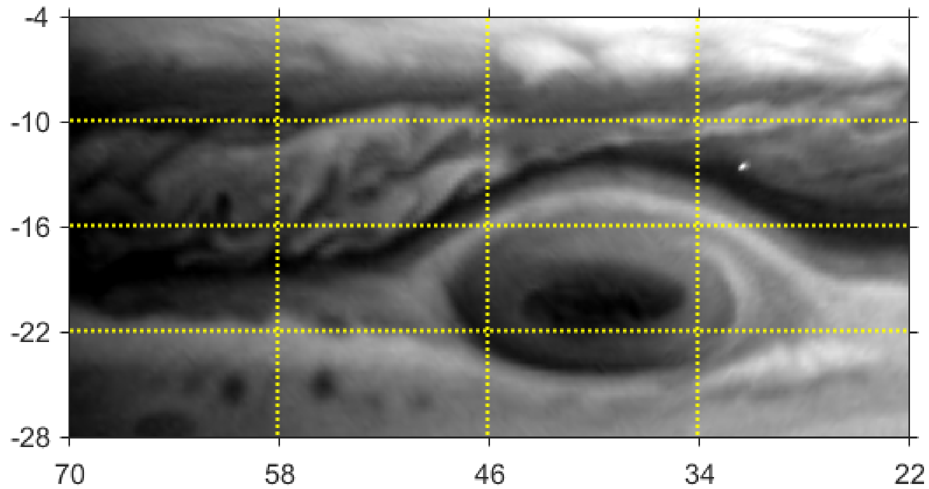

Figure 6.17: Image projection (Centroid method)

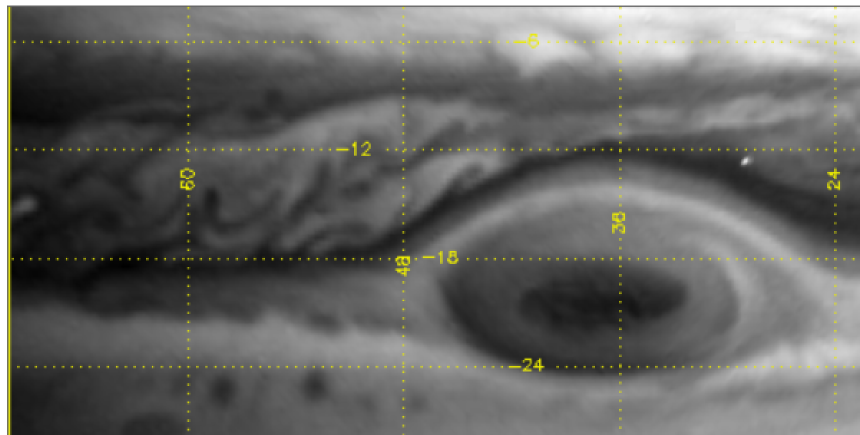

Figure 6.18: Image projection (PLIA)

In this case, the minimum illumination value has been set to 0.03, due to the better result obtained in the previous case. Thus, the projections made with both methods appear to be the same, so both present the same error. This fact already suggests that if the planet is seen completely, both methods are equally correct with a properly minimum illumination value.

Besides, in the first method, the limits of the right and of the bottom have been used to correct the image, since they are the limits in which a greater error appears. Therefore, since the side of the unlighted planet is on the left limit, it is expected that when the minimum value of illumination is changed, the projection with this method will not be affected. However, a better delimitation of the illumination side in the right limit favours an initial better correction.

Looking at Figure 6.18, some spots can be seen in the lower left side of the image. In Figures 6.16 and 6.17, these same spots appear to be displaced to the left, which means that their longitudes are displaced to the right.

However, as in the previous case, the image projected by PLIA must be corrected manually, so that the following projection is obtained:

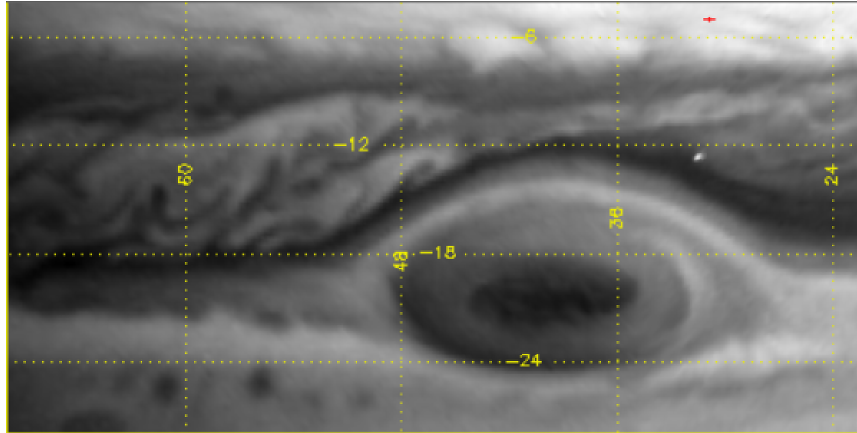

Figure 6.19: Image projection (PLIA)

In this case, both Figures 6.16 and 6.17 are practically equal to the corrected PLIA projection in Figure 6.19, so no other correction is needed.

This reaffirms that the second method correction does not work well in images in which the entire planet does not appear, but that works just as well as the first method in images with the entire planet. Moreover, in the first method, it is preferable to use as limits those with no unlighted side of the planet, due to a better delimitation between the illuminated side of the planet and the space.

A final conclusion that is made from the results presented is that the navigations carried out with the developed program are quite good and closely resemble those generated by specialized software such as PLIA. Additionally, the degree of error of the images is not always the same, that is, in some images the error is considerable, while in others there is almost no error. This fact suggests that the error in the navigations may be due to orientation errors of the CK *kernels* or because the *kernels* loaded in the program are not appropriate.

Finally, the projections can be obtained in RGB format by superposing the projection of the same image taken by each filter. Due to the small errors of the navigations, the projections of a same image may not fit perfectly, but some manually correction can be made if so.

So, below is the projection of *Great Red Spot* of the image of Figure 6.1 in RGB format:

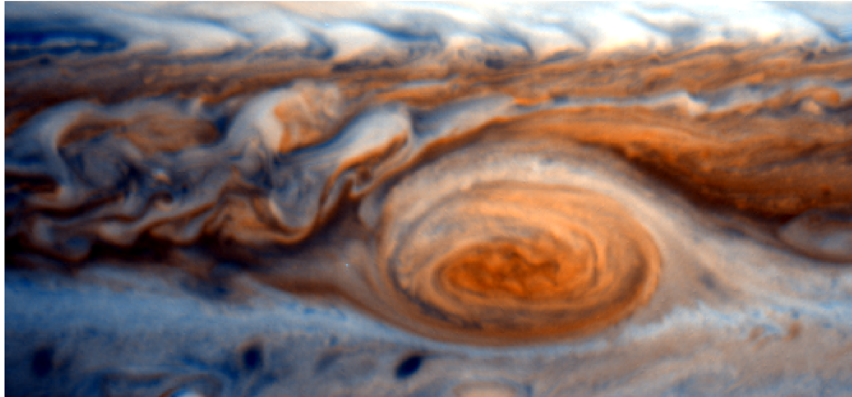

Figure 6.20: Image projection in RGB format

Chapter 7

Environmental impact

In this chapter, the environmental impact that the completion of this project has meant is presented.

As mentioned in the Budget document, the costs related to the author's displacements have not been taken into account, so neither the environmental impact of the same[1].

Thus, the only environmental impact of this project is due to carbon dioxide (CO_2) emissions and radioactive waste generated during the obtaining of the electricity used by the computers.

This environmental impact of the electricity depends on the energy sources used for its generation. According to the National Commission of Markets and Competition (CNMC, for its acronym in Spanish, *Comisión Nacional de los Mercados y la Competencia*) [5], the energy marketed in Spain represents a national average of 0.26 kgCO_2eq/kWh and 0.51 mg/kWh of radioactive waste in 2018, so throughout this project 9.97 kgCO_2eq and 19.56 mg of radioactive waste have been produced.

According to the author's latest electric bill, these values rise to 0.43kgCO_2eq/kWh and 0.76 mg/kWh of radioactive waste, so these values would translate into 16.49 kgCO_2eq and 29.15 mg of radioactive waste.

[1]The displacements have been made through public transport, so they would not have been taken into account as environmental impact.

Chapter 8

Conclusions

At the end of this study, it can be affirmed that the objectives set at the beginning of the project have been achieved. A decoder for reading and converting the RAW images in VICAR format and a program capable of implementing the PLIA algorithms have been developed successfully, obtaining results quite similar to those of specialized software. In addition, a program capable of automatically correcting navigational errors has been developed, which in the PLIA software must be done manually.

As it has been shown, the program has allowed, from images of planets, to obtain the longitudes and latitudes of the pixels in order to know the location of clouds or storms at all times, such as the *Great Red Spot* of Jupiter. Thus, through the different navigations of the *Great Red Spot*, videos can be made with which to know and observe the evolution of the storm.

From here, the next steps would be to refine the developed solver and adapt it to work with more spacecraft, such as Galileo or New Horizons, and try to find alternatives for spacecrafts in which the *kernels* are not good enough to perform navigation, as has happened with Voyager. In this sense, the end of this project has been left ready to enter with a computational approach in the Juno probe, which is presented as a challenge due to its image and transmission systems.

Bibliography

[1] A. M. Juez – Grupo de Ciencias Planetarias, University of the Basque Country UPV/EHU. PVOL – Planetary Virtual Observatory & Laboratory (PPT), 2006. [Online; accessed 5-June-2019]. URL: https://svo.cab.inta-csic.es/docs/files/svo/Public/Meetings/SVO_thematic_network_First_Meeting/PVOL_SVO-06042006.ppt.

[2] A. P. Gasull. *Analysis and study of a shallow water model code for applications to planetary atmospheres*, 2018.

[3] B. Knowles – Cassini Imaging Central Laboratory for Operations (CICLOPS), Space Science Institute. Cassini Imaging Science Subsystem (ISS) Data User's Guide (PDF), 2018. [Online; accessed 5-June-2019]. URL: http://www.ciclops.org/sci/docs/iss_data_user_guide_180916.pdf.

[4] Cassini Imaging Central Laboratory For Operations (CICLOPS). Cassini ISS CALibration (CISSCAL), 2018. [Online; accessed 5-June-2019]. URL: http://ciclops.org/sci/cisscal.php?js=1.

[5] Comisión Nacional de los Mercados y la Competencia (CNMC), 2019. [Online; accessed 5-June-2019]. URL: https://gdo.cnmc.es/CNE/resumenGdo.do?anio=2018.

[6] Deen, Robert G. – Jet Propulsion Laboratory (JPL), California Institute of Technology. *The VICAR file format (PDF)*, 1994. [Online; accessed 5-June-2019]. URL: https://www-mipl.jpl.nasa.gov/external/VICAR_file_fmt.pdf.

[7] E. G. Melendo. Provider of software and installation and help manuals, 2019.

[8] F. Bagenal & R. J. Wilson, LASP – University of Colorado. *Jupiter Coordinate System*, 2016.

[9] Hueso, R., Legarreta, J., Rojas, J.F., Peralta, J., Pérez-Hoyos, S., del Río-Gaztelurrutia, T., Sánchez Lavega, A. *The Planetary Laboratory for Image Analysis*. *Advances in Space Research*, 46(9):1120–1138, 2010. doi:10.1016/j.asr.2010.05.016.

[10] Jet Propulsion Laboratory (JPL), California Institute of Technology. *VICAR User's Guide*, 1994. [Online; accessed 5-June-2019]. URL: https://www-mipl.jpl.nasa.gov/PAG/public/vug/vugfinal.html.

[11] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Planetary Voyage, 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/science/planetary-voyage/.

[12] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Cosmic Ray Subsystem (CRS), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/crs/.

[13] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Imaging Science Subsystem (ISS), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/iss/.

[14] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Infrared Interferometer Spectrometer and Radiometer (IRIS), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/iris/.

[15] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Low-Energy Charged Particles (LECP), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/lecp/.

[16] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Magnetometer (MAG), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/mag/.

[17] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Photopolarimeter Subsystem (PPS), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/pps/.

[18] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Planetary Radio Astronomy (PRA) and Plasma Wave Subsystem (PWS), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/pws/.

[19] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Plasma Science (PLS), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/pls/.

[20] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft – Ultraviolet Spectrometer (UVS), 2019. [Online; accessed 5-June-2019]. URL: https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/uvs/.

[21] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – Spacecraft Instruments, 2019. [Online; accessed 5-June-2019]. URL: `https://voyager.jpl.nasa.gov/mission/spacecraft/instruments/`.

[22] Jet Propulsion Laboratory (JPL), California Institute of Technology. Voyager – The Spacecraft, 2019. [Online; accessed 5-June-2019]. URL: `https://voyager.jpl.nasa.gov/mission/spacecraft/`.

[23] M. R. Showalter et al. *How to Obtain Cassini Data via NASA'S Planetary Data System (PDF)*, 2019. [Online; accessed 5-June-2019]. URL: `https://pds-imaging.jpl.nasa.gov/help/How_to_obtain_Cassini_data.pdf`.

[24] MathWorks. *Buscar rotación y escala de imágenes mediante la combinación de funciones automáticas*, 2019. [Online; accessed 5-June-2019]. URL: `https://es.mathworks.com/help/images/examples/find-image-rotation-and-scale-using-automated-feature-matching.html`.

[25] NASA Solar System Exploration. Cassini Plasma Spectrometer (CAPS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: `https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/cassini-plasma-spectrometer/`.

[26] NASA Solar System Exploration. Composite Infrared Spectrometer (CIRS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: `https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/composite-infrared-spectrometer/`.

[27] NASA Solar System Exploration. Cosmic Dust Analyzer (CDA) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: `https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/cosmic-dust-analyzer/`.

[28] NASA Solar System Exploration. Imaging Sciecne Subsystem (ISS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: `https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/imaging-science-subsystem/`.

[29] NASA Solar System Exploration. Ion and Neutral Mass Spectrometer (INMS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: `https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/ion-and-neutral-mass-spectrometer/`.

[30] NASA Solar System Exploration. Magnetometer (MAG) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: `https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/magnetometer/`.

[31] NASA Solar System Exploration. Magnetospheric Imaging Instrument (MIMI) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/magnetospheric-imaging-instrument/.

[32] NASA Solar System Exploration. RADAR | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/radio-detection-and-ranging/.

[33] NASA Solar System Exploration. Radio and Plasma Wave Science (RPWS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/radio-and-plasma-wave-science/.

[34] NASA Solar System Exploration. Radio Science Subsystem (RSS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/radio-science-subsystem/.

[35] NASA Solar System Exploration. Ultraviolet Imaging Spectrograph (UVIS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/ultraviolet-imaging-spectrograph/.

[36] NASA Solar System Exploration. Visible and Infrared Mapping Spectrometer (VIMS) | Cassini Orbiter, 2018. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/visible-and-infrared-mapping-spectrometer/.

[37] NASA Solar System Exploration. Cassini Orbiter | Spacecraft, 2019. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/mission/spacecraft/cassini-orbiter/.

[38] NASA Solar System Exploration. Diagrama of the Cassini Spacecraft, 2019. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/resources/12943/diagram-of-the-cassini-spacecraft/.

[39] NASA Solar System Exploration. Overview | Cassini, 2019. [Online; accessed 5-June-2019]. URL: https://solarsystem.nasa.gov/missions/cassini/overview/.

[40] PDS Cartography and Imaging Sciences Node, JPL, NASA. Cassini–Huygens, 2018. [Online; accessed 5-June-2019]. URL: https://pds-imaging.jpl.nasa.gov/portal/cassini_mission.html.

[41] PDS Cartography and Imaging Sciences Node, JPL, NASA. *Introduction to the Cassini Imaging Science Subsystem: Narrow Angle Camera*, 2018. [Online; accessed 5-June-2019]. URL: `https://pds-imaging.jpl.nasa.gov/data/cassini/cassini_orbiter/coiss_2101/catalog/issna_inst.cat`.

[42] PDS Cartography and Imaging Sciences Node, JPL, NASA. *Introduction to the Cassini Imaging Science Subsystem: Wide Angle Camera*, 2018. [Online; accessed 5-June-2019]. URL: `https://pds-imaging.jpl.nasa.gov/data/cassini/cassini_orbiter/coiss_2101/catalog/isswa_inst.cat`.

[43] PDS Cartography and Imaging Sciences Node, JPL, NASA, 2019. [Online; accessed 5-June-2019]. URL: `https://pds-imaging.jpl.nasa.gov/`.

[44] PDS Ring-Moon System Node, JPL, NASA. *CISSCAL User Guide (PDF)*, 2009. [Online; accessed 5-June-2019]. URL: `https://pds-rings.seti.org/cassini/iss/cisscal_manual.pdf`.

[45] PDS: The Planetary Data System, NASA, 2019. [Online; accessed 5-June-2019]. URL: `https://pds.nasa.gov/`.

[46] PLIA: The Planetary Laboratory for Image Analysis, 2010. [Online; accessed 5-June-2019]. URL: `http://www.ajax.ehu.es/PLIA/`.

[47] PVOL – Planetary Virtual Observatory and Laboratory, 2019. [Online; accessed 5-June-2019]. URL: `http://pvol2.ehu.eus/pvol2/`.

[48] R. Hueso et al. – University of the Basque Country UPV/EHU. *The Planetary Virtual Observatory and Laboratory (PVOL) and its integration into the Virtual European Solar and Planetary Access (VESPA) (PDF)*, 2017. [Online; accessed 5-June-2019]. URL: `https://arxiv.org/ftp/arxiv/papers/1701/1701.01977.pdf`.

[49] The Navigation and Ancillary Information Facility (NAIF), JPL, NASA. Illumin SPICE function, 2017. [Online; accessed 5-June-2019]. URL: `https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/ilumin_c.html`.

[50] The Navigation and Ancillary Information Facility (NAIF), JPL, NASA. *An Overview of Reference Frames and Coordinate Systems in the SPICE Context (PDF)*, 2019. [Online; accessed 5-June-2019]. URL: `https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/17_frames_and_coordinate_systems.pdf`.

[51] The Navigation and Ancillary Information Facility (NAIF), JPL, NASA. SPICE Concept, 2019. [Online; accessed 5-June-2019]. URL: `https://naif.jpl.nasa.gov/naif/spiceconcept.html`.

[52] Trigo-Rodriguez, J.M., Sánchez-Lavega, A., Gómez, J.M., Lecacheux, J., Colas, F., Miyazaki, I. The 90-day oscillations of Jupiter's Great Red Spot revisited. *Planetary and Space Science*, 48(4):331–339, 2000. doi:10.1016/s0032-0633(00)00002-7.

[53] Wikipedia contributors. Atmospheric science — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 5-June-2019]. URL: https://en.wikipedia.org/w/index.php?title=Atmospheric_science&oldid=890789242.

[54] Wikipedia contributors. Cassini–Huygens — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 5-June-2019]. URL: https://en.wikipedia.org/w/index.php?title=Cassini%E2%80%93Huygens&oldid=899344218.

[55] Wikipedia contributors. Netpbm format — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 5-June-2019]. URL: https://en.wikipedia.org/w/index.php?title=Netpbm_format&oldid=883066379.

[56] Wikipedia contributors. Terminator (solar) — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 5-June-2019]. URL: https://en.wikipedia.org/w/index.php?title=Terminator_(solar)&oldid=879256178.

[57] Wikipedia contributors. Voyager program — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 5-June-2019]. URL: https://en.wikipedia.org/w/index.php?title=Voyager_program&oldid=900124465.

[58] Wikiwand. Latitude — Wikiwand, 2019. [Online; accessed 5-June-2019]. URL: https://www.wikiwand.com/en/Latitude.