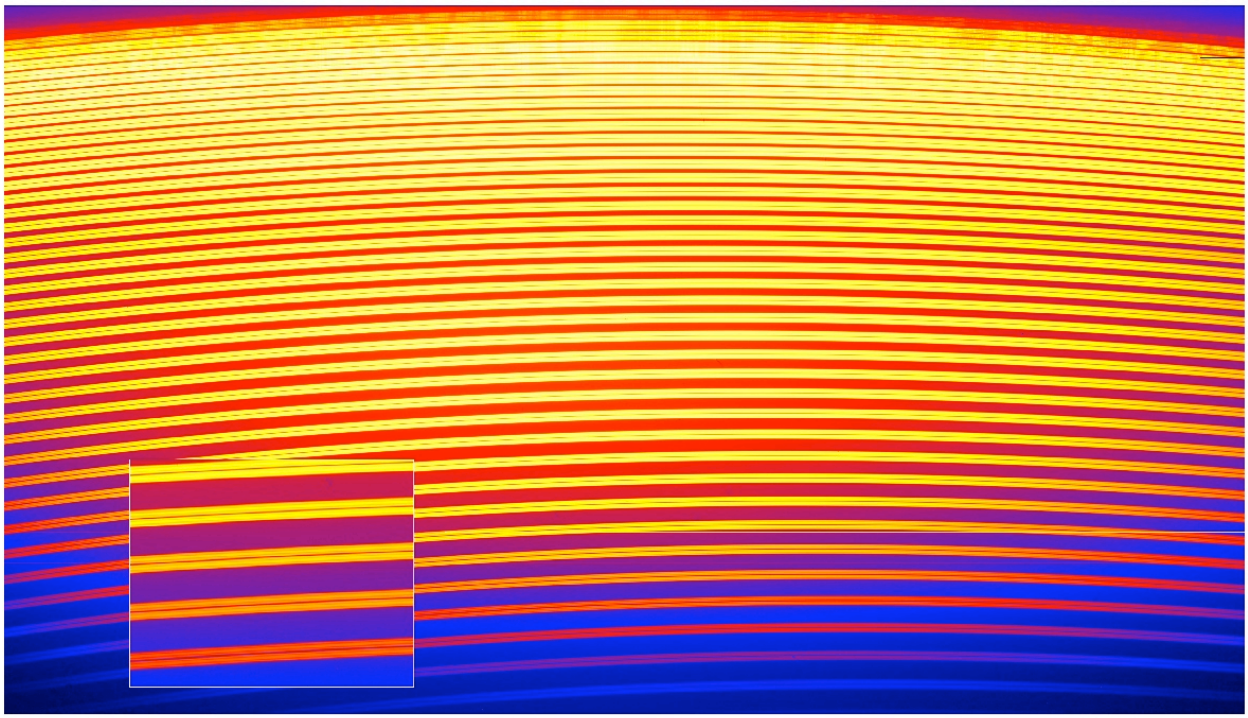
Canada-France-Hawaii Telescope Corporation
65-1238 Mamalahoa Hwy, Kamuela, Hawaii 96743 USA

Société du Télescope Canada-France-Hawaii
Telephone (808) 885-7944 Fax (808) 885 -7288

OPERA PIPELINE PROJECT



DOUG TEEPLE / EDER MARTIOLI
Canada France Hawaii Telescope
Waimea, HI October 2012

Contents

1. Introduction

OPERA (Open-source Pipeline for Espadons Reduction and Analysis) is a Canada France Hawaii Telescope (CFHT) open source collaborative software project currently under development for an ESPaDOnS echelle spectro-polarimetric image reduction pipeline.

There have been a number of echelle spectrum pipelines developed in the past (e.g. REDUCE Package2). Most of them have been designed for a particular instrument or data format, and therefore do not provide an interface that is particularly flexible or maintainable. OPERA, by design, is flexible, while at the same time supporting the demands of a production environment.

The notion of a simple “pipeline” is somewhat misleading in the context of OPERA.

Firstly, pipeline has the connotation of a single linear flow of data in this case from one end of the pipeline to the final product at the other end. This is not at all how OPERA works. OPERA is a scaleable, multi-server, parallel execution, dependency driven framework. Every opportunity to improve throughput is taken. While it can execute on a desktop for a single user, it scales to the needs of an operational telescope, taking hundreds of images a night and delivering product the next morning. That said, for convenience, we will continue to use the term *pipeline* loosely in this document.

Secondly, while OPERA does execute as a pipeline it is also a toolkit for developing pipelines. It has been written in such a way that it may be adapted to new instruments relatively easily. It is completely transparent in that the entire source code is available and the source, algorithms, data structures and data formats are documented.

Thirdly, OPERA is a “production pipeline”. What us meant by “production”? It means in this context that the execution of the pipeline is completely automated and autonomous. No human intervention is required to execute the pipeline on a daily basis. The days when a resource could be dedicated to coddling the reduction are long gone. Fast, autonomous and automated are the keys now.

OPERA iperform calibrations and reduction, producing one-dimensional intensity and polarimetric spectra. The calibrations are performed on two-dimensional images. Spectra are extracted using an optimal extraction algorithm. While primarily designed for CFHT ESPaDOnS data, the pipeline is being written to be extensible to other echelle spectrographs. A primary design goal is to make use of fast, modern object-oriented technologies. Processing is controlled by a harness, which manages a set of processing modules, that make use of a collection of native OPERA software libraries and standard external software libraries. The harness and modules are completely parametrized by site configuration and instrument parameters.

The software is written in a modern object-oriented high level language with abstraction facilities such as operator overloading and hierarchical classes that permit users wishing to extend opera to do so at a high level of abstraction, concentrating on the science and not the details of address arithmetic and memory management. All these features have been designed to provide a portable infrastructure that facilitates collaborative development, code re-usability and extensibility. OPERA is free software with support for both GNU/Linux and MacOSX platforms.

At CFHT OPERA will be used for ESPaDOnS data reduction and replace the existing Upena pipeline based on J.F.Donati’s Libre Esprit software. CFHT is hosting the project and in order to facilitate the exchange of infor-

mation between developers, we have created a project called opera-pipeline on SourceForge hosting the Opera source code and documentation. All source code and documentation is maintained on SourceForge at this URL:

http://sourceforge.net/projects/opera-pipeline/

This document describes modules in the CFHT Core Pipeline: - operaGain, operaReductionSet, operaGeometry, operaOptimalExtraction for star-only mode and operaWavelengthCalibration, extraction aperture calibration, wavelength calibration, normalization, polarimetry, operaTelluricWavelengthCalibration, and operaHeliocentricWavelengthCalibration. The design will also briefly touch on new major classes that have been developed to support these modules. This document also introduces a number of software libraries including functions anticipated to be included each library. Each function's inputs and outputs are given. The algorithm used in the modules or library is given in English. The major data structures, in this case classes, of opera are also given.

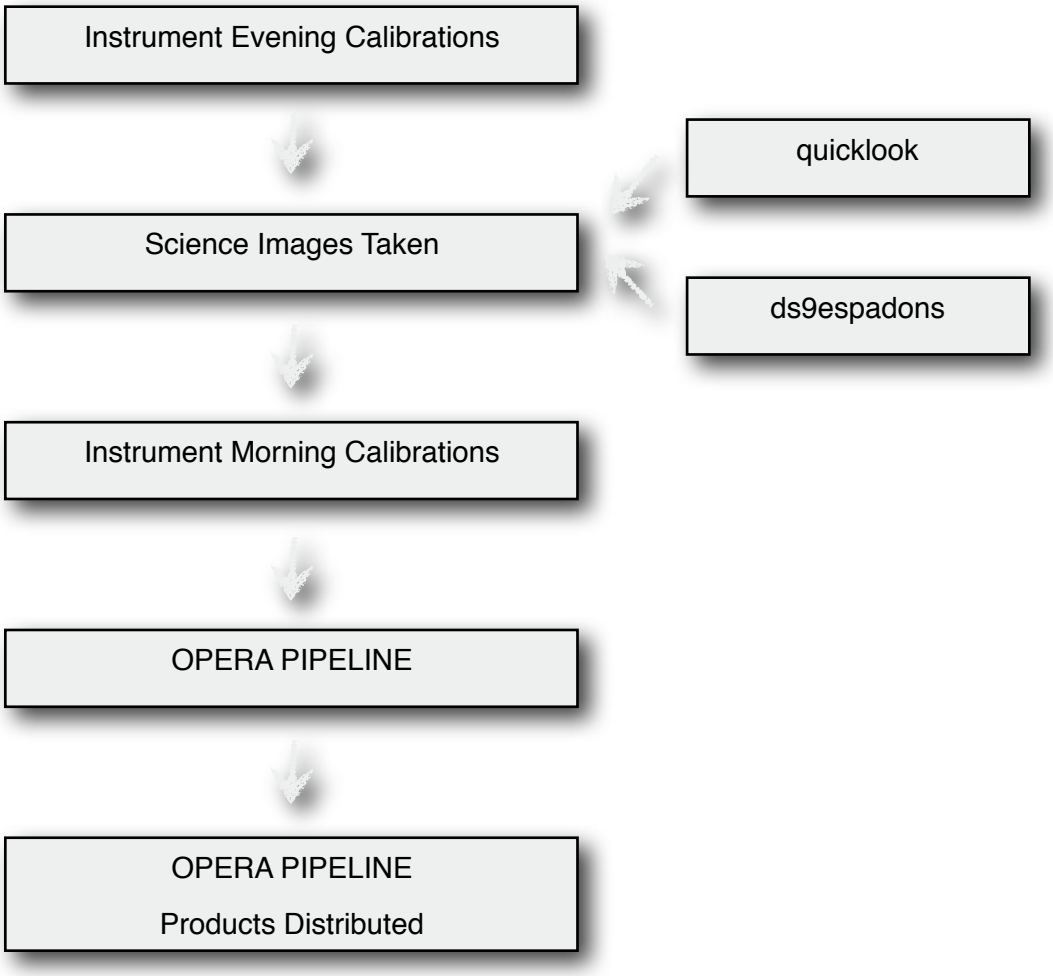
In Section 2 we review the overall data flow of OPERA. Section 3 is an introduction to Extraction, one of the core concepts in echelle spectrograph image reduction. Section 4 introduces the overall architecture of OPERA. Section 5 describes the core modules. Section 6 discusses OPERA libraries. Section 7 discusses major classes and data structures which comprise OPERA. Section 8 shows preliminary results from the prototype implementation. Section 9 concludes this document. Appendix A gives examples of the file formats used to transfer information between modules. Appendix B lists terms used throughout this document. Appendix C gives references to papers used in OPERA algorithms.

Readers should refer to prior OPERA documents in order to understand the level of detail presented here. These documents are: *OPERA Conceptual Design,* which introduces the module, harness and library concepts, *OPERA Scope*, which determines what the Core OPERA will perform and not perform as a pipeline for spectroscopy, the *OPERA Operational Technical and Scientific Requirements* document, which outlines what is required of the pipeline and the *OPERA Preliminary Design*, which outlines the preliminary software design of OPERA.

2. Espadons Overall Data Flow

2.1 Conceptual Data Flow

This diagram shows where OPERA fits in the overall observing process from taking the raw data up to distributing the reduced data to PIs:



quicklook provides SNR values for observers during the night, ds9espadons is an automated image viewer. The OPERA pipeline supports both quicklook and full reductions.

2.2 Processing

2.2.1 Image Modes

OPERA-core takes as input ESPaDOnS raw images, taken in any of the 3 Observing Modes (Polarimetry, Star+Sky, Star only), taken in any of the 3 CCD Readout Modes (Fast, Normal, Slow), and taken in any of the detector modes (EEV1e, EEV1, Olapa in 1-amp mode, Olapa in 2-amp mode).

2.2.2 Calibration

OPERA-core makes use of the following minimum set of exposures:

- Calibration: 1 bias , 5 flat-fields, 1 align (Fabry-Perot), 1 comparison (Th-Ar).
- • Science: 1 exposure for Star+Sky and Star-only modes or 4 consecutive exposures alternating the rotation position of the two Fresnel-Rhomb for Polarimetry mode.

2.2.3 OPERA Products Format

OPERA-core provides reduced data as OPERA-core products in a standard format suitable for analysis by Principal Investigators (PIs): standard FITS format and ascii tables. The OPERA-core products for each respective observing mode contains the following information per spectral bin.

- Star only: wavelength (autowave on/off), flux (normalization on/off), and errors.
- Star+Sky: wavelength (autowave on/off), star flux (normalization on/off), star+sky flux (normalization on/off), sky flux (normalization on/off), and errors.

- Polarimetry: wavelengths (autowave on/off), Stokes I (intensity) (normalization on/off), Stokes Q, U or V (polarimetric normalization on/off), check N1 and N2 spectra (as defined in Donati et al. 19971), and errors.

2.2.4 Removal of Detector Signature

OPERA-core reduction detects and corrects the following CCD detector effects:

- Bias is corrected up to the level of the intrinsic noise of the detector in the readout mode used.
- Bad pixels, hot pixels, saturated pixels, and cosmic rays are identified and masked.
- Pixel-to-pixel sensitivity variations are corrected for variations with amplitude lower than the signal provided by the combined flat-field exposures at the corresponding area of the CCD, and greater than the detector noise.
- Fringing is corrected if its amplitude is greater than the uncertainty in the flux for combined flat-field calibration exposures.

2.2.5 Removal of the Spectrograph Signatures

OPERA-core reduction detects and corrects the following instrumental signatures specific to ESPaDOnS:

- Order curvatures are traced for all extracted orders to an RMS precision better than 0.1 pixel.
- Pseudo-slit shape is measured for all extracted orders to an RMS precision better than 0.1 pixel.

- Pixel-to-wavelength calibration. All orders are calibrated with an average accuracy better than 150 m/s. The wavelength calibration uses a Th-Ar lamp exposure as comparison and provides the option to perform an additional correction (autowave correction) using telluric spectral lines identified in the science exposures.

- Vignetting, instrumental response (sensitivity to different wavelengths), blaze function, optical fiber transmission, etc. OPERA-core uses an archival solar spectra taken from an exposure of the Moon to perform flux calibration with accuracy determined by the archival solar spectra. OPERA-core is not required to account for long-term variations in the instrument effects mentioned above.

2.2.6 Extraction

OPERA-core extracts two-dimensional data and reduces it into one-dimensional spectra from at least 40 spectral orders. OPERA-core offers the optimal extraction algorithm by Horne, K. (1986)[3] and further revised by Marsh, T.R. (1989).[4]

2.2.7 Errors

OPERA-core provide statistical errors for object intensity, sky and Stokes parameters provided in the data products. Errors are propagated through the reduction steps.

2.2.8 Continuum Normalization

OPERA-core normalizes the spectra by the continuum. The OPERA-core normalization algorithm is robust for normalization of the stellar photospheric continuum emission, which may contain a limited number of emission lines, and where molecular band absorption features are not predominant. Objects with either predominant emission lines or molecular wide

band absorption features may not be normalized properly by OPERA-core. However these types of objects may be supported in future versions of OPERA. Even though the normalization does not work properly for some objects, the OPERA-core normalization module must not fail when reducing data from these objects.

2.2.9 Signal-to-Noise Ratio

OPERA-core calculates and retains as a byproduct the signal-to-noise ratio (SNR) for all orders.

2.2.10 Spectral Resolution

OPERA-core calculates and retains as a byproduct the spectral resolution for all orders.

2.3 Operational Considerations

2.3.1 Platforms

The OPERA Core Reduction modules OPERA executes at base level on existing CFHT server hardware. It executes on 32 or 64 bit hardware running GNU/Linux or MacOSX.

2.3.2 Reduction Speed

OPERA Core Reduction modules should a worst case set of 200 calibration images and 250 observational images (one night in the case of CFHT reductions) within 5 hours of completion of a night data in order to meet CFHT commitments.

2.3.3 Instruments

OPERA Core Reduction reduces data taken from the ESPaDOnS EEV1 and OLAPA devices in either one or two amplifier device modes. OPERA

handles device mode changes within a given night. OPERA associates calibrations with science data taken in the same instrument device mode. Parameterization of device-specific characteristics is localized to a parameter table.. Interfaces to data is always done through a parameter access layer for reduction parameters and a data access layer for image data which may resolve to database or other data sources. The harness itself is configurable so that reduction modules can be added or inserted reasonably easily.

2.3.4 Observational Data and Calibrations

The OPERA Core Reduction unit is a set of related calibration and observational data ordered by time. Calibration data must be available in order for the pipeline to operate and the calibration and observational data must correlate within a time span and a given mode, read-out speed and detector.

2.3.5 Reliability and Availability

The OPERA Core Reduction exhibits availability of 95% for the needs of daily reductions at CFHT, in order for CFHT to meet its commitments to the astronomy community.

2.3.6 Autonomous Operation

The OPERA Core Reduction operates autonomously, with no human intervention to complete calibrations and reductions.

2.4 Technical Considerations

2.4.1 Programming and System

The OPERA Core Reduction modules should compile and run on any platform with a recent version of the GNU tool chain (GCC, Make, etc). The

development project targets 32 and 64 bit GNU/Linux distributions and MacOSX 10.6 and above and does not guarantee compatibility with other platforms. All software libraries used are open source libraries and freely available. OPERA modules do not require that users purchase licensed software in order to use OPERA.

2.4.1 OPERA Harness

The Harness is a separate piece of software which controls and directs the execution of reduction modules. The Harness and Core Reduction modules support parallel module execution and execution on multiple machines. The harness is command-line driven. It is abort-able, and when restarted, will not unnecessarily repeat processing steps. The harness is responsible for optimal scheduling and marshaling parameters, including processing parameters, file paths and file names, etc, to be sent to the modules.

2.3.3 Parameterization

OPERA pipeline execution is parameterized, based on instrument, detector and other characteristics. The parameters are stored in a data table. Access to parameters is through a parameter access layer provided as a software library or from the harness.

2.4.4 Software Modules and Software Libraries

Software modules take all inputs and the names of all outputs as command line arguments. In addition, the harness passes, as standard arguments, a location to store temporary byproducts. Modules do not arbitrarily store temporaries in hard-coded directories or filename. Nothing in any module precludes multiple instances of a module executing simultaneously. Common software libraries are used by modules where possible.

.2.4.5 External Dependencies

External dependencies are kept to a minimum to ease portability. The OPERA Core Reduction modules do link to proprietary software libraries or make use of proprietary data.

3. The Extraction of Spectra

3.1 Introduction

We will take a brief diversion to discuss a central concept associated with reduction of echelle spectro-polarimetric data - that of extraction.

Extraction of spectra consists of obtaining energy flux information for each wavelength element from a two-dimensional echellogram spectral image. OPERA adopts the fundamental ideas of the Optimal Extraction algorithm introduced by Horne (1986, PASP 98, 609) and the improved approach for highly curved orders by Marsh (1989, PASP 101, 1032). In addition to these, OPERA is also designed to perform extraction using a two-dimensional instrument illumination profile. The latter is a necessary improvement for irregular profiles, typically found in spectrographs that use a slicer to produce the pseudo-slit, as in ESPaDOnS.

Throughout this document there will be references to file names produced by modules in the pipeline using the notation **.<extension>*, where <extension is one of *prof*, *wcal*, *geom*, meaning Instrument Profile, Wavelength Calibration and Geometry Calibration data. The * stands for a unique, detector, amplifier, mode and speed combination. In general these files contain polynomial coefficients for each order and other useful information associated with the particular calibration step. These files are self-identifying. Examples are given in Appendix A.

3.2 Approach to Extraction

The goal of extraction is to obtain one dimensional (1D) intensity spectra from two dimensional (2D) images. However, before the actual extraction is done, a few preliminary calibration processing steps are required. These steps use calibration images such as master bias, flat-field, and compari-

son lamp (also Fabry-Perot if available) images to characterize the illumination pattern on the detector to optimally extract the object spectra.

The calibration for extraction covered in this document is divided into three major steps: geometry, instrument profile, and wavelength.

Each of these steps corresponds to an OPERA module, which produces an associated calibration product. The products are files containing the measured model and statistics for each calibration. These files can be loaded and used by later steps in the pipeline.

The detailed algorithms for the calibration steps above are discussed in the following sections.

3.2.1 Overview of Calibrations Required for Extraction

- Create a bad-pixel mask
- Create a pixel-sensitivity map (normalized flat)
- Call `operaGeometryCalibration`. This routine populates the `operaGeometry` class with photo-center points obtained from a master flat-field image. Then it fits an optimal polynomial model to the data and creates the calibration product *.geom.
- Call `operaInstrumentProfileCalibration`. This routine populates the `operaInstrumentProfile` class with instrument profile data measured from either a master flat-field, comparison or Fabry-Perot image. Then it fits an optimal polynomial model to the data and creates the calibration product *.prof.
- Refine the geometry model by using the new instrument profile measurements. This will update the geometry calibration file *.geom.

- Refine the instrument profile model using new geometry calibration. This will update the geometry calibration file *.prof.
- Call `operaWavelengthCalibration`. This routine uses the `operaWavelength` class to store a vector of wavelengths (nm) and distances (in pixel units) for a set of spectral lines identified from the master comparison image. Then it fits an optimal polynomial model to the data and creates the calibration file *.wave.

The calibration processing steps described above make use of specific algorithms to perform the required measurements from images. These algorithms are discussed in more detail in the following sections.

Model refinement

Note that geometry provides the photo-center positions which are taken as references by operaInstrumentProfile in order to measure the Instrument Profile (IP). However, the IP determines the illumination and therefore the more accurate photo-center position. This implies that once the IP is measured, the geometry calibration can be refined, which can be used to refine the IP as well.

In wavelength calibration there will be a similar situation, where once one finds a set of spectral lines that matches the lines found in the atlas, the better solution can be used to refine the search for lines which will include more matchings and so on.

The refinement of the calibration models is obtained by adding new information from new measurements. opera implements three different ways of refining the calibration models:

Reset Model: delete previous model and obtain a completely independent new model from new data.

Chi-square/Bayesian Reset: obtain a completely independent new model from a fit to the new data set. Then reset the old model only if the new chi-square is closer to unit. Analogously, in a bayesian fashion, the new model is accepted only if the joint posterior probability is greater than the previous one.

Bayesian Combine (NewData+OldModel => NewModel): consider the previous model (OldModel) as prior information with uncertainty. Consider new measurements (NewData) as random variables. Then find set of parameters for new model that maximizes the likelihood function. The likelihood function is given by the probability of obtaining NewData, if NewModel and prior information (OldModel) are true.

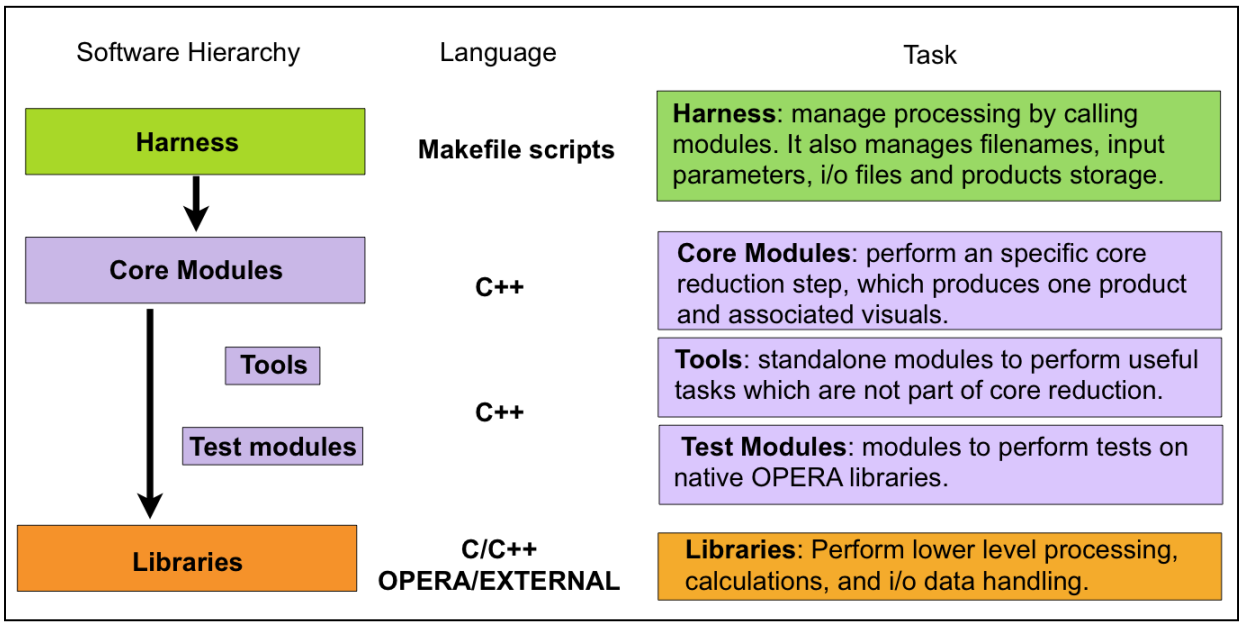
4. OPERA Architecture

4.1 Introduction

4.1.1 What is a Reduction Pipeline?

The reduction of ESPaDOnS data consists of processing steps to convert the raw CCD images of cross- dispersed echelle spectra obtained by the spectrograph into calibrated physical quantities. In Section 4 we discuss each of these steps in more detail. The processing involves, among other things, reading and writing header and pixel data from FITS images, performing statistical analysis of the data, fitting functions, logging the processing steps, producing visualization outputs, and storing the products and by-products. The processing can become quite complex and difficult to manage due to the number of operations involved, particularly as the pipeline is meant to run autonomously without human intervention.

The overall software architecture looks like this:



The pipeline is set up in a hierarchical system, with three main tiers: a harness, core modules, and libraries. The harness at the top of the hierarchy controls all processing by calling modules. The modules then make use of the libraries of methods and external software tools to perform specific reduction tasks.

4.1.2 The Harness or Driver

The harness is required to control execution of the modules, access parameter and configuration data from the parameter and configuration access layer, and support abort-ability and restart-ability.

The harness is also required to support parallel execution of modules on a single or multiple machines and to support multiple simultaneous executions of the pipeline.

The harness is flexible in order to permit easy integration of new modules and adaptation to new instruments. As such the Harness will gather configuration and parameter values from the configuration and parameter access layer and shall not store any such data in the harness itself.

The harness manages core process steps: reduction list generation, calibration and reduction as well as one post reduction step for CFHT: distribution.

4.1.3 The Module

Conceptually, a module executes a single, specific processing step in the pipeline producing one output product. Core modules, as described in the *OPERA Technical, Operational and Scientific Requirements* document, will be developed by CFHT. Analysis and post-reduction modules will be developed by both CFHT and collaborators.

Generally speaking a module is an executable processing step. Each module should take all input parameters on the command line and endeavor to produce a single output. It may not be possible to have a module produce a single output, due to a large overlap in processing required to generate the outputs. However, the only way that the harness can fulfill the requirement that the pipeline be restartable and recover from aborts, is that the output be known to the Harness. If there is more than one output then managing the outputs becomes very difficult. So, while exceptions can be made, in general good module design should follow the guideline that the module produces a single output. The output may be a vector, or a table in a file, but the output should be a single entity.

Modules do not contain numeric or textual “hard-coded” parameters. Examples of such parameters would be temporary file paths, byproduct paths, gain or noise estimates for a particular instrument, and the like.

The input and output files are required to be plain text or FITS images or FITS tables. The final output products are required to be FITS tables in image format.

Modules do not use shared memory. This design decision stems from the requirement to support multiple simultaneous execution instances of the pipeline on multiple machines.

There are a number of standard file paths supplied by the harness for use by modules and tools. The creation and use of standard file paths for use by all modules stem from the requirement that modules be parameterized to execute in multiple external environments.

A module does not generate temporary file names, unless the filename is based on the odometer of an input file. Odometers are numbers represent-

ing unique images taken at CFHT. All temporary files are placed in the temporary directory as determined by the harness and configuration.

Modules do not test for the existence of a temporary file, or byproduct file or output product file and skip the processing step if a file is found. This design decision stems from the requirements that the pipeline produce correct results, support 99+% availability, support recovery from aborts and support multiple simultaneous executions of the pipeline.

Modules do not modify any input file. This is a direct requirement from the Requirements document.

Modules write error and warning messages to stderr in C programs or cerr in C++ programs. If the flag --verbose is passed to the module the informational messages is written to stdout in C programs or cout in C++ programs. Libraries should not write to stderr or stdout, but rather should return error codes to be handled by the module. The file operaError.h contains the consistent set of error codes.

Modules are responsible for catching exceptions, printing the result string and exiting with an EXIT_FAILURE on catching an exception.

Modules accept at least the standard parameters:

`--plot, -p`	*generate plot files suitable for gnuplot*
`--verbose, -v`	*print verbose informative messaging to stdout*
`--debug, -d`	*print debug messages to stdout*
`--trace, -t`	*trace module calls*
`--help, -h`	*issue usage information and exit*

Core modules, software libraries and tools will be written by CFHT software engineers in C and C++ computer language. This design decision is driven by the requirement for core reduction speed.

Contributed core modules may be written in any computer language, but contributions in other languages will be rewritten in C or C++ by CFHT personnel. This design decision stems from the requirement that the open source nature of OPERA would be compromised if core modules were written in a computer language that requires a license in order to execute. OPERA was written to have as few dependencies as possible. Introducing dependencies is generally frowned upon.

4.1.4 Software Libraries

A software library consists of a group of functions that can be linked to an executable program (module or tool). For example, there will be OPERA modules that read an image in FITS format. These modules may use the CFITSIO library for basic I/O, but the OPERA library interfaces provide the additional benefit of a more abstract operaImage datatype interface. All external libraries used by OPERA must be open-source and must not have licensing restrictions that would enforce a licensing modification on OPERA.

All libraries are written as re-entrant code. We have opted to write all OPERA software libraries in the C/C++ languages. For those written in C, we lose some of the powerful object properties and exception handling of C++, C libraries may be called from either C or C++ and the converse is not true.

In general the image and statistic libraries are written in C. Any libraries written in C should return error codes from every function. Any libraries or modules or classes written in C++ should check return codes and throw an exception on error. The operaFITSImage class, for example, checks CFITSIO status returns and throws an exception on error. The operaException class has a constructor that includes a message, file, function and line number. This is the preferred constructor as it allows us to quickly locate an

error. The error codes and text tables associated with each code will be stored in a common operaError.h to ensure there is no overlap between modules/libraries. Callers of image libraries should also check for NaNs using the builtin isnan() function. The library functions for some commonly used high profile calls, such as quicksort for finding a median for example, come in various flavors:

i. nondestructive function call

ii. inline function call (for speed)

iii. destructive function call (for more speed)

Why all the different flavors? Inlined functions are faster than called functions, but they take more space as the code is inlined at every call site. Image median is an example of a function with a side effect. It uses quicksort to find the median pixel value, and quicksort has the side effect of sorting the image pixels. This is not what the users might expect the median function to do. So, a copy of the image is made first, then the copy is sorted. However there are times when the side effect is not a problem (median stacking for example, where the inputs are never saved). In this case, to make median stacking fast enough, the caller may opt for the destructive, inlined version, the fastest flavor.

4.2 The Harness or Driver in Detail

Many pipeline use a script or driver written in a high-level language such as python to control execution flow. Some use workflow tools to ease the process of arranging for the processing steps to be executed in the correct order. Scripts however have some serious drawbacks for production pipelines. Let's look at some properties of a driver vs the OPERA harness:

	OPERA	SCRIPT
Uses high level rules to link workflow.	✔	some
Supports parallel processing step execution.	✔	
Dependency driven, so that if aborted, the reduction continues safely from where it left off.	✔	
Supports remote server execution of processing steps.	✔	
Hierarchical commands.	✔	
Scaleable from a single machine to many.	✔	
Configurable and adaptable to new instruments and locations.	✔	
Autonomous execution,	✔	some

Table 1 - A Comparison of the OPERA Harness vs a Script

The harness design is drawn from the experience in creating production harnesses for the CFHT Upena reduction pipeline and the Ii'wi WIRCam pipeline. Upena makes use of the facilities of the Linux tool called "*make*". *make* offers several features useful in meeting the Harness requirements. *make* is dependency-based, so that if a product has been already made in a previous invocation, then *make* does not remake the product, but simply moves on to the next. This satisfies the requirement of restart-ability. *make* also cleans up products on abort. In this way consistency in aborting and restarting the pipeline is guaranteed, thus satisfying another requirement of the Harness.

make also handles parallelism. The rules can be written in such as way that work be distributed on multiple machines, but rules can also be written such that only a single machine is used for reduction.

make also assists in configurability. Rules in the "Makefile" map one-for-one with module invocations.

make also supports "include", which includes other Makefiles. This facility can be used to tailor the reduction process for a particular installation or instrument. For example, a default configuration for CFHT might look like:

Makefile:

```
include $(instrument)/Makefile.core
include $(instrument)/Makefile.calibration
include $(instrument)/Makefile.util
include $(instrument)Makefile.analysis
```

Where:

```
instrument := espadons
```

In this way custom reduction steps may be easily created and installations may easily tailor the OPERA pipeline to their own environment and needs.

The Harness infrastructure developed for the needs of CFHT may also be used by other installations. CFHT will execute OPERA as a production pipeline, meaning that the data must be distributed within a certain period of time after the data is taken, and as such, multiple machines must be used for reduction. Other installations may not have production-level needs. The OPERA infrastructure services both situations. For example, CFHT might define the reduction machine list as:

```
MACHINES    := polena  halea      ula
LOADMAP     := 8       4          4
```

Meaning that 3 machines with different load capabilities may be used for reduction. Another installation may desire only one machine:

```
MACHINES    := pluto
LOADMAP     := 8
```

The default configuration is a single process running on this host:

```
MACHINES    := %%{HOSTNAME}
LOADMAP     := 1
```

The simplest method found for accessing the parameter access layer and configuration access layer was to have those repositories actually be part of the Make harness structure. So,

Makefile.configuration

stores the configuration data and

Makefile.parameters

stores the parameters data. In this way, both parameters and configuration data are readily available to the harness:

Makefile:

```
include $(instrument)/Makefile.parameters
include $(instrument)/Makefile.configuration
include $(instrument)/Makefile.core
include $(instrument)/Makefile.calibration
include $(instrument)/Makefile.util
include $(instrument)/Makefile.analysis
```

This scheme has the advantage that features of Make such as ifdefs and variable substitution can be used in addition to the simple data definitions. The parameters and configuration access layers can then use the same repository as the harness.

Thus an entry in the configuration Makefile may be:

```
##############################################################################################
# operaReductionSet
##############################################################################################
DETECTORS              := EEV1 OLAPA
INTMODES               := sp1 sp2
POLMODES               := pol
MODES                  := sp1 sp2 pol
SPEEDS                 := Slow Fast Normal XSlow
EEV1_AMPLIFIERS        := a
OLAPA_AMPLIFIERS       := ab
AMPLIFIERS             := a ab
OSETS                  := a b c
SPLITKEY               := INSTMODE
```

and the parameters Makefile may be:

```
##############################################################################################
# gain / noise
##############################################################################################
#                           xstart ystart dx dy
EEV1_gainsubformata          :=  500 1000 1500 1500
OLAPAa_gainsubformata        :=  500 1000 1500 1500
OLAPAab_gainsubformatab      :=  500 500 1500 1500 1100 500 1500 1500

gainMinPixPerBin             := 1000
gainMaxNBins                 := 30
gainLowestCount              := 1500
gainHighestCount             := 25000
```

The rule for a reduction list looks like:

```
################################################################################################
# create a reduction list
# Note: The reduction list may be split in the case of returning to a mode with an
# intervening mode. e.g. sp1->pol->sp1 The two sp1 sets are split with a ###### comment
# in the output from operaReductionlist. The two sets are then split here in the harness.
# Note 2: In the case of CFHT, the ARCHIVE keyword may be set to retrieve images from
# the archive rather than a directory.
################################################################################################
rlist_$(QUALIFIERS).dat:
         @start=$$SECONDS; \
         $(bindir)/operaReductionSet
         --splitkey=$(SPLITKEY)
         --input=$(tmpdir)/filelist.txt
         --output=$(byproductsdir)/rlist_$(QUALIFIERS).dat
         --qualifiers="$(QUALIFIERNAMELIST)" $(word 1,$(QUALIFIERNAMELIST))=$(word 1,$(QUALIFIER-
NAMELIST))_$(DETECTOR) $(word 2,$(QUALIFIERNAMELIST))=$(word 2,$(QUALIFIERNAMELIST))_$(MODE)
$(word 3,$(QUALIFIERNAMELIST))=$(word 3,$(QUALIFIERNAMELIST))_$(SPEED) $(word 4,$(QUALIFIERNAMEL-
IST))=$(word 4,$(QUALIFIERNAMELIST))_$(AMPLIFIER)
         --etype=OBJECT --etype=FLAT --etype=BIAS --etype=COMPARISON --etype=ALIGN $(optargs) ; \

         split reduction set into sets in the case of an intervening mode change
         create gain list by removing split calibrations
```

filelist.txt contains a set of file names, usually derived from a single directory, but whatever the installation chooses as the full set of fils to consider in a reduction. Note that there is a special CFHT-only rule that generates the file list from an archive query.

The rule to make a master bias product looks like:

```
################################################################################################
# create a single master bias
################################################################################################
masterbias_$(QUALIFIERS).fits:
         @start=$$SECONDS; \
         if [[ -s $(byproductsdir)/rlist_$(QUALIFIERS).dat ]] ;  then \
               biases=`getbiases for $(QUALIFIERS)`; \
               $(bindir)/operaMasterBias \
               --output=$(calibrationdir)/$*.fits \
               $${biases} \
               --pick=$(masterbiaspick) \
               $(optargs) 2>>$(errfile) 2>&1 | tee -a $(logfile) ; \
               echo "$(pref) master bias for $(QUALIFIERS) complete."; \
         fi
```

This rule gives the recipe for producing a master bias FITS file product from inputs, calling the operaMasterBias module, described later in this document.

The rule to make a master flat product looks like:

```
##############################################################################################
# create a single master flat
##############################################################################################
masterflat_$(QUALIFIERS).fits:
        @start=$$SECONDS; \
        if [[ -s $(byproductsdir)/rlist_$(QUALIFIERS).dat ]] ;  then \
              flats=`getflats for $(QUALIFIERS)`; \
              $(bindir)/operaMasterFlat \
              --output=$(calibrationdir)/$*.fits \
              $${flats} \
              --pick=$(masterflatpick) \
              $(optargs) 2>>$(errfile) 2>&1 | tee -a $(logfile) ; \
              echo "$(pref) master flat for $(QUALIFIERS) complete."; \
        fi
```

This rule gives the recipe for producing a master flat FITS file product from inputs, calling the operaMasterFlat module, described later in this document.

The rule to make a master comparison product looks like:

```
##############################################################################################
# create a single master comparison
##############################################################################################
mastercomparison_$(QUALIFIERS).fits:
        @start=$$SECONDS; \
        if [[ -s $(byproductsdir)/rlist_$(QUALIFIERS).dat ]] ;  then \
              comparisons=`getcomparisons for $(QUALIFIERS)`; \
              $(bindir)/operaMasterComparison \
              --output=$(calibrationdir)/$*.fits \
              $${comparisons} \
              --pick=$(mastercomparisonpick) \
              $(optargs) 2>>$(errfile) 2>&1 | tee -a $(logfile) ; \
              echo "$(pref) master commparison for $(QUALIFIERS) complete."; \
        fi
```

This rule gives the recipe for producing a master comparison FITS file product from inputs, calling the operaMasterComparison module, described later in this document.

The rule to make a master fabryperot product looks like:

```
##############################################################################################
# create a single master fabperot
##############################################################################################
masterfabperot_$(QUALIFIERS).fits:
        @start=$$SECONDS; \
        if [[ -s $(byproductsdir)/rlist_$(QUALIFIERS).dat ]] ;  then \
              fabperots=`getfabperots for $(QUALIFIERS)`; \
              $(bindir)/operaMasterFabperot \
              --output=$(calibrationdir)/$*.fits \
              $${fabperots} \
              --pick=$(masterfabperotpick) \
              $(optargs) 2>>$(errfile) 2>&1 | tee -a $(logfile) ; \
```

```
        echo "$(pref) master fabperot for $(QUALIFIERS) complete."; \
    fi
```

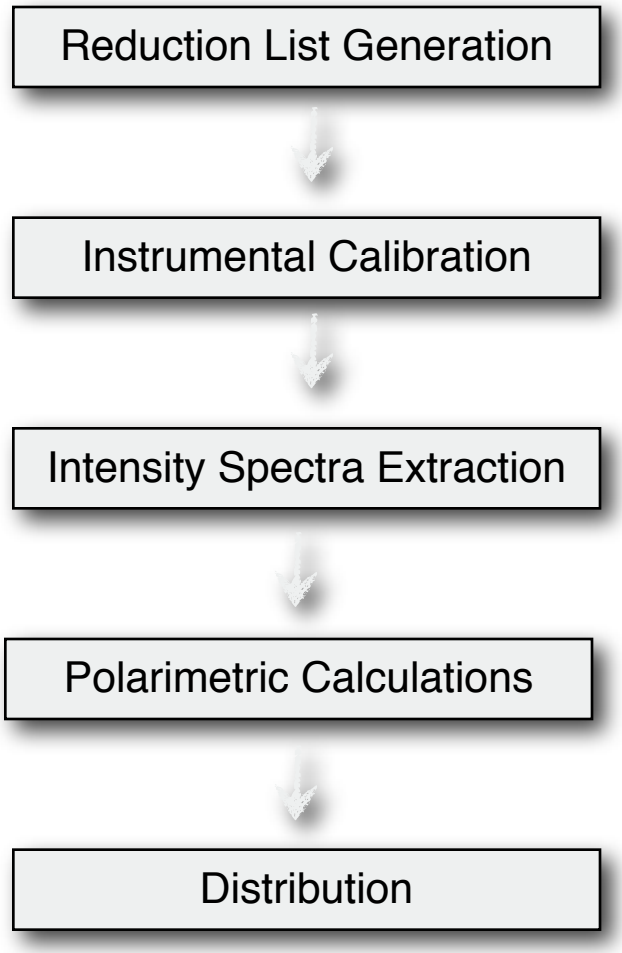
This rule gives the recipe for producing a master fabperot FITS file product from inputs, calling the operaMasterFabperot module, described later in this document.

The rule to calculate a gain noise vector looks like:

```
##############################################################################################
# calculate gain for a given mode, speed amp set
##############################################################################################
gain_$(QUALIFIERS).dat:
    @start=$$SECONDS; \
    if [[ -s $(byproductsdir)/glist_$(QUALIFIERS).dat ]] ;  then \
        flats=`get flats`; \
        biases=`get biases`; \
        echo "$(pref) Starting gain calculation for $(QUALIFIERLIST)." ; \
        gainnoise=`$(bindir)operaGain \
            $${flats} \
            $${biases} \
            --badpixelmask=$(badpixelmask) \
            --defaultgain=$(gain) \
            --defaultnoise=$(noise) \
            --gainsubformata="$($(DETECTOR)$(AMPLIFIER)_gainsubformata)" \
            --gainsubformatab="$($(DETECTOR)$(AMPLIFIER)_gainsubformatab)" \
            --gainMinPixPerBin=$(gainMinPixPerBin) \
            --gainMaxNBins=$(gainMaxNBins) \
            --gainLowestCount=$(gainLowestCount) \
            --gainHighestCount=$(gainHighestCount) \
            $(optargs) 2>>$(errfile)` ; \
        gain=`$(bindir)/operagetword 1 $$gainnoise` ; \
        noise=`$(bindir)/operagetword 2 $$gainnoise` ; \
        echo $${gainnoise} >$(byproductsdir)$* ; \
        echo "$(pref) >>> expected gain       (e/adu): $(gain) for $(QUALIFIERLIST)" ; \
        echo "$(pref) >>> estimated gain      (e/adu): $${gain} for $(QUALIFIERLIST)" ; \
        echo "$(pref) >>> expected readout noise  (e): $(noise) for $(QUALIFIERLIST)" ; \
        echo "$(pref) >>> estimated readout noise (e): $${noise} for $(QUALIFIERLIST)" ; \
        echo "$(pref) gain for $(QUALIFIERS) complete in $(deltat) seconds." ; \
    fi
```

4.3 Overall Core Conceptual Data Flow Through the OPERA Pipeline

The harness manages all process steps, reduction list generation, instrumental calibration, intensity spectra extraction and polarimetric calculations as well as, for CFHT, one post reduction step: distribution. Note that a reduction list is created for each detector, mode, speed and set.The entire diagram is shown so that the reader may grasp the entire flow..



4.4 Pipeline Operation

There are two major steps involved in obtaining spectroscopic and polarimetric reduced data. First is calibration and second is reduction. Calibration consists of many smaller steps: creating master calibration images, creating a bad-pixel mask, calculation of gain and bias, geometry calibration, instrument profile calibration, aperture calibration, creating a pixel-by-pixel sensitivity map and wavelength calibration. Reduction consists, at a high level, of optimal intensity extraction and polarimetry. Calibration must precede reduction.

OPERA harness commands are designed to be dependency-driven and also to be hierarchical. Dependency- driven, in this context, means that if a calibration or reduction step requires a prior step in order to complete, that all prior commands will automatically be issued. Thus rather than having to issue errors such as masterbias calibration missing for example, the pipeline simply proceeds to create all required prior steps in order to complete the requested step. Hierarchical means that you have on your demand, commands at a very high level, like Perform all calibrations for all modes and speeds of images. You may also issue commands at a finer granularity. For example Perform gain and bias calculation or Create reduction lists or Perform master calibrations or Perform Geometry Calibrations.

4.1.1 Calibration

Calibration consists of the steps of obtaining information from calibration images that will be used later in the reduction of science data. The highest level OPERA command to perform all calibration steps is:

```
opera <location of data > mastercalibrations
```

This performs the following steps:

1. reduction set creation

2. master calibration images (masterflats, mastercomparisons, masterfabryperots, masterbiases, normalized- flats)

3. gain and noise values

4. geometry calibrations

5. instrument profile calibration

6. extraction aperture calibrations

7. create badpixel mask

8. create pixel-by-pixel sensitivity map

9. wavelength calibrations

Each of these calibration steps can be done individually, if desired.

4.1.2 Overview of Calibration Modules

Below we briefly describe the OPERA calibration modules.

- operaReductionSet.

  This module generates a vector of reduction units qualified by mode, speed, detector and set. The calibration is placed in a calibration file with extension *.rlst. The Reduction units are lists of file paths to the relevant source data. It groups the calibration files and object files that share the same instrument configuration for reduction.

- operaMaster(Flat, Bias, Comparison, FabPerot).

  These modules median combine a set of exposures into a single master calibration image, with lower noise and free of cosmic rays. The product of this module is a master compressed FITS image. We

note here that all intermediate products in the OPERA pipeline are stored in compressed FITS format (RICE compressions by default) and that the type of compression used is defined as a pipeline parameter.

- operaGain.

  This module performs measurements of the CCD gain and detector noise, for each amplifier, obtained from a set of raw flat-field and bias exposures. It creates the calibration product $*$.gain.

- operaGeometryCalibration.

  This module first detects spectral orders from a master flat-field image. Then it populates the operaGeometry class with photo-center points for each order and fits an optimal polynomial model to the photo-center data and creates the calibration product .geom. This file contains the fit polynomial coefficients, uncertainties, and the $\chi 2$.

- operaInstrumentProfileCalibration.

  This module first measures a one-dimensional spatial profile along rows from a flat-field exposure. Then it uses the measured spatial profile to detect spectral lines from either a comparison lamp or a Fabry-Perot exposure, by means of cross-correlation. The two-dimensional profile is truncated to a given size. Typically for ESPaDOnS we set a 30 pixels (columns) x 5 pixels (rows) window, which is oversampled by a factor of 5 for each direction. Then it stacks a normalized two-dimensional oversampled profile for a given minimum bin-size and minimum number of spectral lines. Then it fits a polynomial in the dispersion direction for each oversampled sub-pixel data. It populates the operaInstrumentProfile class with instrument profile

data, which consists of an oversampled image, where each sub-pixel is a set of polynomial coefficients. This gives a model for the two-dimensional IP at any position within the order, or at any dispersion element. The calibration data is stored into the calibration product $*$.prof.

Note that since the IP gives a more accurate measurement of the illumination profile, it allows a better estimate of the photo-center and therefore a refinement of the geometry model. This will update the geometry calibration file $*$.geom. Also, once geometry is refined, the instrument profile can be measured with better accuracy and refined as well. This will update the instrument profile calibration file $*$.prof. One iteration of this refining process is found to be sufficient to attain the levels of uncertainty to meet the requirements of OPERA.

- operaExtractionApertureCalibration.

  This module uses the two-dimensional IP to measure the orientation of the slit. The slit shape is assumed to be rectangular, allowing the user to select a number of beams within which the slit will be divided into for independent flux measurements. Each aperture beam is an oversampled tilted rectangle with fixed dimensions. The tilt is measured as the angle that maximizes the flux fraction per spectral element. Two additional apertures are created on both side ends of the slit for background estimates. This module produces the calibration file $*$.aper, which contain the tilt angle and all the necessary information used by extraction to create the aperture elements.

- operaCreateBadpixMask.

This module creates a bad-pixel mask. A bad-pixel mask is a FITS image with binary values (0 or 1), which is used by extraction to identify and ignore bad pixels.

- operaPixelbyPixelSentivityMap.

  This module creates a pixel-sensitivity map (normalized flat). The map is a FITS image where each pixel contains a normalized value, which is used to weigh pixels in order to account for pixel-by-pixel sensitivity inhomogeneities. This is measured from a master flat-field image.

- operaWavelengthCalibration.

  This module uses the two-dimensional profile and geometry to detect spectral lines in a comparison lamp exposure by means of cross-correlation. ESPaDOnS uses a Th-Ar atlas as comparison. Calibration starts from an initial hint (wavelength at the center of each order) to select a set of known spectral lines from the reference atlas. OPERA currently uses the atlas of Lovis & Pepe (2007).5 The spectral lines in the atlas are matched against those measured from the comparison image. The match is performed by searching the highest correlation between the line positions in the image and in the atlas. The module uses the operaWavelength class to store a vector of wavelengths (nm) and distances (in pixel units) for a set of spectral lines identified from the master comparison image. Then it fits an optimal polynomial model to the data and creates the calibration file $*$.wave.

4.1.3 Reduction

Reduction consists at a high level, of intensity optimal extraction and polarimetry. Other intermediate steps are normalization, wavelength telluric correction and (at CFHT) FITS product packaging. The basic command to perform all reductions is:

```
opera <location of data> reduce
```

This performs the following steps:

1. extraction and signal-to-noise calculation
2. normalization
3. telluric correction
4. FITS product creation

Polarimetry proceeds in parallel. Again there are subcommands available for finer control of the pipeline:

```
opera <location of data> intensity
```

```
opera <location of data > polarimetry
```

```
opera <location of data> spectrum FILE=...
```

The first commands perform as one would expect, while the third performs a reduction on a single image for quicklook purposes while observing.

4.1.4 Overview of Reduction Modules

Below we briefly describe the OPERA reduction modules.

- operaExtraction.

This module makes use of all calibrations to optimally extract the energy flux information for each wavelength element from an input two-dimensional echellogram spectral image. The bias subtraction, flat-field corrections, and bad-pixel mask are used in extraction. Extraction populates an operaSpectralElement class instance, which contains multiple vectors of flux, variances and wavelength values, obtained for each individual beam, using each one of the following extraction methods:

1. Raw Flux Sum,
2. Standard Extraction (with background subtraction),
3. Optimal Extraction
4. Opera Optimal Extraction.

- operaCreateBadpixMask.

  This module creates a bad-pixel mask. A bad-pixel mask is a FITS image with binary values (0 or 1), which is used by extraction to identify and ignore bad pixels.

- operaPixelbyPixelSentivityMap.

  This module creates a pixel-sensitivity map (normalized flat). The map is a FITS image where each pixel contains a normalized value, which is used to weigh pixels in order to account for pixel-by-pixel sensitivity inhomogeneities. This is measured from a master flat-field image.

- operaWavelengthCalibration.

  This module uses the two-dimensional profile and geometry to detect spectral lines in a comparison lamp exposure by means of cross-

correlation. ESPaDOnS uses a Th-Ar atlas as comparison. Calibration starts from an initial hint (wavelength at the center of each order) to select a set of known spectral lines from the reference atlas. OPERA currently uses the atlas of Lovis & Pepe (2007).5

The spectral lines in the atlas are matched against those measured from the comparison image. The match is performed by searching the highest correlation between the line positions in the image and in the atlas. The module uses the operaWavelength class to store a vector of wavelengths (nm) and distances (in pixel units) for a set of spectral lines identified from the master comparison image. Then it fits an optimal polynomial model to the data and creates the calibration file $*$.wave.

- operaNormalize. This module detects the continuum emission in an object spectrum and performs flux normalization to the continuum. The continuum is not necessarily the intrinsic emission of the source but a combination of any (possibly many) low frequency elements imprinted on the spectra. This operation will populate the OPERA products with an additional flux vector which is normalized to 1.

- operaPolar. This module produces the spectro-polarimetric OPERA products, which consist of the measurements of the degree of polarization for a given Stokes parameter. For spectro-polarimetry, ESPaDOnS is operated in polar mode, which requires a minimum of four exposures in order to create a single polarimetric product for each Stokes parameter (Q, U or V). Each of these exposures is obtained with the retarding plates in specific rotations with interleaving positions, which allow a differential measurement of the degree of polarization. This module uses the flux information obtained from extraction. Therefore it also

provides the usual intensity spectrum (Stokes I) for each individual exposure.

- operaStarPlusSky. ESPaDOnS in star+sky mode is operated with two fiber channels, which produces two beams, one for sky flux and another for the source. Therefore, this module produces a product which also contains independent measurements for the sky flux, and also the sky-subtracted object flux.
- operaTelluricWavelengthCorrection. This module identifies atmospheric emission/absorption spectral lines on science images to improve the wavelength calibration. The signal-to-noise for these lines depends on the exposure time and on the target brightness, therefore the quality of this correction is not consistent for all objects observed. For this reason, this module produces an additional vector of corrected wavelength values.
- operaHeliocentricWavelengthCorrection. TBD...
- operaCreateProduct. This module packages OPERA pipeline products into a single FITS table. This final product is delivered to scientists and stored in the archive.

5. Core Modules

5.1 operaReductionSet

Description:

This module produces a list of FITS files that matches a set of pre-defined qualifiers and also that matches a given set of observation types (like FLAT, OBJECT, CALIBRATION, etc.).

The module reads information from a configuration file (config-file). The directories where the program will parse data can be either defined in the config-file by the entry **DATADIR**, or in the command line, which will override the config-file information.

The etype (observation type) of files to be selected should be provided in the command line. All etype options must be listed under the entry **ETYPE**. Each option should have a respective `$OPTION_HEADERVALUE` in the config-file. The header keyword in the FITS file to be inspected for the etype must also be defined in the config-file under the entry **OBSTYPE_HEADERKEY**.

The qualifiers can be either specified in the command line or defined under the entry **QUALIFIERNAMELIST** in the config-file. For each defined qualifier there should be a header keyword definition under the entry $QUALIFIER_**HEADERKEY**. Note that there should be only one header key for each qualifier. Also for each qualifier one should define options for the header value, under the keyword $QUALIFIER_**HEADERVALUE**. Note that qualifier header values can have multiple definitions. All possible values are listed in the config-file and the choice of which value will be picked for the selection is entered directly from the command line.

Below is an example of a config-file (Makefile.configuration):

```
-----------------------------------------------------------------------
DATADIR                 := /home/TestData1 /home/TestData2
#
OBSTYPE_HEADERKEY       := OBSTYPE
ETYPE                   := OBJECT BIAS FLAT ALIGN COMPARISON
OBJECT_HEADERVALUE      := 'OBJECT  '
BIAS_HEADERVALUE        := 'BIAS    '
```

```
FLAT_HEADERVALUE         := 'FLAT    '
ALIGN_HEADERVALUE        := 'ALIGN   '
COMPARISON_HEADERVALUE   := 'COMPARISON'
#
QUALIFIERNAMELIST        := DETECTOR MODE SPEED AMPLIFIER
DETECTOR_HEADERKEY       := DETECTOR
MODE_HEADERKEY           := INSTMODE
SPEED_HEADERKEY          := EREADSPD
AMPLIFIER_HEADERKEY      := AMPLIST
#
DETECTOR_EEV1_HEADERVALUE     := 'EEV1    '
DETECTOR_OLAPA_HEADERVALUE    := 'OLAPA   '
AMPLIFIER_a_HEADERVALUE       := 'a       '
AMPLIFIER_ab_HEADERVALUE      := 'a,b     '
MODE_pol_HEADERVALUE          := 'Polarimetry, R=65,000'
MODE_sp1_HEADERVALUE          := 'Spectroscopy, star+sky, R=65,000'
MODE_sp2_HEADERVALUE          := 'Spectroscopy, star only, R=80,000'
SPEED_Fast_HEADERVALUE        := 'Fast:%'
SPEED_Normal_HEADERVALUE      := 'Normal:%'
SPEED_Slow_HEADERVALUE        := 'Slow:%'
-------------------------------------------------------------------------
```

Note the entries in black (`DATADIR`, `OBSTYPE_HEADERKEY`, `ETYPE`, and `QUALIFIERNAMELIST`) are those that will be searched by the module, where `OBSTYPE_HEADERKEY` and `ETYPE` are mandatory entries.

The entries shown in green are definitions of qualifiers that are passed to the module. These qualifiers can be either listed under the entry `QUALIFIERNAMELIST` in the config-file or in the command line option `--qualifier`. In this case, there are four qualifiers defined: `DETECTOR MODE SPEED AMPLIFIER`, each of which has a correspondent entry headerkey entry, `$QUALIFIER_HEADERKEY`, and multiple options for the keyword values. For instance the qualifier MODE, which in our FITS images is recorded under the keyword INSTMOD, bear three different modes of operations, and therefore three definitions for header value options are required. In the config-file above it is shown the three modes of operation for ESPaDOnS: `pol`, `sp1`, and `sp2`. The choice of which mode is used as a qualifier is made by entering the following option in the command line: `MODE=pol`. Then, in this example, the header value to be matched is `INSTMOD = 'Polarimetry, R=65,000'`.

If there is a qualifier and a mode for which the definition is missing in the config-file, then the module would ignore that qualifier and consider only the ones that have been defined.

Command line options:

```
-h, --help            display help page
-v, --verbose         activate message sending
-d, --debug           activate debug messages
-t, --trace           activate trace messages
-i, --input           input file with list of paths
-o, --ouput           output file to save list of reduction set
-r, --directory       input directories (overrides config)
-q, --qualifier       qualifiers to select reduction set
-e, --etype           obstype to select reduction set
-s, --splitkey        keyword to be watched for mode changes
```

Output:

The module outputs a list of FITS file paths. The default is to send the output to `stdout` unless the `--output` command line option is provided, and in this case the list is sent to a file.

Usage:

```
operaReductionSet [-hvdt] [--input=<LIST_OF_PATHS>] [--output=<OUTPUT_FILE>] [--directory=<DIR1 DIR2 ...>] [--splitkey=<KEY>] --etype=<OBSTYPE1 OBSTYPE2 ...> --qualifiers=<'QUALIFIER1 QUALIFIER2 ...'> QUALIFIER1=<MODE1> QUALIFIER2=<MODE3> ...
```

Example:

```
operaReductionSet --input=$(tmpdir)/filelist.txt --splitkey=$(SPLITKEY) --etype=BIAS --etype=FLAT --qualifiers='DETECTOR MODE SPEED' DETECTOR=EEV1 MODE=pol SPEED=Slow
```

The `--splitkey` option sets the keyword `INSTMODE` to be watched. This means that when parsing the stack of files if this keyword value has

changed, then the module will insert a split-point mark "##########" in the list.

The `--etype` option tells the module to select files in the two categories "BIAS" and "FLAT", which are defined in the config-file. From the definitions given in the config-file shown above, these categories are considered to be images where the observation type value in the header matches the strings `'BIAS '` and `'FLAT '`, respectively. Note that the keyword for the observation type is defined as `OBSTYPE_HEADERKEY := OBSTYPE`.

The `--qualifiers` option tells the module to select images qualified by `DETECTOR`, `MODE`, and `SPEED`. If you look at the config-file given above, you will notice that each of these qualifiers has a defined header keyword:

```
DETECTOR_HEADERKEY  := DETECTOR
MODE_HEADERKEY      := INSTMODE
SPEED_HEADERKEY     := EREADSPD
```

For each of these qualifiers there is a selected option, `DETECTOR=EEV1 MODE=pol SPEED=Slow`, each of which has its header value definition,

```
DETECTOR_EEV1_HEADERVALUE  := 'EEV1    '
MODE_pol_HEADERVALUE       := 'Polarimetry, R=65,000'
SPEED_Slow_HEADERVALUE     := 'Slow:%'
```

Then these values are the ones that will be matched to select images to be printed out. Note the wild card % in the last entry: `'Slow:%'`.

Algorithm:

1. Read input parameters.

2. Handle etypes:

- Read `ETYPE` categories listed in the config-file.
- Search category provided in the command line and get header value definition from the entry `ETYPE_HEADERVALUE` in config-file.

2. Get obstype keyword value from entry `OBSTYPE_HEADERKEY` in config-file.

3. Handle qualifiers:

- Read default qualifiers from entry `QUALIFIERNAMELIST` in config-file.
- Search for command line qualifiers; keep default if no input.
- Search keyword definitions for each qualifier: `QUALI_HEADERKEY`
- Search config-file for default selected options for each qualifier.
- Parse command line to search for selected options for each qualifier. (overrides default from config-file)
- Read header values for each selection: `QUALI_OPT_HEADERVALUE`

4. Open input directories and get file names:

if (there is a command line `--input` = list.txt) **then**
read all `FILE_PATH` listed in list.txt
else if (if there is command line --directory = DIRS) **then**
read `FILE_PATH` of files inside DIRS
else
read `FILE_PATH` of files inside default directories listed in config-file (DATADIR).

5. Parse files and generate reduction list:

for i=0 to i=(nfiles-1) **do** {
if `FILE_HEADERVALUE` matches **any** `ETYPE_HEADERVALUE`
Condition based on obstype: *obstype1* || *obstype2* || ..
and
if `FILE_HEADERVALUE` matches **all** `QUALI_OPT_HEADERVALUE`
Condition based on qualifiers: *quali1* && *quali2* && *quali3* && ..
then {
Print `FILE_PATH`
if `SPLIT_KEY` has changed **then**
Print ##########
}
}

Example:

```
operaReductionSet --input=filelist.txt
--output=rlist_OLAPAa_pol_Slow.dat --splitkey=INSTMODE --qualif-
ers=" detector mode speed" --etype-BIAS --etype=COMPARISON
--etype=ALIGN
```

5.2 operaMasterFlat

Description:

- Creates a master flat field fits image from a number of flat calibration images, rejecting bad pixels.

Inputs:

- all flat calibration images of given detector, mode, speed, set
- bad pixel mask
- an option “pick”, which models the upena “choose one image from the input list. A pick of 0 is taken to mean,do not pick but rather take the median stack of all images.

Output:

- master flat field calibration FITS file

Usage:

```
operaMasterFlat [-hvdt] --images=<...> --pick=<...>
--badpixelmask=<...> --output=<...>
```

Algorithm:

```
Read inputs
```

```
mask bad pixels

If pick is not given,

     For each flat

          For esch pixel:

               Find the Median value of the stack of im-
ages and return the image of medians. The median is found
using the widely known Quick Sort algorithm.

else

     return the pick image pixels
```

Example:

```
operaMasterFlat

--images=/data/opera/byproducts/@ff_OLAPA_pol_Slow_a.lst
--pick=0
--badpixelmask=/data/opera/upena_config/badpixelmask.fits
--output=/data/opera/calibrations/masterflat_OLAPA-a_Pol_Sl
ow.fits
```

5.3 operaMasterBias

Description:

- Creates a bias fits image from a number of bias calibration images.

Inputs:

- all bias calibration images of given detector, mode, speed, set
- bad pixel mask
- an option “pick”, which models the upena “choose one image from the input list. A pick of 0 is taken to mean,do not pick but rather take the median stack of all images.

Output:

- master bias calibration FITS file

Algorithm:

```
Read inputs

mask bad pixels

If pick is not given,

     For each flat

          For esch pixel:

               Find the Median value of the stack of im-
ages and return the image of medians. The median is found
using the widely known Quick Sort algorithm.

else

     return the pick image pixels
```

Usage:

```
operaMasterBias [-hvdt] --images=<...> --pick=<...>
--badpixelmask=<...> --output=<...>
```

Example:

```
operaMasterBias

--images=/data/opera/byproducts/@bias_OLAPA_pol_Slow_a.lst
--pick=1
--badpixelmask=/data/opera/upena_config/badpixelmask.fits
--output=/data/opera/calibrations/masterbias_OLAPA-a_Pol_Sl
ow.fits
```

5.4 operaMasterFabPerot

Description:

- Creates a Fabry-Perot fits image from a number of Fabry-Perot calibration images.

Inputs:

- all Fabry-Perot calibration images of given detector, mode, speed, set
- bad pixel mask
- an option “pick”, which models the upena “choose one image from the input list. A pick of 0 is taken to mean,do not pick but rather take the median stack of all images.

Output:

- master Fabry-Perot calibration FITS file

Usage:

```
operaMasterFabPerot [-hvdt] --images=<...> --pick=<...>
--badpixelmask=<...> --output=<...>
```

Algorithm:

```
Read inputs

mask bad pixels

If pick is not given,

     For each flat

          For esch pixel:

               Find the Median value of the stack of im-
ages and return the image of medians. The median is found
using the widely known Quick Sort algorithm.

else

     return the pick image pixels
```

Example:

```
operaMasterFabPerot
```

```
--images=/data/opera/byproducts/@fabper_OLAPA_pol_Slow_a.ls
t --pick=1
--badpixelmask=/data/opera/upena_config/badpixelmask.fits
--output=/data/opera/calibrations/masterfabperot_OLAPA-a_Po
l_Slow.fits
```

5.5 operaMasterComparison

Description:

- Creates a comparison fits image from a number of comparison calibration images.

Inputs:

- all Comparison calibration images of given detector, mode, speed, set
- bad pixel mask
- an option “pick”, which models the upena “choose one image from the input list. A pick of 0 is taken to mean,do not pick but rather take the median stack of all images.

Output:

- master Comparison calibration FITS file

Usage:

```
operaMasterComparison [-hvdt] --images=<...> --pick=<...>
--badpixelmask=<...> --output=<...>
```

Algorithm:

```
Read inputs
mask bad pixels
```

```
If pick is not given,
     For each flat
          For esch pixel:
               Find the Median value of the stack of im-
ages and return the image of medians. The median is found
using the widely known Quick Sort algorithm.
else
     return the pick image pixels
```

Example:

```
operaMasterComparison
--images=/data/opera/byproducts/@comparison_OLAPA_pol_Slow_
a.lst --pick=2
--badpixelmask=/data/opera/upena_config/badpixelmask.fits
--output=/data/opera/calibrations/mastercomparison_OLAPA-a_
Pol_Slow.fits
```

5.6 operaGain

Description:

- Calculates the overall gain and readout noise for each detector, mode, speed, set, amplifier. configuration

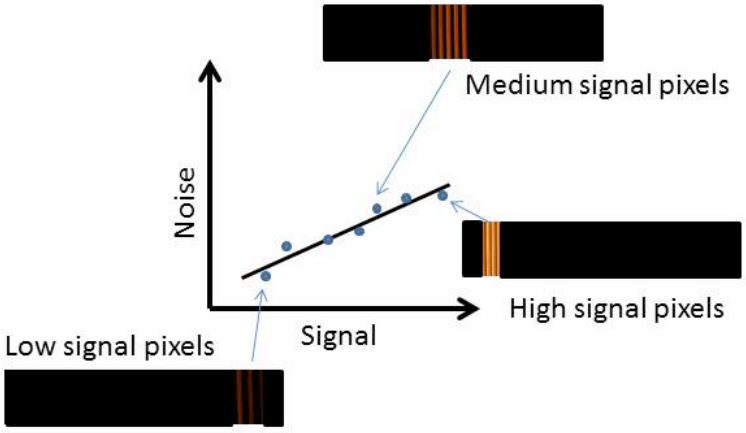
- The gain of a CCD camera is the conversion between the number of electrons ("e-") recorded by the CCD and the number of digital units ("counts") contained in the CCD image and readout by the electronics.

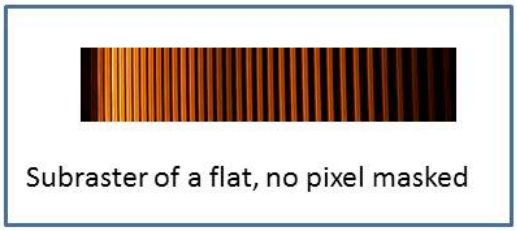
Since quantities in the CCD image can only be measured in units of counts, knowing the gain permits the calculation of quantities such as readout noise and full well capacity in the fundamental units of electrons.

The gain calculation uses the concepts of "signal" and "noise". The signal is defined as the quantity of information measured in the image. The noise is the uncertainty in the signal. KEY POINT: the gain of the CCD is measured by comparing the signal level to the amount of variation (noise) in the signal. This works because the relationship between counts and electrons is different for the signal and the variance (noise).

Different levels of signal are selected in flat field images with a mask, and the signal and noise are calculated from all the values in the selected pixels.

The gain is the inverse slope when the total noise measured in counts, squared, is the y axis, and the signal in counts is the x axis.





Inputs:

- at least 2 flats of same detector, mode, speed, set
- at least 1 bias FITS image (different algorithm is used in the case of a single bias)
- bad pixel mask
- CCD subformat (pix0, npix for each axis) parameter

Output:

- gain and readout noise vector

Usage:

```
operaGain [-hvdt] --biasimgs=<...> --flatimgs=<...>
--masterbias=<...> --badpixelmask=<...>
--gainsubformata=<...> --gainsubformatab=<...>
--gainMinPixPerBin=<...> --gainMaxNBins=<...>
--gainLowestCount=<...> --gainHighestCount=<...>
--output=<...>
```

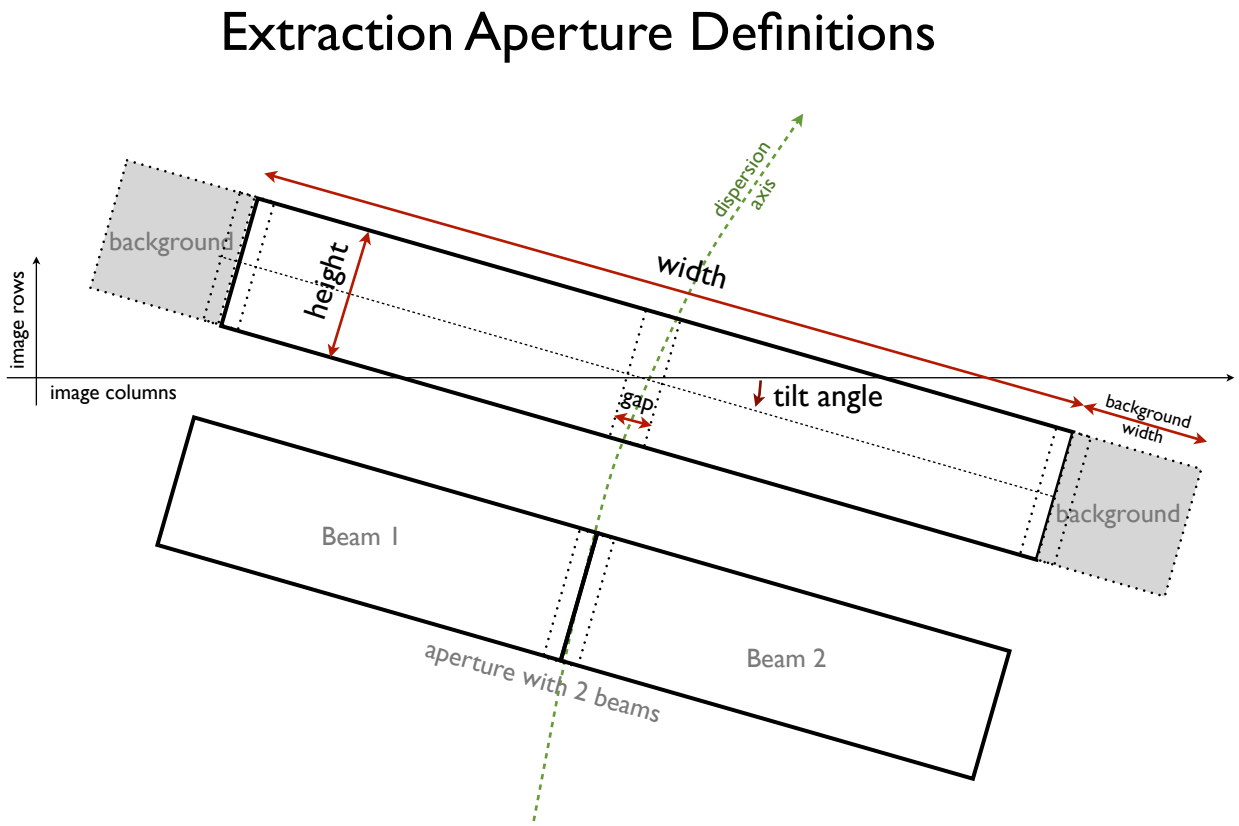
Algorithm:

1. Check for availability of all inputs:
2. Create pixel sub arrays to pass to CCD library
3. Call opearCCDGainNoise library function (see libraries section for description of algorithm)
4. return gain and noise vector

Example:

```
operaGain

--flatimgs=/data/niele/espadons/11BQ01-Mar13/101010f.fits

--flatimgs=/data/niele/espadons/11BQ01-Mar13/101011f.fits

--biasimgs=/data/niele/espadons/11BQ01-Mar13/101012b.fits

--biasimgs=/data/niele/espadons/11BQ01-Mar13/101013b.fits

--badpixelmask=/data/opera/upena_config/badpixelmask.fits

--gainsubformata="500 1000 1500 1500"

--output=/data/opera/byproducts/gain_OLAPA_Pol_Slow_a.dat
```

5.7 operaExtractionApertureCalibration



Description

The module `operaExtractionApertureCalibration` calibrates the aperture for extraction. The aperture is a set of tilted oversampled rectangles, each of which defines an extracting area of an individual beam or background window. The whole set defines the extraction area for each spectral element (or spectral bin). The set contains two background windows, one on each side of the extracting area. The module expects as input the number of beams, the gap between beams (in pixels units), aperture height and width and background aperture width (all in pixel units). Given these parameters the module uses the measurements provided by the instrument profile calibration to measure the aperture tilt angle that provides highest flux fraction within the profile window.

Inputs

- Order spacing calibration file
- Geometry calibration file
- Instrument profile calibration file
- Number of equally spaced rows to sample IP (default = 1)
- Minimum angle value in degrees to measure tilt
- Maximum angle value in degrees to measure tilt
- Precision for angle values in degrees to measure tilt
- Number of beams to split aperture
- Gap between beams in pixel units
- Aperture width in pixel units
- Aperture height in pixel units
- Background aperture width in pixel units

Algorithm

For each spectral order do the following:

2. Select a given number of samples along the order
3. For each sample retrieve the instrument profile model. Then calculate the integrated flux fraction within the profile window for a range of testing angles. The trial angles vary from “MinTiltAngle” to “MaxTiltAngle” in steps of “TiltAnglePrecision”. The latter three quantities are provided as input.

4. Then it calculates the median of selected sample tilt angles. This gives the final tilt angle and uncertainty for a given order.

Then it calculates the median tilt angle between all orders to provide the global tilt, which is adopted as the final aperture tilt angle.

Then again, for each spectral order do the following:

4. Use global tilt to calculate aperture parameters and correspondent flux fraction.

Save aperture information to *.aper calibration file (see Appendix 1).

5.8 operaWavelengthCalibration

Description

The module `operaWavelengthCalibration` obtain measurements of spectral line positions on a comparison lamp exposure and matches those lines with an atlas to obtain the pixel-to-wavelength solution for each spectral order. The module saves the results into the calibration file *.wave.

Input

- Either an atlas of Th-Ar lines or the full Th-Ar calibrated spectrum
- Either a table of uncalibrated line positions in pixel units or a full extracted spectrum
- Geometry calibration file (*.geom)
- Wavelength initial guess file (*.wave) - see Appendix 1
- Line width in pixel units (to set scale)

Algorithm

- Read geometry calibration file.

- Read wavelength calibration file (first guess). This provides a first order wavelength solution.

- Read uncalibrated spectrum or lines. This provides a vector of intensities versus distance in pixel units:

I(d) vs. d

- Read atlas spectrum or lines.

- For each spectral order do:

1. Calculate wavelength vector for uncalibrated raw spectrum using a first guess solution.

2. Measure total distance "D" (in pixel units) covered by the order. This is given by the line integral of the polynomial that describes the center of the order.

3. Calculate wavelength range covered by the order based on the geometry calibration. This range will be used to select the atlas data points lying within this range. The range is chosen to be over the edges, so the search for better solutions will be able to pick lines outside of boundaries.

4. If the input is a list of raw lines then it just reads the lines into a vector. However if the input is a spectrum (for either uncalibrated or atlas), then it runs an algorithm to detect spectral lines in the spectrum. The algorithm uses a cross correlation with a normalized gaussian function. For the uncalibrated spectrum, the opera format (*.e) spectrum file already contains an entry per spectral bin with the cross-correlation between the echelle spectral image and the two-dimensional instrument profile. The user may choose to use either one.

5. Read atlas of spectral lines for the range covered by the order.

At this point all possible lines either in the comparison or in the Atlas have been read. So, below it starts the identification of lines and refining wavelength solution.

6. First step is to adjust the wavelength solution to the maximum correlation between the two sets of lines. The adjustment is done by varying each coefficient of the polynomial solution and finding the maximum cross-correlation between simulated spectrum for the comparison and the atlas. The simulated spectrum is obtained by assigning a gaussian profile for each spectral line. The search for a solution is done by changing the values of the lowest order coefficients of the polynomial solution and obtaining a number (`NpointsPerPar`) of trial solutions. The parameter `ParRangeSizeInPerCent` is a quantity that defines the range for the search. The range is set as a percentage which is relative to the value of the coefficient. This process works better iteratively, so it is performed in a loop where the parameter `nIterforXcorr` defines the number of iterations. The solution that presents the highest correlation between the two sets of lines is saved.

7. Now it identifies the lines by matching the two sets using the best wavelength solution obtained above. This step uses the following routine:

```
matchAtlaswithComparisonLines(acceptableMismatch)
```

where `acceptableMismatch` is an integer number in units of line width, and it defines the maximum acceptable difference between the center of two matching lines.

The algorithm for the matching routine above searches for the closest matching centers of all lines in both data sets. Each line can only be assigned one corresponding line of the other set. Since not all lines in the comparison are expected to be found in the atlas or vice-versa, all lines

that cannot be identified are discarded. This process ends with a one-by-one list of comparison and respective atlas lines.

8. Now it uses the table (l_d+/-l_derr) versus (l_wl+/-l_wlerr) to find the wavelength solution by fitting a polynomial to these data. Note that the polynomial is an update to the first-order solution.

9. Steps 7 and 8 iterate to refine the polynomial solution. For each iteration the full set of lines are set back to the original set, so the refined wavelength solution is able to improve the number of matching lines. As the solution gets better the acceptable mismatch shrinks until either the chi-square remains unchanged for at least 3 consecutive iterations, or until it reaches the maximum number of iterations, or until the minimum number of matching lines becomes lower than the limit. All of these limits are input parameters.

10. Then it calculates the final wavelength range covered by the order, the spectral resolution, the radial velocity precision and the wavelength precision.

5.9 operaNormalize

Description

This module simply runs `applyNormalization` routine for each order. This routine first detect the continuum and then normalize the spectrum to one.

Input

- Input spectrum file (*.e)
- Bin size

- Polynomial option and degree of polynomial to fit continuum. Spline is set by default and it should work better for most of the cases.

Algorithm

For each spectral order do:

```
1.applyNormalization(binsize,orderOfPolynomial,usePolynomial,fspecdata,fcontinuumdata, TRUE);
```

where `binsize` is the number of points to bin for continuum measurements. Then `usePolynomial` is a boolean to chose either (as default) to use a spline interpolation or a polynomial of a given degree `orderOfPolynomial` to obtain the continuum model. The options `fspecdata,fcontinuumdata` are output data for plotting, and the last option is a boolean to choose whether or not one wants to overwrite the input spectrum with the normalized one.

The routine `applyNormalization` calls the `normalizeSpectrum` routine for each beam spectrum and also for the master spectrum. This routine generates the normalized spectrum using the algorithm below:

1. Measure continuum using the `measureContinuum` routine

2. Divide each data point in the spectrum by the continuum obtained in step

The algorithm for `measureContinuum` is explained below:

1. First divide spectrum into N samples, where N is calculated as the total number of points divided by the bin size.

2. For each sample do the following:

- Obtain a robust linear fit with `ladfit` using the current sample points plus the two neighbor sets of `binsize` number of points. For first and last

samples it considers the two following samples and the two preceding samples, respectively.

- Calculate the residuals
- Remove points for which the residual is greater than the absolute median deviation.
- Perform robust linear fit again using only the remaining points selected above.
- Sort residuals
- Go over the first half highest ranked residuals and pick the first highest positive value lower than `nsigcut*abdevm`, where `abdevm` is the median absolute deviation and `nsigcut` is an input threshold.
- Set the shift as the picked residual. If no value was picked, which means all searched values are larger than `nsigcut*abdevm`, then set the shift as `nsigcut*abdevm`.
- Apply the shift to the robust fit and set the continuum point as

y_i = `b*x_i + a + shift`

where `a` and `b` are obtained from the robust fit, x_i is the middle point within the bin and `shift` is obtained as described above.

The result is a set of sample data points for the continuum. Note that the bin size is a key parameter for normalization. Since each object spectrum may contain different characteristics, such as broad band absorption features or emission lines, the most appropriate bin size will depend on the nature of the source.

3. Now that we have a vector of sample values for the continuum, the next step is to produce a model for the continuum. There are two options available. First is set by default which is to obtain the continuum values at any point of the spectrum by using a cubic spline interpolation, which can be performed using the routine `operaFitSpline`. The other option is to use a polynomial to fit the sample points. The degree of the fit polynomial is also an input.

5.10 operaFluxCalibration

Description

This module generates the flux calibration product (fcal), which may be used in the final step of an OPERA reduction to generate flux in physical units. As it is known, the continuum transmission of ESPaDOnS optical fibers can be variable at significant levels that would discourage any attempt to obtain an accurate measurement of the continuum flux calibration. However, a pipeline with the capability to perform flux calibration would allow one to assess not only the flux conversion factor but also the throughput of the instrument. This is a useful tool to inspect the evolution and performance of the instrument.

Input

- Uncalibrated spectrum (*.s)
- Calibrated reference spectrum
- Extraction aperture calibration file (*.aper)
- Exposure time
- Bin size (for continuum detection)

Algorithm

For each spectral order:

1. Read calibrated reference standard spectrum
2. Read uncalibrated standard spectrum
3. Degrade observed uncalibrated spectrum resolution by applying a convolution filter
4. Match sampling for both observed and reference spectra.
5. Calculate flux conversion factor for the entire range
6. Calculate instrument throughput
7. Write results out to a calibration file *.fcal.

5.11 operaPolar

Description

This module calculates the polarimetry measurements. It describes the polarization using the 4 Stokes parameters. It can switch between two different methods to calculate a Stokes parameter: difference or ratio method. It also calculates the null polarization spectrum of each of those methods to detect systematic errors.

For each Stokes parameter, there are 4 exposures, numbered 1,2,3,4. There is an option to use only 2 exposures, in which case there can be no null polarization spectrum calculated.

For each exposure, there is a "perpendicular" beam (E) and a "parallel" beam (A) within each order. The left beam is (E) and the right beam is (A).

Exposure #1 has in its left beam an intensity i1E, and in its right beam, i1A. etc.

Those intensities are the 1D extracted spectra.

Input

- 2 or 4 input spectrum files (*.e).
- The number of exposures to use: 2 or 4.
- The method to use: difference or ratio method.
- The order number to analyze if it's not needed to calculate polarization for all orders.

Algorithm

Read the 2 or 4 spectra.

For each spectral order do:

Read the beams.

Calculate the intensity by adding (E) and (A) beams of each exposures to each other and taking the arithmetic mean over all exposures.

Then, from the Bagnulo et al. 2009 paper and the Donati et al. 1997 paper, here are the formulas to calculate a Stokes parameter (also called "polarization"):

Calculate ratio of beams for each exposure (Eq #12 on page 997 of Bagnulo et al. 2009 paper)

- r1 = i1E / i1A
- r2 = i2E / i2A

- r3 = i3E / i3A
- r4 = i4E / i4A

Difference method

STEP 1 - calculate the quantity Gn (Eq #14 on page 997 of Bagnulo et al. 2009 paper), n being the pair of exposures

- G1 = (r1 - 1.0)/(r1 + 1.0)
- G2 = (r2 - 1.0)/(r2 + 1.0)
- G3 = (r3 - 1.0)/(r3 + 1.0)
- G4 = (r4 - 1.0)/(r4 + 1.0)

STEP 2 - calculate the quantity Dm (Eq #18 on page 997 of Bagnulo et al. 2009 paper), m being the pair of exposures

- D1 = G1 - G2
- D2 = G3 - G4

STEP 3 - calculate the degree of Stokes parameter (Eq #19 on page 997 of Bagnulo et al. 2009 paper), aka "the degree of polarization" P

- P/I = (D1 + D2) / (2.0 * pairOfExposures)

STEP 4 - calculate the difference NULL spectrum (Eq #20 on page 997 of Bagnulo et al. 2009 paper)

- Nd = (D1 - D2) / (2.0 * pairOfExposures)

Ratio method

STEP 1 - calculate the quantity Rm (Eq #23 on page 998 of Bagnulo et al. 2009 paper), m being the pair of exposures

- R1 = r1 / r2
- R2 = r3 / r4

STEP 2 - calculate the quantity R (Part of Eq #24 on page 998 of Bagnulo et al. 2009 paper)

- R = Pow(R1 * R2 , 1.0/(2.0 * pairOfExposures))

STEP 3 - calculate the degree of Stokes parameter (Simplification with STEP 2 of Eq #24 on page 998 of Bagnulo et al. 2009 paper), aka "the degree of polarization" P

- P/I = (R - 1.0) / (R + 1.0)

STEP 4 - calculate the quantity RN (Part of Eq #25-26 on page 998 of Bagnulo et al. 2009 paper)

- RN = Pow(R1 / R2 , 1.0/(2.0 * pairOfExposures))

STEP 5 - calculate the ratio NULL spectrum (Simplification with STEP 4 of Eq #25-26 on page 998 of Bagnulo et al. 2009 paper)

- Nr = (RN - 1.0) / (RN + 1.0)

Finally, calculate the polarization using the intensity and the degree of polarization.

5.12 operaGeometryCalibration

Description

This module obtain the geometry calibration. The geometry calibration include the detection, trace, and enumeration of orders. It produces a *.geom calibration file with the geometry calibration.

Input

- A well-exposed, unsaturated, flat-field master image.
- A master bias image.

- A bad-pixel mask image.
- Selection of one of the three detection methods: (1) gaussian, (2) instrument profile, or (3) top-hat.
- Dispersion direction (either along columns or rows).
- Number for the first order.
- Aperture - slit size in pixel units.
- Bin size - number of pixels to bin in the dispersion direction.

Algorithm

For each spectral order:

1. read in a masterbias and a masterflat.
2. remove the masterbias from the masterflat
3. sample 3 equally spaced regions in the vertical direction and depending on the *detectionmethod* parameter call one of `operaCCDDetectPeaksWithGaussian`, `operaCCDFitIP`, `operaCCDDetectPeaksWithTopHat`, and finally `operaLMFitPolynomial` to determine a polynomial that describes the spacing between orders across the CCD.
4. Set the aperture width and binsize from the input parameters, moving from x=x1 incrementing by nx, and the y bin size, create bins of pixel values, detect peaks using the above mentioned library functions. Call the library function `traceOrder` to calculate the polynomial which describes each order (See the `operaGeometry` class methods).
5. Write out the geometry polynomials and the order spacing polynomial.

5.13 operaInstrumentProfileCalibration

Description

The module `operaInstrumentProfileCalibration` performs measurements of the instrument illumination profile (IP) on all orders defined in the geometry calibration file *.geom. The module saves the results into the calibration file *.prof. The results are saved as a sub-pixel sampled matrix of polynomial models.

Input

- A bias master image.
- A well-exposed, unsaturated, flat-field master image.
- A well-exposed, comparison lamp (ThAr) master image.
- A well-exposed align Fabry-Perot image.
- A bad-pixel mask image.
- Aperture - slit size in pixel units.
- IP dimensions - x-size (in pixels), x-sampling, y-size (in pixels), and y-sampling.

Algorithm

The `operaInstrumentProfileCalibration` module executes as follows:

Load images: masterbias, masterflat, mastercomparison, masterfabryperrot (if they exist), and the badpixelmask

Subtract bias from the masterflat, mastercomparison, and masterfabryperot images.

Load the geometry calibration from *.geom

For each order do the following {

Initialize the spectral order

Set up the aperture

Calculate the order length in pixel units

Set the size of the spectral elements

Initialize IP with dimensions (x-size, y-size) in pixels, and pixelation (x-sampling, y-sampling). The pixelation gives the sub-pixel sampling, or spatial resolution, for which IP will be determined.

Run routine `measureInstrumentProfileAlongRows`, which uses a master flat-field exposure to obtain measurements of the 1D IP along rows (or columns). The algorithm for this routine is discussed in the following section.

At this point the IP measurements should suffice if the extraction to follow is made along the detector rows. For ESPaDOnS, since the pseudo-slit is almost aligned with the detector rows, this IP should already be very close to the real spatial profile. Therefore, the 1D IP provides a reasonably good estimate for the weight function to be used in extraction.

However, the actual pseudo-slit on ESPaDOnS is slightly tilted and of irregular shape, caused by the image slicer. Therefore, a more accurate measurement of the spatial profile is only possible by considering the full extent of a 2D IP, which is done by performing the steps below.

Call the routine:

`measureInstrumentProfileAlongRowsInto2DWithGaussian`

This routine replaces `measureInstrumentProfileAlongRows` and in fact it does almost the same thing, except that it creates a 2D IP using the measurements along rows and a gaussian model along columns. This model can also include a tilt angle for the slit. This IP is then used to detect and measure the actual 2D profile from sharp spectral lines.

Call the routine `measureInstrumentProfile`. This method uses a previously measured IP as initial guess for the 2D IP. Then it takes either a comparison lamp or a Fabry-Perot image to detect spectral lines, from which the measurements will be performed. These provide information in two-dimensions, allowing an empirical determination of the actual 2D IP. The algorithm for this routine is also discussed in the following section.

}

Finally the module saves the IP model into a *.prof calibration file.

Methods

1. `measureInstrumentProfileAlongRows`

Input

- master Flat-field image (FITS image)
- Badpixel mask (FITS image)
- Binsize (number of pixels to bin)

Algorithm

Read master flat-field image.

Enforce IP y-size = 1 pixel and y-sampling = 1. This is necessary because the routine measures the IP along the rows.

Calculate the number of bins based on the binsize and the number of rows to be binned.

For each bin do {

Initialize the IP data matrix

For each IP x-position do {

Calculate median flux value for all rows in the bin

Calculate errors

Sum median values into a normalization factor

}

Calculate IP, which is given by the median flux value at a given x-position divided by the normalization factor.

Save IP data into a data cube

}

Fit a low order polynomial for each sub-pixel of the IP to the entire order. The polynomial variable is given by the distance in the dispersion direction in pixel units.

2. `measureInstrumentProfileAlongRowsInto2DWithGaussian`

Input

- master Flat-field image (FITS image)
- Badpixel mask (FITS image)

- Binsize (number of pixels to bin)
- Sigma (PSF in pixel units)
- Tilt (angle in degrees)

Algorithm

Read the master flat-field image

Calculate the number of bins based on the binsize and the number of rows to be binned.

For each bin do {

    Initialize IP data matrix

    For each IP x-position do {

        Calculate median flux value for all rows in the bin

        Calculate errors

        Sum median values into a normalization factor

    }

    Calculate 1D IP, which is given by the median flux value at a given x-position divided by the normalization factor

    Create the 2D Instrument Profile from the measured 1D Instrument Profile along rows plus a Gaussian model and tilt along columns. This is done as follows:

    For IP y points do {

        get y-coordinate (y) in the IP reference frame

        For IP x points do {

```
                get x-coordinate (x) in the IP reference frame
                y0 = - tan(tilt) * (x)
                2D_IP = (1D_IP) * Gaussian(y - y0, sigma)
                save 2D_IP into data cube
            }
        }
}
```

Fit a low order polynomial for each sub-pixel of the IP to the entire order. The polynomial variable is given by the distance in the dispersion direction in pixel units.

3. `measureInstrumentProfileImage`

Input

- master comparison lamp or master Fabry-Perot (FITS image)
- Badpixel mask (FITS image)
- Detection threshold

Algorithm

Read the master comparison or Fabry-Perot image

Call the method `detectSpectralLinesUsingIP`. This method first calculates a cross-correlation between the IP and the image along the order. Then it detects spectral lines, using the peak-probability method, similar to the one previously used in Geometry calibration. The output of this function is a vector with x and y center positions and flux for the set of detected lines.

For each detected line do {

Measure 2D IP

Filter out mistaken or/and blended lines

Save to data cube

}

Fit a low order polynomial for each sub-pixel of the IP to the entire order. The polynomial variable is given by the distance in the dispersion direction in pixel units.

Re-run `detectSpectralLinesUsingIP`. Given that the IP measurements have been improved the detection routine is expected to be more efficient and therefore be able to find more lines.

Filter out mistaken or/and blended lines.

Measure 2D IP from detected lines and save to data cube.

Re-fit IP polynomial model.

Evaluate quality of fit and re-iterate if necessary.

4. `detectSpectralLinesUsingIP`

Input

master comparison lamp or master Fabry-Perot (FITS image)

Badpixel mask (FITS image)

Algorithm

Read the master comparison or master Fabry-Perot image

For each spectral element do {

Reject bad-pixels and saturated pixels.

estimate scattered light and remove it from image data

calculate cross-correlation between comparison or Fabry-Perot image and 2D IP

}

For each spectral element do {

calculate peak probability (see geometry).

if (peak probability > threshold) then save x,y photo-center and flux

}

5.14 operaWavelengthCalibration

Description

The module `operaWavelengthCalibration` obtain measurements of spectral line positions on a comparison lamp exposure and matches those lines with a atlas to obtain the pixel-to-wavelength solution for each spectral order. The module saves the results into the calibration file *.wave.

Input

- Master bias FITS image
- Master comparison FITS image
- opera geometry calibration file (*.geom)
- opera instrument profile calibration file (*.prof)
- Atlas of spectral lines

Algorithm

For each spectral order:

1. Read ThAr raw spectrum; intensity versus distance in pixel units:

   I(d) vs. d

2. Measure total distance "D" (in pixel units) covered by the order. This is given by the line integral of the polynomial that describes the center of the order.

3. Read wavelength range covered by the order: wl0, wlf

4. Calculate first order solution:

   wl = f(d), where f(d) as first order is given by

f(d) = wl0 + ((wlf - wl0)/D)*d

assuming f(d=0) = wl0.

5. Read ThAr atlas of spectral lines within the range covered by the order [wl0:wlf]. The atlas consists of line wavelength (l_wl), error (l_wlerr), and line relative intensity (l_i).

6. Detect lines in ThAr raw spectrum. The detection can make use of the opera function operaCCDDetectPeaksWithErrorsUsingGaussian.

Note that dist_pixels is the distance along the center of the order.

7. Once there is a table of l_i, l_d, and l_derr, then calculate the maximum cross-correlation between this and the atlas data to identify the lines. Not all lines in the comparison exposure will be found in the atlas, and not all atlas lines will be found in the comparison exposure. Some cleanup will be needed.

8. Now use the table (l_d+/-l_derr) versus (l_wl+/-l_wlerr) to find the wavelength solution by fitting a polynomial to these data. Note that the polynomial is an update to the first-order solution. Iterate refining the higher orders of the polynomial with the best fit.

5.15 operaPixelSensitivityMap

Description

This module produces a pixel-by-pixel sensitivity map (normalized flat-field) image for an echelle spectral image.

Input

- Master bias FITS image
- Master flat-field FITS image
- opera geometry calibration file (*.geom)
- Aperture in pixels
- Binsize in pixels.

Algorithm

Load images: masterbias and masterflat.

Subtract bias from flat.

Set pixel values of output image to 1.

Load order geometry information from *.geom

For each order do the following:

Initialize spectral order

Set up aperture and sampling

Call routine `NormalizeFlat`, which uses a master flat-field exposure to obtain measurements of the pixel-by-pixel sensitivity variations on the illuminated regions of the detector. This function assumes a smooth flat-field spectrum.

Save output image, which contains the normalized pixel-by-pixel sensitivity map for the regions defined by the aperture size around order tracks, which are defined in *.geom. The map assigns value 1 for undefined regions.

Methods

NormalizeFlat

This method perform the normalization of a flat-field exposure.

Input

- Flat-field image data.
- Nx, Ny: image dimensions.
- Binsize: number of pixels to bin in the dispersion direction.

Algorithm

For each image row do {

For each IP x-position (column number defined by a sub-pixel sampled IP) do {

Calculate median of pixel values from rows in the range:

[currenRow - binsize/2, currentRow + binsize/2]

}

Calculate the expected value for each pixel in the row by evaluating a cubic interpolation from the median data collected above.

Divide flat-field image pixel values by the expected pixel value.

Save values into output image.

}

5.16 operaOptimalExtraction

Description

The `operaOptimalExtraction` module loads the calibration files (*.geom, *.prof, *.wave) and extracts the energy flux information from a given input “object” image.

Input

- Master bias FITS image
- Pixel sensitivity map FITS image
- Bad pixel map FITS image
- Object FITS image
- opera geometry calibration file (*.geom)
- opera instrument profile calibration file (*.prof)
- opera wavelength calibration file (*.wave)
- Spectral element width
- Aperture for object extraction
- Aperture for sky extraction

Algorithm

The `operaOptimalExtraction` module executes as follows:

Load object data (FITS image)

Load master bias data (FITS image)

Load pixel sensitivity map data (FITS image)

Load badpixel mask data (FITS image)

Subtract bias from object

Divide (object-bias) by pixel sensitivity map

Load geometry calibration from *.geom

Load instrument profile calibration from *.prof

Load wavelength calibration from *.wave

Load spectral order vector

Call `extractBackgroundLight`. This routine does the following:

For each order do {

Call `createBackgroundLightMask`. This function uses the IP and geometry calibration to find out which pixels in the object image the background light are the dominant source of flux. Then it creates a mask of these pixels for background light measurements.

Increment the mask (starts as a copy of bad-pixel mask)

}

Mask out non-background pixels in object image

Bin data. This step outputs a set of vectors with the following data: x, y, flux, flux variance.

Fit 2D spline to both background flux data and variances.

Resample background flux and variance with same sample as object image.

Output background light image and variance image.

For each order do {

Set spectral elements and define extraction regions.

Call `extractOptimalSpectrum`. This routine measures the flux and associated uncertainties using each one of the first three methods presented above. It saves the extracted spectrum into the `operaSpectralElement` member class.

Call `extractOperaoptimalSpectrum`. This routine implements extraction using the fourth method presented above. The algorithm for this method is discussed in the following section. It also saves the extracted spectrum into the `operaSpectralElement` member class.

}

Save calibrated spectrum into opera products

Methods

- `extractOptimalSpectrum`

Description

This routine implements each one of the first three extraction methods for extraction presented above. The algorithm follows below.

Algorithm

1. Load calibration, parameters and image data

2. For each spectral element do {

Extract raw sum flux

Calculate variance of raw sum flux. Include detector noise and photon noise.

if (sky) {

Measure sky (S). This feature is important for long slit spectrographs.

}

}

3. if (sky) {

Fit sky flux.

}

4. For each spectral element do {

Extract standard spectrum = object flux (O), which is given by the raw flux (R) minus (sky+background) flux (S+B); O = R - S - B

Calculate the variance of the standard spectrum. Include contributions from detector noise, photon noise, sky flux variance, and background variance.

}

5. For each spectral element do {

Measure instrument profile from object image.

}

6. Fit object IP. Compare with IP from calibration.

7. For each spectral element do {

    Revise variance based on flux and IP

    Mask cosmic rays and outliers

    Extract optimal spectrum

    Calculate variance of optimal spectrum

}

Iterate steps 5 through 7.

- `extractOperaOptimalSpectrum`

Description

This method extends the fundamental ideas of the optimal extraction using a 2D IP for the extraction. The theory for this implementation is presented in a separate section below. The algorithm for opera Optimal Extraction is as follows:

Algorithm

1. Load calibration, parameters and image data

2. if (sky) {

    Fit sky

}

3. Subtract (sky + background) from object data

4. Extract optimal spectrum using IP from calibration

5. Use optimal spectrum and IP from calibration to mask out cosmic ray hits or pixels with high noise. The method presented here takes every pixel as an individual linear estimator that may contribute to the flux of many spectral elements. This entanglement between pixels and the spectrum is removed by solving a system of linear equations. Therefore, the contribution of outliers to the noise also propagates to many elements. Therefore in order to mitigate this effect it is important that bad pixels are efficiently masked out.

6. Build vector of pixels where the spectrum will be extracted from

7. Build IP super-matrix from IP calibration data

8. Extract opera optimal flux by solving matrix equation

5.17 operaTelluricWavelengthCorrection

Description

The module `operaTelluricWavelengthCalibration`

Input

- Master bias FITS image
- Algorithm

For each spectral order:

5.18 operaHelioCentricWavelengthCorrection

Description

The module `operaHeliocentricCalibration`

Input

- Master bias FITS image

- Algorithm

For each spectral order:

6. Software Libraries

A software library consists of a group of functions that can be linked to an executable program (module or tool). For example, there will be OPERA modules that read an image in FITS format. These modules may use the CFITSIO library for basic I/O, but the OPERA library interfaces provide the additional benefit of a more abstract operaImage datatype interface. OPERA has internal libraries that will be developed as part of the project. All external libraries used by OPERA must be open-source and must not have licensing restrictions that would enforce a licensing modification on OPERA. The latter is a consequence of the OPERA technical requirements (see OPERA documentation).

All libraries should be written as re-entrant code but not use threads. Modules also should not use threading, as previously noted, since we cannot guarantee whether an installation will have a re-entrant or non-re-entrant CFITSIO library installed.

We opted to write all OPERA software libraries in the C language. While we lose some of the powerful object properties and exception handling of C++, C libraries may be called from either C or C++ and the converse is not true. That said, we will also develop a C++ wrapper for the libraries This wrapper will call the C functions and make use of the C++ Exception facility to handle errors. Classes will also be written to mirror the structs used in the C library interfaces. The C++ interface will add a number of Exception types to the existing set of C error codes.

Thus instead of the C style:

```
if ((errorCode=operaLibCall())) {
    return errorCode;
}
if ((errorCode=operaLibCall2())) {
    ...
}
```

The C++ style would be:

```
try {
    operaLibCall();
    operaLibCall2();
} catch (operaException::exception) {
...
}
```

In general the image and statistics libraries are written in C. Any libraries written in C should return error codes from every function. A C library should not print anything and should not EXIT(). Any libraries or modules or classes written in C++ should check return codes and throw an exception on error. The operaFITSImage class, for example, checks cfitsio status returns and throws an exception on error. The operaException class has a constructor that includes a message, file, function and line number. This is the preferred constructor as it allows us to quickly locate an error.

The error codes and text tables associated with each code will be stored in a common operaError.h to ensure there is no overlap between modules/ libraries.

Numeric libraries should check for NaNs and return an error code.Callers of image libraries should also check for NaNs using the builtin isnan() function.

The library functions for some commonly used “high profile” calls, such as quicksort for finding a median for example, come in various flavors:

i. nondestructive function call

ii. inline function call (for speed)

iii. destructive function call (for speed)

Why all the different flavors? Well, inlined functions are faster than called functions, but they take more space as the code is inlined at every call site. Image median is an example of a function with a side effect. It uses quicksort to find the median pixel value, and quicksort has the side effect of sorted the image pixels. This is not what the users might expect the median function to do. So, a copy of the image is made first, then the copy is sorted. However there are times when the side effect is not a problem (median stacking for example, where the input are never saved). In this case, to make median stacking fast enough, the caller may opt for the destructive, inlined version, the fastest flavor.

6. 1. Parameter Access Library

Name: libOperaParameterAccess

This library accesses the parameter file, which stores instrument parameters. The entries consist of `name := value [list]` entries. The parameter file is a restricted form of configuration access file, in that

1. Variables are not supported.
2. Line is restricted to 80 characters. The routines take a `const char* name` as the first parameter. The second parameter is a `char* &value` list in the case of a "get", or a `const char*` in the case of a "set", "add" or "delete". "remove" removes the entire `name`/`value` entry.

Functions:

a. operaParameterAccessSetParameterFilepath

This function sets the static `filepath` for a module calling this library. Modules should really only use the default except under extenuating circumstances.

Prototype: `operaParameterAccessSetParamaterFilepath(const char *filepath);`

Return type: `operaErrorCode`

Parameters:

- `filepath` is a char pointer to the file path or NULL to set to default.

b. operaParameterAccessGet

This function gets a value [list] for a given name. Note that this function allocates storage, which must be freed by the caller.

Prototype: `operaParameterAccessGet(const char *name, char **value);`

Return type: either an `operaErrorCode` or `value` = value of name or NULL if not known function

Parameters:

- `name` is a char pointer to the name.
- `value` is a char pointer address to the value [list]

c. operaParameterAccessSet

This function sets a value for a given name.

Prototype: `operaParameterAccessSet(const char *name, const char *value);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name.
- `value` is a char pointer address to the value [list]

d. operaParameterAccessAdd

This function adds a name value list entry.

Prototype: `operaParameterAccessAdd(const char *name, const char *value);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name
- `value` is a char pointer to the value [list]

e. operaParameterAccessDelete

This function deletes a value from a given name.

Prototype: `operaParameterAccessDelete(const char *name, const char *value);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name
- `value` is a char pointer to the value [list]

f. operaParameterAccessRemove

This function removes a name value entry.

Prototype: `operaParameterAccessRemove(const char *name);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name

6. 2. Configuration Access Library

Name: libOperaConfigurationAccess

This library accesses the Configuration file, which stores instrument Configurations. The entries consist of `name := value [list]` entries. The Configuration file may contain comments, and make-style variables and the \ character at end of line signifying continuation.

The routines take a `const char* name` as the first Configuration. The second parameter is a `char* &value` list in the case of a "get", or a `const char*` in the case of a "set", "add" or "delete". "remove" removes the entire `name`/`value` entry.

Functions:

a. startsWith

This function returns an integer > 0 if `aString` starts with `substring`.

Prototype: `startsWith(const char *aString, const char *substring);`

Return type: `static int EXIT_STATUS`

Parameters:

- `aString` - the string to search
- `substring` - the snippet

b. operaConfigurationAccessSetConfigurationFilepath

This function sets the static `filepath` for a module calling this library. Modules should really only use the default except under extenuating circumstances.

Prototype: `operaConfigurationAccessSetConfigurationFilepath(const char *filepath);`

Return type: `operaErrorCode`

Parameters:

- `filepath` is a char pointer to the file path or NULL to set to default.

c. operaConfigurationAccessGet

This function gets a value [list] for a given name. Note that this function allocates storage, which must be freed by the caller.

Prototype: `operaConfigurationAccessGet(const char *name, char **value);`

Return type: either an `operaErrorCode` or `value` = value of name or NULL if not known function

Parameters:

- `name` is a char pointer to the name.
- `value` is a char pointer address to the value [list]

d. operaConfigurationAccessSet

This function sets a value for a given name.

Prototype: `operaConfigurationAccessSet(const char *name, const char *value);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name.
- `value` is a char pointer address to the value [list]

e. operaConfigurationAccessAdd

This function adds a name value list entry.

Prototype: `operaConfigurationAccessAdd(const char *name, const char *value);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name
- `value` is a char pointer to the value [list]

f. operaConfigurationAccessDelete

This function deletes a value from a given name.

Prototype: `operaConfigurationAccessDelete(const char *name, const char *value);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name
- `value` is a char pointer to the value [list]

g. operaConfigurationAccessRemove

This function removes a name value entry.

Prototype: `operaConfigurationAccessRemove(const char *name);`

Return type: `operaErrorCode`

Parameters:

- `name` is a char pointer to the name

6. 3 Statistics Library

Name: libOperaStats

This library contains the main functions to obtain statistical quantities from arrays.

Function prototypes

```
float operaArrayMean(unsigned np, const float *array);
float operaArrayWeightedMean(unsigned np, const float *array, const float *weigh);
float operaArrayAvgSigClip(unsigned np, const float *array, unsigned nsig);
float operaArraySig(unsigned np, const float *array);
float operaArrayWeightedSig(unsigned np, const float *array, const float *weigh);
float operaArrayMedian(unsigned np, const float *array);
float operaArrayMedSig(unsigned np, const float *array, float median);
float operaArrayMaxValue(unsigned np, const float *array);
float operaArrayMinValue(unsigned np, const float *array);
void  operaArrayHeapSort(unsigned np, float *arr);
void  operaArrayIndexSort(unsigned np, const float *array, unsigned *sindex);
float operaUniformRand(float xcen, float xmax, float xmin);
float operaGaussRand(float xcen, float sig);
unsigned operaCountPixels(unsigned np, const float *array, float minvalue, float max-
value);
```

Inline function prototypes

```
inline float operaArrayMeanQuick(unsigned np, const float *array);
inline float operaArrayWeightedMeanQuick(unsigned np, const float *array, const float
*weigh);
inline float operaArraySigQuick(unsigned np, const float *array);
inline float operaArrayWeightedSigQuick(unsigned np, const float *array, const float
*weigh);
inline float operaArrayAvgSigClipQuick(unsigned np, const float *array, unsigned
nsig);
inline float operaArrayMedianQuick(unsigned np, float *arr);
inline unsigned short operaArrayMedianQuickUSHORT(unsigned np, unsigned short *arr);
inline unsigned short operaArrayMedianQuickUSHORT(unsigned np, unsigned short *arr);
inline void operaArrayIndexedMeanQuick(unsigned np, const float *array, const float
*indexmask, unsigned nb, float *meanbin) {
inline void operaArrayIndexedSigQuick(unsigned np, const float *array, const float
*indexmask, unsigned nb, float *sigbin) {
```

Function parameters

```
unsigned np - number of elements in array/weigh/arr
const float *array - static float pointer to data values
const float *weigh - static float pointer to weigh data values (between 0 and 1)
unsigned nsig - size which data is clipped in sigma units
float median - provided median
float *arr - dynamic float pointer to data values
float xcen - central value
float xmax - upper limit
float xmin - lower limit
```

```
float sig - sigma of a Gaussian distribution
float minvalue - minimum value
float maxvalue - maximum value
const float *indexmask - static float pointer to index mask
unsigned nb - number of bins
float *meanbin - float pointer to array of nb-mean values
float *sigbin - float pointer to array of nb-sigma values
```

Functions

a. operaArrayMean

This function calculates the arithmetic mean <x> of array {x} with N elements.

Return: `float` mean = <x>

Parameters: `unsigned np = N, const float *array = {x}`

Algorithm:

$$\langle x \rangle = \frac{\sum_{i=0}^{N-1} x_i}{N}.$$

b. operaArrayWeightedMean

This function calculates the weighted mean <x> of array {x} with N elements, in which each element is weighted by the elements given in array {w}.

Return: `float` weighted Mean = <x>

Parameters: `unsigned np = N, const float *array = {x}, const float *weigh = {w}`

Algorithm:

$$\langle x \rangle = \frac{\sum_{i=0}^{N-1} x_i w_i}{\sum_{i=0}^{N-1} w_i} .$$

c. operaArrayAvgSigClip

This function calculates the average <x'> of an array {x'}, which is obtained from the cut of an original array {x} with N elements. The cut is made for values outside the range <x> ± n_σ x σ, where σ is the standard deviation of the original array.

Return: `float` AvgSigClip = <x'>

Parameters: `unsigned np = N, const float *array = {x}, unsigned nsig = n`$_\sigma$

Algorithm:

$$\langle x' \rangle = \frac{\sum_{i=0}^{N-1} x_i w_i}{\sum_{i=0}^{N-1} w_i}$$, where the following condition applies

if $|x_i - \langle x \rangle| \leq n_\sigma \sigma$ **then** {

$$w_i = 1$$

} **else then** {

$$w_i = 0$$

}

for σ being defined as the standard deviation of the original array about the mean (see operaArraySig).

d. operaArraySig

This function calculates the standard deviation σ about the mean <x> of array {x} with N elements.

Return: `float` sig = σ

Parameters: `unsigned np = N, const float *array = {x}`

Algorithm:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1} \left(x_i - \langle x \rangle\right)^2}{N}}.$$

d. operaArrayWeightedSig

This function calculates the weighted standard deviation σ about the mean <x> of array {x} with N elements, in which each element is weighted by the elements given in array {w}.

Return: `float` sig = σ

Parameters: `unsigned np = N, const float *array = {x}, const float *weigh = {w}`

Algorithm:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1} w_i \left(x_i - \langle x \rangle\right)^2}{\sum_{i=0}^{N-1} w_i}}.$$

e. operaArrayMedian

This function non-destructively finds the median x_M of an array {x} with N elements.

Return: `float` median = x_M

Parameters: `unsigned np = N, const float *array = {x}`

Algorithm:

First sort the orginal array {x}:

$$\{x'\} = \mathrm{SORT}\{\mathrm{x}\}$$

Then the median is calculated as follows

if (N is odd) **then** {

$$x_M = x'_{(N-1)/2}$$

} **else if** (N is even) **then** {

$$x_M = \frac{x'_{(N-2)/2} + x'_{N/2}}{2}$$

}

Note that the index in array {x} ranges from 0 to N-1.

f. operaArrayMedSig

This function finds the median deviation σ_M from a given central value x_M of an array {x} with N elements.

Return: `float` medsig = σ_M

Parameters: `unsigned np = N, const float *array = {x}, float median =` x_M

Algorithm:

Build an array of the deviations, where each element is calculated as the absolute value of the difference between the element of the original array and the given central value x_M,

$\{\sigma\} = \{x - x_M\}$

Then find the median of array {σ} and divide result by 0.674433,

```
operaArrayMedian(np,*sigarray) / 0.674433;
```

The denominator comes from the fact that when one calculates the median of deviations, this is in fact the two-quartile sigma of the distribution, which can be related to the sigma of the normal distribution σ(68.27%) as follows

σ(50%) = 0.674433 σ(68.27%)

Hence the statistical meaning of the σ given by this function becomes consistent with the σ obtained in the previous functions.

g. operaArrayMaxValue

This function finds the maximum value of an array {x} with N elements.

Return: `float` maxvalue = x_{max}

Parameters: `unsigned np = N, const float *array = {x}`

Algorithm:

Parse all elements in the array and return the one with maximum value x_{max}.

h. operaArrayMinValue

This function finds the minimum value of an array {x} with N elements.

Return: `float` minvalue = x_{min}

Parameters: `unsigned np = N, const float *array = {x}`

Algorithm:

Parse all elements in the array and return the one with minimum value x_{min}.

i. operaArrayHeapSort

This function destructively heap sorts the elements of an array {x} with N elements.

Return: `void`

Parameters: `unsigned np = N, float *arr = {x}`

Algorithm: Heapsort algorithm (Press, W. H. et al. 1992).

j. operaArrayIndexSort

This function constructs an index array `{j}` that ranks the elements of an array `{x}` in increasing order (it doesn't change the original array). Note that the indexes range from 0 to N-1.

Return: `void`

Parameters: `unsigned np = N, const float *array = {x}, unsigned *sindex = {j}`

Algorithm: Indexing and ranking (Press, W. H. et al. 1992)

k. operaUniformRand

This function produces a random number x between x_c - x_{min} and x_c + x_{max}, with uniform probability distribution.

Return: `float` x

Parameters: `float xcen =` x_c, `float xmin =` x_{min}, `float xmax =` x_{max}

Algorithm:

$x = (x_c - x_{min}) + (x_{max} + x_{min})*$`(random()/RAND_MAX)`

where `random()` is an inbuilt C function to generate an integer random number between 0 and `RAND_MAX`. Therefore `RAND_MAX` is the normalization factor.

l. operaGaussRand

This function produces a random number x with normal distribution, given the mean x_c and the standard deviation σ.

Return: `float` x

Parameters: `float xcen =` x_c, `float sig =` σ

Algorithm:

First generate two independent random numbers between 0 and 1 with uniform distribution:

```
x1 = random()/RAND_MAX;
```

x_2 = random()/RAND_MAX;

Then use these number to produce a random number with normal distribution as follows

y_1 = sqrt(- 2 * log(x_1)) * cos(2 * π * x_2);
y_2 = sqrt(- 2 * log(x_1)) * sin(2 * π * x_2);

x = x_c + σ*(y1+y2)/2;

m. operaCountPixels

This function counts the number of elements n_e in an array {x} which are lying between a given minimum value x_{min} and maximum value x_{max}.

Return: unsigned count = n_e

Parameters: unsigned np = N, const float *array = {x} , float xmin = x_{min}, float xmax = x_{max}

Algorithm:

Initialize n_e = 0.

Parse all elements in array {x} and count elements x_i that follows the condition

if (x_{min} < x_i < x_{max}) { increment n_e }

n. operaArrayIndexedMeanQuick

This function calculates the arithmetic mean {<x>} for an array of N_b clusters of elements in array {x}. The elements are clustered by their index number j given in the index mask {m}.

Return: void

Parameters: `unsigned np = N, const float *array = {x}, const float *indexmask = {m},unsigned nb = Nb,float *meanbin = {<x>}`

Algorithm:

for k=1 to k=N_b **do** {

$$\langle x \rangle_k = \frac{\sum_{i=0}^{N-1} x_i w_i}{\sum_{i=0}^{N-1} w_i}$$, where the following condition applies

if (m_i = k) **then** {

$$w_i = 1$$

} **else then** {

$$w_i = 0$$

}

}

o. operaArrayIndexedSigQuick

This function calculates an array of standard deviations {σ} about their respective mean values {<x>} for N_b clusters of elements in array {x}. The elements are clustered by their index number j given in the index mask {m}.

Return: `void`

Parameters: `unsigned np = N, const float *array = {x}, const float *indexmask = {m}, unsigned nb = Nb,float *sigbin = {σ}`

Algorithm:

for k=1 to k=N_b **do** {

$$\langle\sigma\rangle_k = \sqrt{\frac{\sum_{i=0}^{N-1} w_i \left(x_i - \langle x\rangle_k\right)^2}{\sum_{i=0}^{N-1} w_i}},$$ where the following condition applies

if (m_i = k) **then** {

$$w_i = 1$$

} **else then** {

$$w_i = 0$$

}

}

5.4 Image Library

Name: libOperaImage

This library contains the functions to obtain statistical quantities from or to perform arithmetic operations on image arrays.

Function prototypes

```
void operaImMean(unsigned depth, long npixels, float *master, float *arrays[]);
void operaImWeightedMean(unsigned depth, long npixels, float *master, float *arrays[],
float *weights[]);
void operaImSig(unsigned depth, long npixels, float *sigarray, float *arrays[], float
*master);
void operaImWeightedSig(unsigned depth, long npixels, float *sigarray, float *ar-
rays[], float *weights[], float *master);
void operaImAvgSigClip(unsigned depth, long npixels, float *master, float *arrays[],
unsigned nsig);
float operaCCDVarDiff(unsigned depth, long npixels, float *arrays[], float *weight);
void operaImVarDiff(unsigned depth, long npixels, float *arrays[], float *diffvarimg);
float *medianCombineFloat(unsigned depth, long npixels, float *output, float
*images[]);
unsigned short *operaArrayMedianCombineUSHORT(unsigned depth, long npixels, unsigned
short *output, unsigned short *images[]);
```

Inline function prototypes

```
inline void operaImMeanQuick(unsigned depth, long npixels, float *master, float *ar-
rays[])
inline void operaImWeightedMeanQuick(unsigned depth, long npixels, float *master,
float *arrays[], float *weights[])
inline void operaImSigQuick(unsigned depth, long npixels, float *sigarray, float *ar-
rays[], float *master)
inline void operaImWeightedSigQuick(unsigned depth, long npixels, float *sigarray,
float *arrays[], float *weights[], float *master)
inline void operaImAvgSigClipQuick(unsigned depth, long npixels, float *master, float
*arrays[], unsigned nsig)
```

Image arithmetic inline functions

```
inline void operaSumImbyConstant(long npixels, float *img, float number){while (npix-
els--) *img++ += number;}
inline void operaSubtractImbyConstant(long npixels, float *img, float number){while
(npixels--) *img++ -= number;}
inline void operaMultiplyImbyConstant(long npixels, float *img, float number){while
(npixels--) *img++ *= number;}
inline void operaDivideImbyConstant(long npixels, float *img, float number){while
(npixels--) *img++ /= number;}
inline void operaImSubtractIm(long npixels, float *img1, float *img2) {while (npix-
els--) *img1++ -= *img2++;}
```

Function parameters

```
long npixels - number of elements in an image array
float *arrays[] - set of image arrays
float *weights[] - set of weigh arrays
unsigned depth - number of arrays in a set
float *master - output master image array
float *sigarray - input/output sigma array
float *diffvarimg - output variance array
float *img - image array
float number - number to operate an image array
unsigned nsig - size which data is clipped in sigma units
```

Functions

a. operaImMean

This function mean combines a set of image arrays `*arrays[]` into the master image array `*master`.

Return: `void`

Parameters: `unsigned depth = N, long npixels = M, float *master = {m}, float *arrays[] = {{p}1, {p}2,...,{p}N}`

Algorithm:

Each element m_i of the master image array is calculated as follows:

$$m_i = \frac{\sum_{j=0}^{N-1} p_{ij}}{N},$$

where `j-index` is used for members of the set of images and `i-index` for elements of image arrays.

b. operaImWeightedMean

This function weigh-mean combines a set of image arrays `*arrays[]`, with weights `*weights[]`, into the master image array `*master`.

Return: `void`

Parameters: `unsigned depth = N, long npixels = M, float *master = {m}, float *arrays[] = {{p}0, {p}1,...,{p}N-1}, float *weights[] = {{w}0, {w}1,...,{w}N-1}`

Algorithm:

Each element m_i of the master image array is calculated as follows:

$$m_i = \frac{\sum_{j=0}^{N-1} p_{ij} w_{ij}}{\sum_{j=0}^{N-1} w_{ij}} .$$

c. operaImSig

This function calculates the standard deviation image `*sigarray` from a set of images arrays `*arrays[]` with respect to a master image array `*master`.

Return: `void`

Parameters: `unsigned depth = N, long npixels = M, float *sigarray = {σ},float *arrays[] = {{p}0,{p}1,...,{p}N-1}, float *master = {m}.`

Algorithm:

Each element σ_i of the standard deviation image array {σ} is calculated as follows:

$$\sigma_i = \sqrt{\frac{\sum_{j=0}^{N-1} (p_{ij} - m_i)^2}{N}} .$$

d. operaImWeightedSig

This function calculates the weighted standard deviation image `*sigarray` from a set of image arrays `*arrays[]`, weighted by the set of weigh arrays `*weights[]`, with respect to a master image array `*master`.

Return: `void`

Parameters: `unsigned depth = N, long npixels = M, float *sigarray = {σ},float *arrays[] = {{p}0,{p}1,...,{p}N-1}, float *weights[] = {{w}0,{w}1,...,{w}N-1},float *master = {m}.`

Algorithm:

Each element σ_i of the standard deviation image array {σ} is calculated as follows:

$$\sigma_i = \sqrt{\frac{\sum_{j=0}^{N-1} w_{ij}(p_{ij} - m_i)^2}{\sum_{j=0}^{N-1} w_{ij}}}$$

e. operaImAvgSigClip

This function average-sigma-clip combines a set of image arrays `*arrays[]` into the master `*master`. The clip is done at n_σ x σ.

Return: `void`

Parameters: `unsigned depth = N, long npixels = M, float *arrays[] = {{p}0,{p}1,...,{p}N-1}, float *master = {m'}, unsigned nsig = nσ.`

Algorithm:

Each element `m'ᵢ` of the master image array is calculated as follows:

$$m'_i = \frac{\sum_{j=0}^{N-1} p_{ij} w_j}{\sum_{j=0}^{N-1} w_j}$$, where the following condition applies

if $|p_{ij} - m_i| \leq n_\sigma \sigma_i$ **then** {

$$w_j = 1$$

} **else then** {

$$w_j = 0$$

}

f. operaImVarDiff

This function calculates the variance image `*diffvarimg` from the difference between consecutive images in a set of image arrays `*arrays[]`.

Return: `void`

Parameters: `unsigned depth = N, long npixels = M, float *arrays[] = {{p}₀,{p}₁,...,{p}N-1},float *diffvarimg = {v}.`

Algorithm:

Each element `vᵢ` of the variance image array is calculated as follows:

$$v_i = \frac{\sum_{j=0}^{N-2} (p_{i,j+1} - p_{ij})^2}{N-1}.$$

g. operaCCDVarDiff

This function calculates the global pixel variance σ^2 from the difference between consecutive images in a set of image arrays `*arrays[]`, where pixel values are weighted by the weight array `{w}`.

Return: `float *vardiff` = σ^2

Parameters: `unsigned depth = N, long npixels = M, float *arrays[] = {{p}0,{p}1,...,{p}N-1}, float *weight = {w}.`

Algorithm:

The global variance is calculated as follows:

$$\sigma^2 = \frac{\sum_{i=0}^{M-1} w_i \frac{\sum_{j=0}^{N-2} (p_{i,j+1} - p_{ij})^2}{N-1}}{\sum_{i=0}^{M-1} w_i}.$$

h. medianCombineFloat

This function median combines a set of image float arrays `*arrays[]` into the master image array `*master`.

Return: `float *master = {m}`

Parameters: `unsigned depth = N, long npixels = M, float *master = {m}, float *arrays[] = {{p}0, {p}1,...,{p}N-1}`

Algorithm:

Each element m_i of the master image array is calculated as follows:

$$m_i = \text{operaArrayMedianQuick}(\text{depth}, \{p_i\}),$$

where $\{p_i\} = \{p_i^0, p_i^1, ..., p_i^{N-1}\}$, which is an array of elements having the same index position from all images in the set.

i. medianCombineUSHORT

This function median combines a set of image USHORT arrays `*arrays[]` into the master image array `*master`.

Return: `unsigned short *master = {m}`

Parameters: `unsigned depth = N, long npixels = M, unsigned short *master = {m}, unsigned short *arrays[] = {{p}0, {p}1,...,{p}N-1}`

Algorithm:

Each element m_i of the master image array is calculated as follows:

$$m_i = \text{operaArrayMedianQuickUSHORT(depth, } \{p_i\}),$$

where $\{p_i\} = \{p_i^0, p_i^1, ..., p_i^{N-1}\}$, which is an array of elements having the same index position from all images in the set.

6. 5 Least Squares Fitting Library

Name: libOperaFit

This library contains the subroutines to obtain best fit parameters for given functions or to perform interpolations.

Fitting function prototypes

```
void operaLMFitPolynomial(unsigned m_dat, double *x, double *y, int n_par, double
par[], double *chi2, int verbose);

void operaLMFitGaussian(unsigned m_dat, double *x, double *y, double *a, double *x0,
double *sig, double *chi2, int verbose);

void operaLMFit2DPolynomial(unsigned m_dat, double *x, double *y, double *fxy, int
n_par, double par[], double *chi2, int verbose);

void operaLMFit2DGaussian(unsigned m_dat, double *x, double *y, double *fxy, double
*a, double *x0, double *y0, double *sigx, double *sigy, double *chi2, int verbose);

void operaMdefit(float xin[], float yin[], int ndata, float *a, float *b, float *ab-
dev);

void operaFitSpline(unsigned nin, float *xin,float *yin, unsigned nout, float *xout,
float *yout);

void operaFit2DSpline(unsigned nxin, float *xin, unsigned nyin, float *yin,double
*fxyin, unsigned nxout, float *xout, unsigned
```

Functions

a. operaLMFitPolynomial

This function uses the Levenberg-Marquardt method to calculate the coefficients {a} of a n-degree polynomial that best fit {x, y} data.

Return: `void`

Parameters: `unsigned m_dat = N, double *x = {x}, double *y = {y}, int n_par = n, double par[] = {a}, double *chi2 = χ², int verbose.`

Algorithm:

This function uses the LM algorithm from the **operaLMFit** library to find the coefficients {a} that minimize the merit function χ^2, which is given by:

$$\chi^2 = \frac{\sum_{j=0}^{N-1} \left(y_j - \sum_{i=0}^{n-1} a_i x_j^i \right)^2}{N - n}$$

The function call for the minimization algorithm is the following:

```
lmcurve_fit( n_par, par, m_dat, x, y, PolynomialFunction, &con-
trol, &status);
```

where `PolynomialFunction` is the function that evaluates the polynomial, which is given below:

```
double PolynomialFunction(double x, const double *p, int n_par)
{
  double fpoly = 0;

  for(int i=0;i<n_par;i++)
     fpoly += p[i]*pow(x,(double)i);

  return fpoly;
}
```

b. operaLMFit2DPolynomial

This function uses the Levenberg-Marquardt method to calculate the coefficients {{a}} of a (n x m)-degree 2D polynomial that best fit `{x, y, z}` data.

Return: `void`

Parameters: `unsigned m_dat = N, double *x = {x}, double *y = {y}, double *fxy = {z}, int n_par = n x m, double par[] = {{a}}, double *chi2 = χ², int verbose.`

Algorithm:

This function uses the LM algorithm from the **operaLMFit** library to find the coefficients $\{a_{ij}\}$, with `i = 0...n-1 and j = 0...m-1`, that minimize the merit function χ^2, which is given by:

$$\chi^2 = \frac{\sum_{k=0}^{N-1} \left(z_k - \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} a_{ij} x_k^i y_k^j \right)^2}{N - n \times m}$$

Note that each coefficient `par[l]` in the array `par[]` is associated to the coefficients in the array $\{a_{ij}\}$ by the expression `l = i + j.n`, where `l = 0...(n_par = n.m - 1)`.

The function call for the minimization algorithm is the following:

```
lmmin( n_par, par, m_dat, (const void*) &data,
       evaluate_surface, &control, &status, lm_printout_std );
```

where the 2D-polynomial is evaluated by the `evaluate_surface` function, which uses the structure `poly2Ddata_struct`, which contains the `{x,y,z}` data and also the 2D-polynomial function given by:

```
double Polynomial2DFunction(double x, double y, const double *p, int n_par)
{
   double fxy=0;
   int k=0;
   int m,n;

   n = n_par/2;
   m = n_par - n;

   for(int j=0;j<m;j++){
      for(int i=0;i<n;i++){
            fxy += p[k++]*pow(x,(double)i)*pow(y,(double)j);
      }
   }
   return fxy;
}
```

c. operaLMFitGaussian

This function uses the Levenberg-Marquardt method to calculate the parameters A, x_0 and σ of a Gaussian function that best fit `{x,y}` data.

Return: `void`

Parameters: `unsigned m_dat = N, double *x = {x}, double *y = {y}, double *a = A, double *x0 = x0, double *sig = σ, double *chi2 = χ², int verbose.`

Algorithm:

This function uses the LM algorithm from the **operaLMFit** library to find the parameters A, x_0 and σ that minimize the merit function χ^2, which is given by:

$$\chi^2 = \frac{\sum_{i=0}^{N-1} \left(y_i - Ae^{-\frac{(x_i - x_0)^2}{2\sigma^2}} \right)^2}{N-3}.$$

The function call for the minimization algorithm is the following:

```
lmcurve_fit( n_par, par, m_dat, x, y, GaussianFunction, &con-
trol, &status);
```

where `GaussianFunction` is the function that evaluates the Gaussian, which is given below:

```
double GaussianFunction(double x, const double *p, int n_par)
{
  return p[0]*exp(-(x-p[1])*(x-p[1])/(2*p[2]*p[2]));
}
```

Note that the parameter array `p[]` associates each element to the Gaussian coefficients as follows:

`p[0]` = A, `p[1]` = x_0, and `p[2]` = σ.

d. operaLMFit2DGaussian

This function uses the Levenberg-Marquardt method to calculate the parameters A, x_0, y_0, σ_x, and σ_y of a 2D-Gaussian function that best fit `{x,y,z}` data.

Return: `void`

Parameters: `unsigned m_dat = N, double *x = {x}, double *y = {y}, double *fxy = {z}, double *a = A, double *x0 = x0, double *y0 = y0, double *sig = σx, double *sig = σy, double *chi2 = χ², int verbose.`

Algorithm:

This function uses the LM algorithm from the **operaLMFit** library to find the parameters A, x_0, y_0, σ_x, and σ_y, of a 2D-Gaussian function, that minimize the merit function χ^2, which is given by:

$$\chi^2 = \frac{\sum_{i=0}^{N-1}\left(z_i - Ae^{-\left(\frac{(x_i-x_0)^2}{2\sigma_x^2}+\frac{(y_i-y_0)^2}{2\sigma_y^2}\right)}\right)^2}{N-5}$$

The function call for the minimization algorithm is the following:

```
lmmin( n_par, par, m_dat, (const void*) &data,evaluate_2DGaussian,
&control, &status, lm_printout_std );
```

where the 2D-Gaussian is evaluated by the `evaluate_2DGaussian` function, which uses the structure `gauss2Ddata_struct`, which contains the `{x,y,z}` data and also the 2D-Gaussian function given by:

```
double Gaussian2DFunction(double x, double y, const double *p, int n_par)
{
   return
p[0]*exp(-((x-p[1])*(x-p[1])/(2*p[3]*p[3])+(y-p[2])*(y-p[2])/(2*p[4]*p[4])));
}
```

Note that the parameter array `p[]` associates each element to the 2D-Gaussian coefficients as follows:

`p[0] = A, p[1] = x0, and p[2] = y0, p[3] = σx, and p[4] = σy.`

e. operaMedfit

This function uses a robust method to find the parameters a and b from a linear function, `y = a + b.x`, that best fit the `{x,y}` data.

Return: `void`

Parameters: `float xin[] = {x}, float yin[] = {y}, int ndata = N, float *a = a, float *b = b, float *abdev = d.`

Algorithm:

This function finds the best fit parameters for a linear function by minimizing the absolute deviation `d`, which is the merit function, given by

$$d = \sum_{i=0}^{N-1} |y_i - a - bx_i| \quad .$$

The algorithm uses robust statistic and is described in more detail in Press, W. H. et al. (1992).

f. operaFitSpline

This function takes an input array of data `{x,y}` with N elements and performs interpolation using a sufficiently smooth piecewise-polynomial function (cubic spline). The interpolations are provided in the ouput array `{y'}`, which is defined by an input array of ordinates `{x'}` with M elements.

Return: `void`

Parameters: `unsigned nin = N, float *xin = {x}, float *yin = {y}, unsigned nout = M, float *xout = {x'}, float *yout = {y'}`

Algorithm: Press, W. H. et al. (1992)

f. operaFit2DSpline

This function takes an input array of data `{x,y,z}` with N_x elements in x-axis and N_y elements in y-axis, and performs interpolation using a sufficiently smooth piecewise-polynomial function (cubic spline). The interpolations are provided in the ouput array `{z'}`, which is defined by an input array `{x'}` with M_x elements in x-axis and `{y'}` with M_y elements in y-axis.

Return: `void`

Parameters: `unsigned nxin = Nx, float *xin = {x}, unsigned nyin = Ny, float *yin = {y}, float *fxyin = {z}, unsigned nxout = Mx, float *xout = {x'}, unsigned nyout = My, float *yout = {y'}, float *fxyout = {z'}`

Algorithm: Press, W. H. et al. (1992)

6. 6 Image CCD Library

Name: libOperaImCCD

This library contains the functions to obtain CCD quantities from FITS images.

Function prototypes

```
void operaMaskPixbyCountRange(unsigned npixels, float *array, float *badpix-
mask, float *newmask, float minvalue, float maxvalue, unsigned char invert);

void operaMarkPixbyCountRange(unsigned npixels, float *array, float *previ-
ousmarks, float *newmarks, float minvalue, float maxvalue, float index2mark);

void operaCCDGainNoise(unsigned npixels, unsigned nbias, float *biasdata[],
unsigned nflat, float *flatdata[], float *badpixdata, float lowcount, float
highcount, float maxbins, float minnpixelsperbin, float *gain, float *noise);
```

Functions

a. operaMaskPixbyCountRange

This function creates a new mask based on a previous bad-pixel mask and on a defined range of values.

Return: `void`

Parameters: `unsigned npixels = N, float *array = {p}, float *badpixmask {b}, float *newmask {m}, float minvalue = MIN, float maxvalue = MAX, unsigned char invert`

Algorithm:

for (N pixels) **do**
  if (p_i > MIN **and** p_i < MAX) **then**
    $m_i = b_i$ (don't mask pixel)
  else then
    $m_i = 0$ (mask pixel)

b. operaMarkPixbyCountRange

This function creates a mask where it marks pixels with a value within a defined range of values. It keeps the previous mark mask values.

Return: `void`

Parameters: `unsigned npixels = N, float *array = {p}, float *badpixmask {b}, float *newmask {m}, float minvalue = MIN, float maxvalue = MAX, unsigned index2mark = k`

Algorithm:

for (N pixels) **do**
  if (p_i > MIN **and** p_i < MAX) **then**
    m_i = k (mark pixel with index value)
  else then
    m_i = b_i (keep previous pixel value)

c. operaCCDGainNoise

This function uses a set of bias (minimum of 1) and a set of flat (minimum of 2) image arrays to calculate the CCD gain and noise of the detector.

Return: `void`

Parameters: `unsigned npixels = N, unsigned nbias = n_b, float *biasdata[] = {{b}}, unsigned nflat = n_f, float *flatdata[] = {{f}}, float *badpixdata = {m}, float lowcount = L, float highcount = H, float maxbins = M, float minnpixelsperbin = N_min, float *gain = G, float *noise = R`

Algorithm:

1. Calculate bin size B:

$$B = \frac{H - L}{M}$$

2. Create a master flat {F} from averaging all input flats:

$$F_i = \frac{\sum_{j=0}^{n_f - 1} f_{ij}}{n_f}$$

3. Loop over all bins to index each pixel to a correspondent bin, or leave index as zero if its value doesn't fall in any bin. The indexation is made by means of the function `operaMarkPixbyCountRange`, which assign values to pixels lying within a given range.

set j = 0
set {jmask} = 0;
for k=0 to k=(M-1) **do** {
  Count number of pixels (Np) that satisfy: B*k < {F} < B*(k+1)
  if (Np > N_{min}) **then** {
    j = j + 1
    Update {jmask} by assigning value j for pixels that satisfy:
    B*k < {F} <B*(k+1)
  }
}
set N_b = j

4. Calculate quantities from bias images:

set q = 0
set {<b1>} = 0
set {<b2>} = 0
set {<σb>} = 0

if (n_b > 1) **then** {
  for i = 0 to i = (n_b -2) **do** {
    {d} = $\{b\}_{i+1}$ - $\{b\}_i$

```
        {b1} = IndexedMean {b}_i
        {b2} = IndexedMean {b}_i+1
        {σb} = IndexedSig {d}

        for k=0 to k=(N_b-1) do {
            <b1>_k = <b1>_k + b1_k/(n_b -1)
            <b2>_k = <b2>_k + b1_k/(n_b -1)
            <σb>_k = <σb>_k + σd_k/(n_b -1)
            NoiseArr_q = σd_k
            q = q +1
        }
    }
} else if ( n_b = 1) then {

        {b1} = IndexedMean {b}_i
        {σb} = IndexedSig {b}_i
        for k=0 to k= N_b do {
            NoiseArr_q = σd_k
            q = q +1
        }
}
```

5. Calculate quantities from flat images:

```
set {<f1>} = 0
set {<f2>} = 0
set {<σf>} = 0

for i = 0 to i = (n_f -2) do {
    {d} = {f}_i+1 - {f}_i

    {f1} = IndexedMean {f}_i
    {f2} = IndexedMean {f}_i+1
    {σf} = IndexedSig {d}

    for k=0 to k=(N_b-1) do {
        <f1>_k = <f1>_k + f1_k/(n_f -1)
        <f2>_k = <f2>_k + f1_k/(n_f -1)
```

$<\sigma f>_k = <\sigma f>_k + \sigma f_k/(n_f - 1)$

}

}

6. Calculate variance {x} and signal-to-noise {y} for each bin:

for k=0 to k=(N_b-1) **do** {

$$x_k = \frac{\langle \sigma_f \rangle_k^2}{\langle \sigma_b \rangle_k \sqrt{2}}$$

if ($n_b > 1$) **then** {

$$y_k = -\frac{\langle f_1 \rangle_k + \langle f_2 \rangle_k - \langle b_1 \rangle_k - \langle b_2 \rangle_k}{\langle \sigma_b \rangle_k \sqrt{2}}$$

} **else if** ($n_b = 1$) **then** {

$$y_k = -\frac{\langle f_1 \rangle_k + \langle f_2 \rangle_k - 2\langle b_1 \rangle_k}{\langle \sigma_b \rangle_k \sqrt{2}}$$

}

}

7. Calculate noise R_{ADU} in ADU units:

if ($n_b > 1$) **then** {

R_{ADU} = **Median**({NoiseArr})/sqrt(2)

} **else if** ($n_b = 1$) **then** {

R_{ADU} = **Median**({NoiseArr})

}

8. Calculate gain G in e-/ADU by obtaining the angular coefficient of a robust linear fit to {x,y} data. The fit is done using the function `operaMedfit`.

Fit {x,y} data to function f(x) = R + G.x

9. Calculate noise in e- units:

$N = G * R_{ADU}$;

6. 7 Fast Fourier Transform Library

Name: libOperaFFT

This library contains the functions to perform backward and forward Fourier transform of data.

Functions:

a. operaFFTforward: return the forward Fast Fourier Transform of an array of data.

```
OperaErr operaFFTforward(const OperaArray *data, OperaArray *out-
data);
```

b. operaFFTbackward: return the backward Fast Fourier Transform of an array of data.

```
OperaErr operaFFTbackward(const OperaArray *data, OperaArray
*outdata);
```

c. operaFFTzeropad: zero-pad an array of data.

```
OperaErr operaFFTzeropad(const OperaArray *data, OperaArray *out-
data, int M);
```

d. operaFFTConv: return the convolution between two arrays of data using the FFT.

```
OperaErr operaFFTConv(const OperaArray *data1, const OperaArray
*data2, OperaArray *outdata);
```

e. operaFFTAutoCorr: return the auto-correlation of an array of data using the FFT.

```
OperaErr operaFFTAutoCorr(const OperaArray *data, OperaArray
*outdata);
```

f. operaFFTCrossCorr: return the cross-correlation between two arrays of data using the FFT.

```
OperaErr operaFFTCrossCorr(const OperaArray *data1, const Op-
eraArray *data2, OperaArray *outdata);
```

g. operaFFTfiltering: perform filtering of an array of data using the FFT.

```
OperaErr operaFFTfiltering(const OperaArray *data, OperaArray
*outdata, double threshold);
```

6. 8 Astronomy Library

Name: libOperaAstro

This library contains the functions to calculate astronomical quantities.

FUNCTIONS

a. operaAstroTime: function to calculate astronomical times. The inputs are the time in one of the supported formats, sky position, and the Earth location. By calling this function it will calculate and populate the other time formats inside the structure. The supported time formats are the following: UT (Universal Time), ST (Sidereal Time), LT (Local Time), JD2000 (Julian Date for equinox 2000), MJD (Modified Julian Date), HJD (Heliocentric Julian Date), GJD (Geocentric Julian Date).

```
OperaErr operaAstroTime(OperaTime *obstime, OperaSkyPosition *ob-
jcoord, OperaEarthLoc *maunakealoc);
```

where `obstime` is a pointer to an `OperaTime` structure, where the time information is stored, `objcoord` is a pointer to an `OperaSkyPosition` structure, where the sky coordinates are stored, and `maunakealoc` is a pointer to an `OperaEarthLoc` structure, where the observatory global positioning coordinates are stored.

b. operaAstroRV: function to calculate the radial velocity (in m/s) for a given location on Earth with respect to a certain sky direction.

```
OperaErr operaAstroRV(OperaTime *obstime, OperaSkyPosition *ob-
jcoord, OperaEarthLoc *maunakealoc, OperaRV *obsrv);
```

where `obstime` is a pointer to an `OperaTime` structure, `objcoord` is a pointer to an `OperaSkyPosition` structure, `maunakealoc` is a pointer to an `OperaEarthLoc` structure, and `obsrv` is a pointer to an `OperaRV` structure, where the RV information is stored (geocentric, heliocentric, etc).

c. operaAstroVegaVmag2Flux: function to convert from V-magnitude in the Vega system to physical flux units for point-like sources.

```
OperaErr operaAstroVegaVmag2Flux(double *Vmag, OperaFlux *ob-
jflux);
```

where `Vmag` is the magnitude in the V-band and `objflux` is a pointer to an `OperaFlux` structure, where flux is stored for many different units.

d. operaAstroFlux2VegaVmag: function to convert from physical flux units to V-magnitude in the Vega system for point-like sources.

```
OperaErr operaAstroVegaVmag2Flux(double *Vmag, OperaFlux *flux);
```

where `Vmag` is the magnitude in the V-band and `objflux` is a pointer to an `OperaFlux` structure, where flux is stored for many different units.

e. operaAstroStokesQ: calculate the linear polarization parameter, Stokes Q, given the intensities for two orthogonal polarization components aligned with North and South sky directions.

```
double operaAstroStokesQ(double *Inorth, double *Isouth);
```

f. operaAstroStokesU: calculate the linear polarization parameter, Stokes U, given the intensities for two orthogonal polarization components 45 degrees with respect to North and South sky directions.

```
double operaAstroStokesU(double *I45north, double *I45south);
```

g. operaAstroStokesV: calculate the circular polarization parameter, Stokes V, given the intensities for two circular polarization components.

```
double operaAstroStokesV(double *Iplus, double *Iminus);
```

h. operaAstroLSPeriodogram: generates the Lomb-Scargle periodogram for a given array of data.

```
OperaErr operaAstroLSPeriodogram (OperaArray *data, OperaArray
*periodogram);
```

6. 8 External Libraries

Below are the external libraries that will be used by OPERA.

- **CFITSIO**

Description: CFITSIO is a library of ANSI C routines for reading and writing FITS format data files.

Copyright: William D. Pence, HEASARC, NASA/GSFC email: William.D.Pence@nasa.gov

License: Copyright (Unpublished--all rights reserved under the copyright laws of the United States), U.S. Government as represented by the Administrator of the National Aeronautics and Space Administration. No copyright is claimed in the United States under Title 17, U.S. Code.

Permission to freely use, copy, modify, and distribute this software and its documentation without fee is hereby granted, provided that this copyright notice and disclaimer of warranty appears in all copies. (However, see the restriction on the use of the gzip compression code, below).

web location: http://heasarc.gsfc.nasa.gov/fitsio/

- **FFTW**

Description: FFTW is a free collection of fast C routines for computing the Discrete Fourier Transform in one or more dimensions. It includes complex, real, symmetric, and parallel transforms, and can handle arbitrary array sizes efficiently. FFTW is typically faster than other publically-available FFT implementations, and is even competitive with vendor- tuned libraries. (See our web page for extensive benchmarks.) To achieve this performance,

FFTW uses novel code-generation and runtime self-optimization techniques (along with many other tricks).

Copyright: Matteo Frigo and Steven G. Johnson

License: GNU General Public License version 2

web location: http://www.fftw.org

- **LMFIT**

Description: C/C++ library for Levenberg-Marquardt least-squares minimization and curve fitting.

Copyright: (C) 2009-10 Joachim Wuttke.

License: Public Domain.

web location: http://joachimwuttke.de/lmfit/index.html

- **Numerical Recipes**

Description: Numerical Recipes Software.

Copyright: Unless otherwise designated, all files are copyright & copy; 1986-2007 by Numerical Recipes Software.

License: Numerical Recipes Personal Single-User License. web location: http://www.nr.com/

Note that OPERA will used zero-based indexing,unlike Numerical receipes which uses one based indexing so there is no license or copyright conflict.

- **SOFA-Issue: 2010-12-01**

Description: This is the IAU Standards of Fundamental Astronomy (SOFA) Libraries product, issued on 2010-12-01.

License: Using SOFA software is free of charge under the terms and conditions of the SOFA licence. web location: http://www.iausofa.org .

7. Major Classes and Data Structures

7.1 Introduction

OPERA is written in the modern object-oriented high level language C++ with abstraction facilities such as operator overloading and hierarchical classes that permit users wishing to extend opera to do so at a high level of abstraction, concentrating on the science and not the details of address arithmetic and memory management. While this statement seems like motherhood and few would differ, in practice it is little followed. In the abstract is seems obvious, but an illustration from a typical pipeline helps to bring the point home. This example is from WIRCam, but the principle is the same. Here is a fragment typical of a pipeline:

```
size=raw->getImageHeader().naxis1*raw->getImageHeader().naxis2;
array=(float *)malloc(size*sizeof(float));
image=(float *)malloc(size*sizeof(float));
/* Go through each extension */
for (ext=1;ext<=raw->getImageHeader().nextend;ext++){
 /* Go through each slice */
 for (s=1;s<=raw->getImageHeader().naxis3;s++){
       /* Go through each pixel */
       for (i=0;i<size;i++){
            value=raw->getImageArrayValue(((ext-1)*
            raw->getImageHeader().naxis3+(s-1))*size+i);
            value=value-dark->getImageArrayValue((ext-1)*size+i);
            value=value/flat->getImageArrayValue((ext-1)*size+i);
            raw->setImageArrayValue(((ext-1)*
            raw->getImageHeader().naxis3+(s-1))*size+i,val);
       }
 }
}
free(image);
free(array);
```

This example implements the algorithm:

> *To detrend an image subtract the dark image and then divide by the flat.*

The exact same algorithm is expressed with the facilities of OPERA as this:

```
output = (image - dark) / flat;
```

Note the difference. The typical pipeline software is filled with memory management (malloc/free) and address calculations. The first example is several hundred times slower than the OPERA example since it is performing many useless address calculations using the * (multiply) operator. and function calls. Secondly, in the first example the algorithm is lost in the details of performing address calculations that have little to do with the science algorithm. The OPERA version benefits from the observation that this algorithm is dimensionless and thus no address calculations need be performed at all. The OPERA example reads quite like the English expression of the algorithm, it is much more readable. The OPERA version takes care of the details and lets the scientist coding the algorithms concentrate on science. The scientist would spend much less time writing and debugging the algorithm given the facilities of OPERA.

OPERA classes define images as first class objects, which can be operated upon with arithmetic operators just like scalars. There are many other support classes implementing objects such as fluxvectors, etc.

In general the Classes follow typical class definitions, they consist of constructors, destructors, useful methods according to the class and getters/setters.

The classes are largely self-documenting and doxygen tags are given.

7.2 operaSpectralOrderVector

Description

This class manipulates the information related to the entire set of spectral orders in the echellogram. This class is mainly used to read and write the calibration files (*.geom, *.prof, *.wave, etc.), each of which contains the calibration information that will be used to feed the calibration classes lying within each order, and then will allow the calibration information to be accessed at any time. `operaSpectralOrderVector` contains a vector of references to `operaSpectralOrder` class instances. This class supports serialization and deserialization of the following types of information: geometry, wavelength, spectral elements and instrument profiles.

Major Methods

Major methods of note are:

- `operaSpectralOrderVector(string Filename)` - creates a spectral order vector from the information in “filename”. The file may contain geometry, wavelength, spectrum, instrument profile or order spacing data. All these file formats are self-identifying.
- `ReadSpectralOrders(string Filename)` - augments an existing spectral order vector with new information. Typically one would create a spectral order vector from a geometry calibration and augment that with wavelength information.
- `WriteSpectralOrders(string Filename, operaSpectralOrder_t Format)` - serializes a specific type of spectral order information
-

7.3 operaSpectralOrder

Description

This class manipulates the information concerned with a spectral order. The spectral order is defined by the following members:

(a) `operageometry`

(b) `operaInstrumentProfile`

(c) `operaWavelength`

(d) `operaSpectralElements`.

Each of these members is a class of itself. The first three members are calibration classes. They are used either to read calibration data and produces a calibration model or to read the model and provide the calibration information requested by other processing steps. Notice that these three classes are very similar in nature. They all have a data structure to store measurements and errors. They also have methods to produce a model from the measurements. Then they have structures to store the model information and methods to retrieve data back from the model. Since all of these calibrations assume smooth variations along the dispersion, they all use polynomial functions to obtain the calibration models.

The last member of this class, `operaSpectralElements`, is a structure to store the calibrated information. This is essentially a set of data points, which includes flux, wavelength, photo-center positions, etc.

`operaSpectralOrder` contains references to instances of the four types of information retained to represent a single spectral order. The four types are the geometric model, the wavelength model, the spectral element and the instrument profile of the order as described below.

Methods

Major methods of note are:

- `NormalizeFlat` - create a normalized pixel sensitivity map for the illuminated regions of the detector, where it is possible to obtain this information.
- `extractRawSum` - used by wavelength calibration as a first order approximation to the spectral intensities of the order.
- `measureInstrumentProfileAlongRows` - creates the instrument profile model.
- `CalculateWavelengthSolution` - creates a distance and wavelength vector pair from geometry and the wavelength polynomial.
- `extractOptimalSpectrum` - obtain measurements for the flux, wavelength and uncertainties using four different methods of extraction.

7.4 operaGeometry

Description

This class is used to manipulate the geometry calibrations. The geometry calibration provides the location on the FITS image for the photo-center along the dispersion direction, which is basically the accurate position of the order. The class `operageometry` stores two views of geometry. One is the polynomial describing the position of the order and the second is the set of data points describing the order centers and vectors of pixel values.

Methods

Major methods of note are:

`traceOrder` - performs polynomial fits to the data vector with varying polynomials. The polynomial fit with chisqr closest to 1 is chosen as the best fit.

There are no other major methods, the other function of this class is simply as a container of x, y, and flux values of this order.

7.5 operaWavelength

Description

The class `operaWavelength` is the object that defines the pixel-to-wavelength solution. This object provides the tools to obtain the calibration to convert from pixel space to a relevant physical unit. nm is the choice for the opera primary physical unit of wavelength since the reference atlas of spectral lines used by opera is in the visible range with units of wavelength in nm. The class `operaWavelength` stores two views of wavelength calibration information for a given order. The first view is a polynomial and the second is the elaboration of the polynomial in the dimensions of distance along the order and wavelength. `operaWavelength` contains no major methods and is simply a container.

7.6 operaInstrumentProfile

Description

This class manipulates the measurements and modeling for the Instrument Illumination Profile (IP). The data and model for this class are given by a pixelized image. The fiducial set of coordinates for the IP is set with the ordinate oriented along the FITS image columns, the abscissa along the

rows, and the center (zero) is defined at a given photo-center position provided by the geometry class.

7.7 operaSpectralElements

Description

`operaSpectralElements` encapsulates the calibrated physical information extracted from an object image such as flux and the corresponding calibration such as position and wavelength. It contains references to vectors of data which may each be considered a “spectral element”, a single unit along the spectral order. There may be various views on these vectors, for instance x,y coordinates vs flux and flux uncertainty, or wavelength vs flux and flux uncertainty, the choice of view being dependent on the needs of the calling module.

7.8 Polynomial

Description

This class stores the polynomial order and coefficients and provides an evaluation function.

7.9 operaExtractionAperture

Description

This class is a container for the information obtained from the extraction aperture calibration. The class basically contains an extraction shape (see `apertureShape`), a rectangle that defines the bounding box, `x` and `y` sampling factors, and a set of pixels which is defined by the class `PixelSet`.

Major Methods

The major methods in this class are meant to interface the interaction between the aperture for extraction and the FITS image or the instrument profile image. These major methods are briefly explained below:

- Set Subpixels:

```
void setSubpixels(void);
void setSubpixels(operaFITSImage &Image);
void setSubpixels(operaInstrumentProfile *instrumentProfile,
float d);
void setSubpixels(operaInstrumentProfile *instrumentProfile);
```

There are three versions of this method. All of them have basically the same functionality, which is to assign the values relative to the set of sub pixels. In the first version it only assigns sub-pixel coordinates and indexes values, and the pixel values are set to zero. The second version uses the images coordinates, indexes and pixel values to populate PixelSet. The last two versions do the same but for an instrument profile, where the first doesn't copy the values.

- Methods to modify aperture coordinate system:

```
void shiftAperture(float xshift, float yshift, operaFITSImage
&Image);
void recenterAperture(operaPoint &NewCenter, operaFITSImage
&Image);
void recenterAperture(operaPoint &NewCenter);
```

The three methods above are mainly used to change the center of an aperture and propagate this change to all members in the aperture. The first two versions recenter the aperture on a FITS image and copy the new matching pixel values and coordinates to the sub-pixel set. The first applies an x and y shift and the second takes the new center point as input. The last version only moves the aperture coordinates and doesn't copy any pixel value.

7.10 PixelSet

Description

The PixelSet class stores an indexed set of pixels or sub-pixels. The pixels are stored with origin coordinate x and y centers as well as the pixel value. Indices are maintained of the x and y coordinates. The sub-pixel area, in the case where the PixelSet is over sampled, is retained. The class is designed to be re-usable, storage allocation is only done if the initial constructor did not allocate adequate storage.

Major Methods

The constructors are:

- `PixelSet::PixelSet(void) - the base constructor.`
- `PixelSet::PixelSet(float SubPixelArea) - sets the sub-pixel area.`
- `PixelSet::PixelSet(unsigned NPixels, float SubPixelArea) - sets the sub-pixel area and allocated storage for the pixel set and indices.`

In addition to getters and setters, the major methods are:

- `void PixelSet::setSubPixels(PixelSet &pixels) - copies a pixel-set.`
- `void PixelSet::resize(unsigned newsize) - resizes a pixel set, reallocating storage if needed.`

7.11 operaGeometricShapes

Description

This class encapsulates geometric shapes which are useful in instrument profile calibration and aperture calibration. The major shapes are:

- operaPoint - x, y coordinates

- Box - x, y, dx, dy xoordinates
- Rectangle - dx, dy, angle, operaPoint center
- Circle - radius, operaPoint center
- Polygon - number of sides, operaPoint [] vertices
- Line - intercept, dx, dy, operaPoint midpoint

Major Methods

In addition to the constructors, which simply store the parameters, the major methods of note are:

- `operaPoint::operaPoint(Line &inputLine1, Line &inputLine2)` - creates a point of the intersection of 2 lines.
- `Rectangle::Rectangle(float Width, float Height, float Angle)` - stores parameters and calculates the four corner points.
- `Rectangle::Rectangle(float Width, float Height, float Angle, operaPoint &Center)` - stores parameters and calculates the four corner points.
- `void Rectangle::setRectangle(Rectangle &aRectangle)` - copies parameter.
- `void Rectangle::setRectangle(float Width, float Height, float Angle, operaPoint &Center)` - sets parameters into current rectangle.
- `Line::Line(Line &inputLine, LinePosition_t LinePosition)` - creates a top, left, right, or bottom Line based on the given line.
- `void Polygon::simplePolygonization(void)` - calculates the anchor-point from the vertices and sorts the vertices.
- `bool Polygon::pointInPolygon(operaPoint &testPoint)` - tests whether the testPoint is inside the polygon.
- `bool Circle::pointInCircle(operaPoint &TestPoint)` - tests whether the testPoint is inside the circle.
- `bool Rectangle::pointInRectangle(operaPoint &TestPoint)` - tests whether the testPoint is inside the rectangle.

- bool Line::pointOnLine(operaPoint &TestPoint) - tests whether the testPoint is inside the rectangle.

7.12 operaWavelength

Description

The class operaWavelength is the object that is used to handle the data and modeling that define the pixel-to-wavelength solution. This object provides the tools to obtain the calibration data and to convert from pixel space to a relevant physical unit. Nanometer (nm) is the choice for the OPERA primary physical unit of wavelength since the reference atlas of spectral lines used by OPERA is in the visible range with units of wavelength in nm. The class operaWavelength stores the wavelength calibration information as a polynomial which can be accessed through some of the major methods. The major methods in operaWavelength are described below.

Major Methods

Major methods of note are:

1. void calculateSpectralResolution(doubleValue_t ResolutionElementInPixels);

This method calculates the spectral resolution using the existing wavelength solution and an input resolution element, which is expected to be in instrumental units (pixels).

2. double evaluateWavelength(double distanceValue);

This method evaluates the wavelength for a given distance value (in pixels).

3. double convertPixelToWavelength(double DeltaDistanceInPixels);

This method converts a distance range from pixels to wavelength units.

4. void calculateRadialVelocityPrecision(void);

This method calculates the radial velocity precision from the residuals and applying the existing model for the wavelength calibration.

```
5.  double calculateWavelengthRMSPrecision(void);
```

This method calculates the RMS wavelength precision from the residuals.

```
6. double calculateWavelengthMedianPrecision(void);
```

This method calculates the median wavelength precision from the residuals.

```
7. void refineWavelengthSolutionByXCorrelation(unsigned
nPointsPerParameter, double parameterRangetoSearch, unsigned
maxpolyorder);
```

This method improves the wavelength solution by varying the values of the polynomial coefficients and searching for the solution that gives the highest correlation between the comparison and the atlas set of the spectral lines.

```
8. unsigned createAtlasSimulatedSpectrum(double *outputwl,
double *outputSpectrum, unsigned nstepspersigma);
```

This method produces a simulated atlas spectrum from a vector of spectral lines. The line widths are assumed to be constant and is obtained from the spectral resolution and the wavelength solution that are saved as members of the class.

```
9. unsigned createComparisonSimulatedSpectrum(double *out-
putwl, double *outputSpectrum, unsigned nstepspersigma);
```

This method produces a simulated comparison spectrum from a vector of spectral lines. The line widths are assumed to be constant and are obtained from the spectral resolution and the wavelength solution that is saved as members of the class.

```
10. void calculateXCorrelation(void);
```

This method use the two methods above to create simulated spectra using the comparison and atlas spectral lines, and then it calculates the cross-correlation between two simulated spectra.

```
   11. void matchAtlaswithComparisonLines(double acceptableMis-
match);
```

This method takes the two data sets containing atlas lines and comparisons lines where it applies the wavelength calibration to match the lines in both spectra. A given comparison line is matched with the atlas line if the difference between the centers lies within a given factor multiplied by sigma. Sigma is the resolution element and the factor is a given input. Only one comparison line can be identified by an atlas line, which is chosen to be the one with smallest center-to-center distance.

7.13 operaSpectralLines

Description

This class is a container for a set of spectral features, where each operaSpectralFeature is an object itself for storing a set of spectral lines (see operaSpectralFeature below). This class also has tools to store and extract information from a comparison spectrum. The spectrum is stored as an operaSpectralElements object, and the methods in the class perform operations in this spectrum to obtain information concerned to spectral lines.

Major Methods

Major methods of note are:

```
- void detectSpectralFeatures(double DetectionThreshold, double
LocalMaxFilterWidth, double MinPeakDepth);
```

This method implements an algorithm to detect spectral lines and organize them into spectral features. This also performs a multiple gaussian fit with background for each spectral feature, where the number of gaussians is

chosen to be equal to the number of lines detected in the feature. The algorithm for detecting lines is equivalent to the peak-detection algorithm used in operaGeometry and operaInstrumentProfile.

```
- void printLines(ostream *pout);
- void printReferenceSpectrum(ostream *pout);
```

These methods are tools to print spectral lines and reference spectrum with all respective information.

```
- unsigned selectLines(double MaxContamination, unsigned nSig,
double amplitudeCutOff, double *LinePositionVector,double *Li-
neSigmaVector,double *LineAmplitudeVector);
```

This method implements a filter to get rid of bad lines using the following criteria: 1. line presenting high levels of contamination by neighboring lines; 2. a sigma-cut based on the full width at half maximum; and 3. an amplitude cut-off.

7.14 operaSpectralFeature

Description

This class is a container for a set of blended lines. A feature consists of a data set and a model. The model is a superposition of multiple gaussians plus a background slope. The class performs a least squares fit of multiple gaussians by making use of the `Gaussian` class, which is described below.

Major Methods

Major methods of note are:

```
- void fitGaussianModel(void);
```

This method simply passes the data and arguments to the `Gaussian` class and then it calls the method that performs least-square fitting of the data.

```
- void fitBackground(void);
```

This method performs an independent fit to the background. The background obtained by this function may be used as the initial guess for the final background, which is obtained together with the multiple gaussian function in the fit performed by the `Gaussian` class.

7.15 Gaussian

Description

This class contains a set of vectors for the three parameters of a gaussian function and respective errors. Those vectors allow one to store a set of gaussian fits of indefinite size. It also has two additional parameters to account for a baseline slope as the background. In addition to these model parameters it also contains the data vectors. The major methods for this class involve the fitting routines which apply the Levenberg-Marquardt least-squares method. The major methods of note are presented below.

Major Methods

Major methods of note are:

```
- double EvaluateGaussian(double x);
```

This method evaluates and returns the value of the gaussian model at a given point x.

```
- void MPFitModeltoData(unsigned nPeaks);
```

This method performs a least-squares MP fit of the model to the data. The model is considered to be a sum of `nPeaks` multiple gaussians without background.

```
- void MPFitModeltoData(unsigned nPeaks, double BaselineIntercept, double BaselineSlope);
```

This method performs a least-squares MP fit of the model to the data. The model is considered to be a sum of `nPeaks` multiple gaussians plus a background slope.

7.16 operaSpectralEnergyDistribution

Description

This class is a container for the data and the model for the flux calibration. This also contains the main tools to handle the data and to obtain and apply the flux calibration to other dataset.

Major Methods

Major methods of note are:

- `TBD`

9.17 operaPolarimetry

Description

This class encapsulates the polarimetry results. It holds, in `operaStokesVector` classes, the 4 Stokes parameters, the 4 associated degrees of polarization and the 2 null polarization spectra for each Stokes parameter. This class contains the tools to perform measurements of the degree of polarization given the Stokes parameters or to calculate the Stokes parameters given the degree of polarization.

Major Methods

Major methods of note are:

- `calculatePolarization(void)` : A function that calculates the polarization from the degree of polarization and stores it in the Stokes vector.

- `calculateDegreeOfPolarization(void)` : A function that calculates the degree of polarization from the polarization and stores it in the degree of polarization vector.

7.18 operaStokesVector

Description

This class encapsulates the Stokes vector. It contains an array of 4 `operaFluxVector` to store each Stokes parameter or any results with the same format, such as the degree of polarization or the 2 null polarization spectra. `operaStokesVector` contains no major methods and is simply a container.

7.19 operaMuellerMatrix

Description

This class encapsulates the Mueller matrix. It holds all the parameters and their variances to create a Mueller matrix. It also holds the Mueller matrix itself and its variance matrix.

The variances are propagated as follows :

F = F(a,b)

DF = Pow(dF/da,2) * Da + Pow(dF/db,2) *Db

where DF is the resulting variance, Da and Db are the variance of the variables a and b, dF/da and dF/db are the partial derivatives of F. The variables are supposed uncorrelated.

Major Methods

Major methods of note are:

- `createRotatedPolarizer(double P, double PVariance, double Alpha, double AlphaVariance, double Theta, double ThetaVariance)` : A function that sets the 16 elements of the Mueller matrix and its variance matrix so that it represents a rotated polarizer.
- `createRotatedRetarder(double Phi, double PhiVariance, double Theta, double ThetaVariance)` : A function that sets the 16 elements of the Mueller matrix and its variance matrix so that it represents a rotated retarder.
- `createRotator(double Theta, double ThetaVariance)` : A function that sets the 16 elements of the Mueller matrix and its variance matrix so that it represents a rotator.
- `createAttenuator(double P, double PVariance)` : A function that sets the 16 elements of the Mueller matrix and its variance matrix so that it represents an attenuator.

7.20 operaFluxVector

Description

The fluxvector class encapsulates the notion of a vector of flux values with a corresponding vector of variance values. The variance can be used to calculate the error, There are a large number of operators such as subscripting, adding, multiplying, etc which are defined to operate point-wise along the entire vector.

The fluxvector is a key class for polarimetry. Proper calculation of polarimetry requires:

- that fluxes be calculated along the entire vector
- that variances be calculated along the entire vector according to the mathematical rules of variance propagation of the given operator.

- that there be control of the result of division by infinities.

This example from the polarimetry test case shows intended usage of the fluxvector:

```
/*
 * Populate the rn vectors with the E data
 */
operaFluxVector r1(length);
operaFluxVector r2(length);
operaFluxVector r3(length);
operaFluxVector r4(length);

/*
 * STEP 1 - calculate ratio of beams for each exposure
 * r1 = i1E / i1A
 * r2 = i2E / i2A
 * r3 = i3E / i3A
 * r4 = i4E / i4A
 */
r1 = i1E / i1A;
r2 = i2E / i2A;
r3 = i3E / i3A;
r4 = i4E / i4A;
```

The fluxvectors of the different beams are created and can be manipulated as entire vectors. The variances of the beams are propagated through each operation, and between lines of the algorithm. In this way variances, and thus errors, propagate from the beginning of the data through all transformations done on that data in the algorithm,

```
/*
 * STEP 2 - calculate the quantity R (Eq #2 on page 663 of paper)
 *     R^4 = (r1 * r4) / (r2 * r3)
 *     R = the 4th root of R^4
 */
operaFluxVector R(length, ToZero);
R = Sqrt(Sqrt((r1 * r4) / (r2 * r3)));
```

In these steps we introduce the "*Towards*" functionality, Here, algebraically, dividing r1*r3 by r2*r3 should give a zero value if both the numerator point and the denominator point along the vector have the value infinity. By specifying "*Towards*" as "*ToZero*", the operator definition accomplishes the desired result.

This example shows the use of "*ToOne*":

```
/*
 * STEP 3 - calculate the Stokes parameter (Eq #1 on page 663 of paper)
 *     aka "the polarization" P
 *     P/I = (R-1) / (R+1)
 */
operaFluxVector PoverI(length, ToOne);
PoverI = (R-1.0) / (R+1.0);
```

Here, algebraically, dividing R-1.0 by R+1.0 should give a one value if both the numerator point and the denominator point along the vector have the value infinity. By specifying "*Towards*" as "*ToOne*", the operator definition accomplishes the desired result.

Constructors

The major constructors are:

- `operaFluxVector::operaFluxVector(unsigned Length, TendsTowards_t Towards, bool Istemp)` - creates flux and variance vectors of the given length. The Towards parameter is optional, and controls division of infinities, defaulting to Default hardware results. Istemp is used internally by the operators.
- `operaFluxVector::operaFluxVector(double *Fluxes, double *Variances, unsigned Length, TendsTowards_t Towards, bool Istemp)` - creates flux and variance vectors of the given length and copies the given flux and variance vectors into the constructed vectors. The Towards parameter is optional, and controls division of infinities, defaulting to Default hardware results. Istemp is used internally by the operators.
- `operaFluxVector::operaFluxVector(operaFluxVector &b, TendsTowards_t Towards, bool Istemp)` - clones a fluxvector.

Major Methods

Major methods of note are:

- `void operaFluxVector::setVector(operaFluxVector &Fluxvector)` - copies the parameter fluxvector.

- `void operaFluxVector::setVectors(double *Fluxes, double *Variances)` - copies the parameter fluxes and variances.

Major Operators

The key to the fluxvector class is the power of the defined operators. All of the normal C++ operators are given an overloaded definition for the context of flux vectors. Note that the definitions are given here for completeness. The user of the class need never know this level of detail, but rather can simply use the operators to achieve a level of abstraction, The operators work in a similar way, so only a few of note are described:

```
/*!
 * \brief Assignment operator.
 * \details The operator copies the fluxes and variances from the right side of the operator to
the left side.
 * \details If the operaFluxVector on the right side is temporary, it is deleted.
 * \param b An operaFluxVector address
 * \note Usage: operaFluxVector a = operaFluxVector b;
 * \return An operaFluxVector address
 */
operaFluxVector& operator=(operaFluxVector& b) {
        double *afluxes = (double *)fluxes;
        double *bfluxes = (double *)b.fluxes;
        double *avariances = (double *)variances;
        double *bvariances = (double *)b.variances;
        if (length < b.length) {
              throw operaException("operaFluxVector: ", operaErrorLengthMismatch, __FILE__,
__FUNCTION__, __LINE__);
              }
        unsigned n = length;
        while (n--) { *afluxes++ = *bfluxes++; *avariances++ = *bvariances++; }
        if (b.istemp) delete &b;
        return *this;
};
```

The = operator is a simple copy operation. Both the fluxes and variances are copied.

```
/*!
 * \brief Multiplication operator.
 * \details The operator multiplies the elements on the right side of the operator to the corre-
sponding elements on the left side.
 * \param b An operaFluxVector pointer
 * \note Usage: operaFluxVector t = operaFluxVector a * operaFluxVector b;
 * \return An operaFluxVector address
 */
operaFluxVector& operator*(operaFluxVector* b) {
        double *tfluxes, *tvariances;
        operaFluxVector *t = NULL;
        if (this->istemp) {
              t = this;
              tfluxes = (double *)this->fluxes;
              tvariances = (double *)this->variances;
        } else {
              t = new operaFluxVector(*this, towards, true);
```

```
            tfluxes = (double *)t->fluxes;
            tvariances = (double *)t->variances;
        }
        t->towards = this->towards;
        double *afluxes = (double *)this->fluxes;
        double *bfluxes = (double *)b->fluxes;
        double *avariances = (double *)this->variances;
        double *bvariances = (double *)b->variances;
        unsigned n = length;
        while (n--) { *tfluxes++ = *afluxes * *bfluxes; *tvariances++ = ( pow(*afluxes,2) *
*bvariances++ ) + ( pow(*bfluxes,2) * *avariances++ ); afluxes++; bfluxes++; }
        if (b->istemp) delete b;
        return *t;
};
```

here note that the fluxes are multiplied and the variances propagate by the mathematical definition of propagation of variance by multiplication. This operator definition is more complex and deals with temporaries that are created by compound expressions.

The definition of the division operator is the most complex, but also the most powerful:

```
/*!
 * \brief Division/assignment operator.
 * \details The operator divides the elements on the left side of the operator by the
corresponding elements on the right side and copies them to the left side.
 * \param b An operaFluxVector pointer
 * \note Usage: operaFluxVector a /= operaFluxVector b;
 * \return An operaFluxVector address
 */
operaFluxVector& operator/=(operaFluxVector* b) {
        double *afluxes = (double *)fluxes;
        double *bfluxes = (double *)b->fluxes;
        double *avariances = (double *)variances;
        double *bvariances = (double *)b->variances;
        unsigned n = length;
        switch (this->towards) {
            case ToDefault:
                while (n--) { *afluxes /= *bfluxes; double td = *avariances; *avari-
ances++ = ( pow(*bfluxes,-2) * td) + ( pow(*afluxes / pow(*bfluxes,2),2) * *bvari-
ances++ ); afluxes++; bfluxes++; }
                break;
            case ToINF:
                            while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes))
{*afluxes++ = FP_INFINITE; *avariances++ = 0.0; bfluxes++; bvariances++;} else
{*afluxes /= *bfluxes; double td = *avariances; *avariances++ = ( pow(*bfluxes,-2) *
td) + ( pow(*afluxes / pow(*bfluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
                break;
            case ToNAN:
                while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes)) {*afluxes++ =
FP_NAN; *avariances++ = FP_NAN; bfluxes++; bvariances++;} else {*afluxes /= *bfluxes;
double td = *avariances; *avariances++ = ( pow(*bfluxes,-2) * td) + ( pow(*afluxes /
pow(*bfluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
                break;
            case ToZero:
                while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes)) {*afluxes++ = 0.0;
*avariances++ = 0.0; bfluxes++; bvariances++;} else {*afluxes /= *bfluxes; double td =
*avariances; *avariances++ = ( pow(*bfluxes,-2) * td) + ( pow(*afluxes / pow(*b-
fluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
                break;
            case ToOne:
                while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes)) {*afluxes++ = 1.0;
*avariances++ = 0.0; bfluxes++; bvariances++;} else {*afluxes /= *bfluxes; double td =
*avariances; *avariances++ = ( pow(*bfluxes,-2) * td) + ( pow(*afluxes / pow(*b-
fluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
                break;
```

```
            default:
                break;
        }
        if (b->istemp) delete b;
        return *this;
};
```

7.21 Exception Class

The exception class is derived from exception and carries information specific to opera. There is an interface that captures, FILE, FUNCTION, and LINE number information when thrown.

```
class operaException: public exception {

private:

        operaErrorCode errorcode;// the error code itself
        string file;             // optional file of originating exception
        int line;                // optional linr number of originating exception
        string function;         // optional function name of originating exception
        string message;          // optional message

public:

        /*!
         * \class operaException()
         * \brief Basic Exception class constructor.
         * \return none
         */
        operaException();
        /*!
         * \class operaException
         * \brief operaException(operaErrorCode Errorcode)
         * \brief Basic Exception class constructor with an errorcode.
         * \param Errorcode
         * \return none
         */
        operaException(operaErrorCode Errorcode);
        /*!
         * \class operaException
         * \brief operaException(string Message, operaErrorCode Errorcode)
         * \brief Basic Exception class constructor with an additional informatonal
message and error code.
         * \param Message
         * \param Errorcode
         * \return none
         */
        operaException(string Message, operaErrorCode Errorcode);
        /*!
         * \class operaException
         * \brief operaException(string Message, operaErrorCode Errorcode, string
Filename, string Function, int Line)
         * \brief Basic Exception class constructor with an additional informatonal
message, error code, and file, function, line.
         * \param Message
```

```
         * \param Errorcode
         * \param Filename
         * \param Function
         * \param Line
         * \return none
         */
        operaException(string Message, operaErrorCode Errorcode, string Filename,
string Function, int Line);
        ~operaException() throw() {}

        /*
         * getters and setters
         */

        /*!
         * operaException::getFormattedMessage()
         * \brief Return a string containing a fully formatted error message.
         * \return string
         */
        string getFormattedMessage();

        /*!
         * operaException::setErrorCode(const operaErrorCode errcode)
         * \brief Set the error code part of an error message.
         * \return void
         */
        void setErrorCode(const operaErrorCode errcode);
        /*!
         * string operaException::getErrorCode()
         * \brief Return the error code part of an error message.
         * \param errcode const operaErrorCode
         * \return string
         */
        operaErrorCode getErrorCode();

        /*!
         * string operaException::getMessage()
         * \brief Return the string part of an error message.
         * \param m - message string
         * \return string
         */
        void setMessage(string m);
        /*!
         * string operaException::getMessage()
         * \brief Return the string part of an error message.
         * \return string
         */
        string getMessage();

        /*!
         * operaException::setLine((int l)
         * \brief Set the line number part of an error message.
         * \param l int
         * \return void
         */
        void setLine(int l);
        /*!
         * string operaException::getLine()
         * \brief Return the line number part of an error message.
         * \return int
         */
        int getLine();
```

```
        /*!
         * operaException::setFunction(string func)
         * \brief Set the function name part of an error message.
         * \param func string
         * \return void
         */
        void setFunction(string func);
        /*!
         * string operaException::getFunction()
         * \brief Return the function name part of an error message.
         * \return string
         */
        string getFunction();

        /*!
         * operaException::setFile(string func)
         * \brief Set the file name part of an error message.
         * \param func string
         * \return void
         */
        void setFile(string f);
        /*!
         * string operaException::getFile()
         * \brief Return the file name part of an error message.
         * \return string
         */
        string getFile();
};
```

7.22 operaFITSImage

An operaFITSImage is most easily created with a simple constructor:

```
operaFITSImage anImage();
```

And then a FITS image on disk is deserialized to create the object:

```
anImage << "foo.fits";
```

Alternately the constructor can take a filename and a desired type for computation on the image object:

```
operaFITSImage in("foo.fits", tfloat);
```

Here an instance of the class operaFITSImage is created from a file. The headers and the pixels are read. The pixels are requested to be of type

float, so no matter what the actual file contains, the class contains the image converted to floating point numbers.

You may also create in-memory operaFITSImages:

```
operaFITSImage out("out.fits",
    in.getnaxis1(),
    in.getnaxis2(),
    tfloat,
    cNone);
```

The operator << may be used to copy the headers and data from one object to another:

```
out << in;
```

There are other useful methods such as:

```
out.operaFITSImageSave();
out.operaFITSImageSaveAs("foocopy.fits");
out.operaFITSImageClose();
```

Object serialization, such as SaveAs can also be written as:

```
out >> "foocopy.fits";
```

These examples show the C++ feature called operator overloading. You cannot create new operators in C++ but you can define a meaning of existing operators specific to a given class. OPERA defines operators for each of the major classes that have specific meaning for images. For example, the assignment operator "=" is overloaded:

```
out = 0.0;    // zero the whole image
```

In this case, the assignment operator of a float is defined to mean: set every pixel in the image to the floating point value zero.

```
out = in;      // copy the pixels from in to out
```

Here the pixels (not the headers, as in the << operator case) are copied from one image to another.

The operators +=, -=, *=, /= also have meaning:

```
flat -= bias;              // remove the bias from the flat
out /= flat;               // divide out by the flat
out *= badPixelMask;       // mask out
out += 300.0;              // add a constant bias to out
```

The binary operators +, /, *, - also have meaning:

```
// out is the pixel-by-pixel difference between the flats
out = flat1 - flat2;
// mask out with the badpixel mask
out = out * badpixelImage;
```

The operators <, >, <=, >= also have meaning:

```
out *= out > 0.0;    // zero any negative values
```

Here `out` > 0.0 creates an in-memory pixel map of 1.0's and 0.0's of the dimensionality of anImage, where there is a 1.0 wherever anImage has a positive pixel values, else 0.0. This map is then multiplied by `out`, effectively masking all negative pixels from `out`.

Take care that this expression:

```
out = out > 0.0;     // save the mask in out
```

is very different, because here anImage takes the value of the mask itself, becoming all 1.0's and 0.0's.

Note that binary operators create in-memory temporary images that are reclaimed after use automatically by the operator definition.

It is also possible to get the value of a pixel by using the [] operator:

```
// print some pixel values from the image
// Note that y is the first dimension
for (unsigned y=0; y<2048; y++) {
     for (unsigned x=0; x<2048; x++) {
          cout << out[y][x] << ' ';     // print the pixel value
          out[y][x] = 0.0;              // zero the pixel value
     }
     cout << endl;
}
```

So, you can see that while most operators allow the user to operate on entire images, it is also possible to index to particular pixel positions using the [] operator both in assignment and referencing.

There is also an ImageIndexVector type defined that is created by the "where" function, which has similar semantics to the "where" function in IDL:

```
// store the null-terminated vector of indices into vector
// ImageIndexVectors are one-based image positions
ImageIndexVector vector = where(flatImage>0.0);
```

An ImageIndexVector is defined as a null-terminated, one-based vector of image indices, where the image is treated as being single dimensioned. The where function is useful for creating bins of image pixel values:

```
// mask pixels in a range of ADU values
out[where(in >= 400.0 && in <= 800.0)] = -1.0;
```

This example introduces a number of concepts.

Firstly the || (or) and && (and) operators are defined for operaFITSImages. The meaning assigned to && is: for every pixel if the either the corresponding lhs is zero or the rhs is zero, then 0.0, else 1.0. The operator || means: for every pixel if the either the corresponding lhs is non-zero or the rhs is non-zero, then 1.0, else 0.0.

Secondly, the where function takes an operaFitsImage and returns a null-terminated, one-based vector of operaFITSImage *indices* where the operaFITSImage is non-zero.

Thirdly, a operaFITSImage can be multiply indexed from another image.

The unary ! operator is also defined. It has the same meaning as ! in C++, except that it creates a map of every pixel.

```
// mask anImage with the inverse of badpixel mask
out *= !badPixelMask;
```

Here is an interesting snippet of executable C++ code that creates an operaFITSImage, removes the bias image, divides by a flat image, masks bad pixels, finds the maximum pixel value and normalizes the entire image!

```
operaFITSImage out("out.fits", tfloat);
out -= bias;                        // remove the bias from the image
out /= flat;                        // divide the image by the flat
out *= badPixelMask;                // mask with the badpixel mask
float maxvalue = anImage.max();     // find the maximum pixel value
out /= maxvalue;                 // normalize the image
```

Whew! And it is fast inlined executable code, compiled directly from C++. How fast is it? There is a small test program in opera-1.0/test called operaAsmTest.cpp.

```
void foo(operaFITSImage *a, operaFITSImage *b) {
      *a -= *b;
}
```

```
int main()
{
      operaFITSImage a("a.fits");
      operaFITSImage b("b.fits");
      return 0;
}
```

Image b (say a bias) is deleted pixel by pixel from image a. The operation is isolated in function foo for convenience. The machine instructions produced by gcc are:

```
void foo(operaFITSImage *a, operaFITSImage *b) {
          0:     55                      push   %ebp
          1:     89 e5                   mov    %esp,%ebp
          3:     56                      push   %esi
          4:     53                      push   %ebx
          5:     83 ec 10                sub    $0x10,%esp
          8:     8b 45 08                mov    0x8(%ebp),%eax
b:        8b 75 0c                                mov    0xc(%ebp),%esi

          operaFITSImage& operator-=(operaFITSImage& b) {
                float *p = (float *)pixptr;
                float *bp = (float *)b.pixptr;
                unsigned n = npixels;
          e:    8b 50 2c                mov    0x2c(%eax),%edx
                operaFITSImage& operator-=(operaFITSImage& b) {
                      float *p = (float *)pixptr;
                      11:     8b 48 3c                mov    0x3c(%eax),%ecx
                      float *bp = (float *)b.pixptr;
                      14:     8b 5e 3c                mov    0x3c(%esi),%ebx
                      unsigned n = npixels;
                      while (n--) *p++ -= *bp++;
                      17:     85 d2                   test   %edx,%edx

                      19:     74 16                   je     31 <_Z3fooP14operaFITSImageS0_+0x31>
                      1b:     31 c0                   xor    %eax,%eax
                      1d:     8d 76 00                lea    0x0(%esi),%esi
                      20:     d9 04 01                flds   (%ecx,%eax,1)      <-- top of assign loop
                      23:     d8 24 03                fsubs  (%ebx,%eax,1)      | 2
                      26:     d9 1c 01                fstps  (%ecx,%eax,1)      | 3
                      29:     83 c0 04                add    $0x4,%eax          | 4
                      2c:     83 ea 01                sub    $0x1,%edx          | 5
                      2f:     75 ef                   jne    20                 <-- bottom of assign loop
                      if (b.istemp) delete &b;
                      31:     80 7e 38 00             cmpb   $0x0,0x38(%esi)
                      35:     74 19                   je     50 <_Z3fooP14operaFITSImageS0_+0x50>
                      37:     89 34 24                mov    %esi,(%esp)
                      3a:     e8 fc ff ff ff          call   3b <_Z3fooP14operaFITSImageS0_+0x3b>
                      3f:     89 75 08                mov    %esi,0x8(%ebp)
                      *a -= *b;
                }
                42:     83 c4 10                add    $0x10,%esp
                45:     5b                      pop    %ebx
                46:     5e                      pop    %esi
                47:     5d                      pop    %ebp
                48:     e9 fc ff ff ff          jmp    49 <_Z3fooP14operaFITSImageS0_+0x49>
                4d:     8d 76 00                lea    0x0(%esi),%esi
                50:     83 c4 10                add    $0x10,%esp
                53:     5b                      pop    %ebx
                54:     5e                      pop    %esi
                55:     5d                      pop    %ebp
                56:     c3                      ret
                57:     89 f6                   mov    %esi,%esi
                59:     8d bc 27 00 00 00 00    lea    0x0(%edi,%eiz,1),%edi
```

You can see that the inner loop where the pixels are processed consists of just 6 instructions! Look for the annotation top of assign loop and bottom of assign loop. These instructions are executed NAXIS1 * NAXIS2 times. The function entry and exit instructions are executed only once.

7.23 operaMultiExtensionFITSImage

The operaMultiExtensionFITSImage class encapsulates the FITS notion of a FITS Image that contains multiple images. Ordinarily this would be used to store images with more than one detector chip:

```
operaMultiExtensionFITSImage in("foo.fits", tfloat);
```

Here a "MEF" image object is created.

All of the operators defined for operaFITSImages also work for operaMultiExtensionFITSImage, but the indexing operator [] has a special meaning:

```
operaFITSImage out(in[4]);
```

In this example a regular FITSImage object is created from extension 4 of the MEF FITS image. The definition of [] for the operaMultiExtensionFITSImage class carries the meaning of indexing into an extension, so to access pixels in an image:

```
unsigned lastextension = in.getNExtensions();
for (unsigned extension=1; extension<=lastextension; extension++) {
     cout << in[extension][1024][1024] << endl;
}
```

The constructor for the operaMultiExtensionFITSImage class adds an additional argument: *isLazy*. The operaMultiExtensionFITSImage supports lazy reads, meaning that an extension is not read from disk until it is referenced and it is not written to disk until it takes on a new value. Only one extension is in memory at any one time. Very large images cannot be stored in en-

tirety in memory on 32 bit architecture hardware. The lazy read permits operations on large images where I/O is largely transparent to the user. For example:

```
operaMultiExtensionFITSImage in("foo.fits", tfloat);
operaMultiExtensionFITSImage out("out.fits",
     in.getXDimension(),
     in.getYDimension(),
     inImage.getNExtensions(),
     tfloat);

unsigned lastextension = in.getNExtensions();
for (unsigned extension=1; extension<=lastextension; extension++) {
     out[extension] = in[extension];
}
```

This example is interesting because the operaMultiExtensionFITSImage objects are by default lazy. That means the extension is not read until requested and the extension is not written until assigned. So, in the loop above, first extension 1 of in is read from disk, then the pixels are copied into the out object in memory, and then stored to disk, since out[] was assigned. Then the next iterations of the loop to reads/writes for each extension happen automatically. The user need not be concerned about file I/O, but can rather concentrate on the algorithm.

7.24 operaMultiExtensionFITSCube

The operaMultiExtensionFITSCube extends the operaMultiExtensionFITSImage class to handle the FITS image notion of a FITS Cube. The meaning of the [] operator is defined for this class to index into a slice:

```
unsigned lastextension = in.getNExtensions();
unsigned lastslice = in.getNSlices();
for (unsigned extension=1; extension<=lastextension; extension++) {
     out[extension] = in[extension];
     for (unsigned slice=1; slice<=lastslice; slice++) {
           cout << out[extension][slice][1024][1024] << endl;
     }
}
```

The operaMultiExtensionFITSCube inherits the laziness of the The operaMultiExtensionFITSImage class. The [] operator does add another layer of granularity of image access. In this example, slices rather than extensions are read/written:

```
operaMultiExtensionFITSCube in(“foo.fits”, tfloat);
operaMultiExtensionFITSCube out("out.fits",
     in.getXDimension(),
     in.getYDimension(),
     in.getZDimension(),
     inImage.getNExtensions(),
     tfloat);

unsigned lastext = in.getNExtensions();
unsigned lastslice = in.getNSlices();
for (unsigned extension=1; extension<=lastext; extension++) {
     for (unsigned slice=1; slice<= lastslice; slice++) {
          out[extension][slice] = in[extension][slice];
     }
}
```

If available memory is very tight, single slices of single extensions may be read/written.

7.25 Casting and Converting Classes

The opera FITS image classes are object-oriented and hierarchical and so may be inter-converted. This is an example of the creation of a Cube from 3 WIRCam images taken with different filters of the same object. The output Cube is similar to the color cube taken by a DSLR camera. This example is abstracted from the “wircolorcomposite” tool that is distributed with OPERA, used to create false color WIRCam images. First, we create the three rgb input image objects:

```
operaWIRCamImage red(r, tfloat, READONLY);
operaWIRCamImage green(g, tfloat, READONLY);
operaWIRCamImage blue(b, tfloat, READONLY);
```

Next, we create the output 3 - Cube and populate the headers from one of the inputs:

```
operaFITSCube out(output,
     red.getXDimension(),
     red.getYDimension(),
     3,
     tfloat);
out.operaFITSImageCopyHeader(&red);
```

Next, we create the output Cube and populate the headers from one of the inputs:

```
// median collapse to get rid of cosmic rays and reduce noise
red.medianCollapse();
green.medianCollapse();
blue.medianCollapse();
```

In this step we create regular fits images of the 4th extension, where the image we want resides:

```
// create FITS images of the 4th extension
operaFITSImage redfits(red[4]);
operaFITSImage greenfits(green[4]);
operaFITSImage bluefits(blue[4]);
```

Now remove the chip bias offset, and mask the guide windows and bad pixels to the image median value (do the same for green and blue):

```
/*
 * 1. remove the chip bias
 * 2. get the image median
 * 3. set the inverse of the badpix to image median
 * 4. mask the guidewindow to image median
 */
redfits -= red.getChipBias();
float imageMedian = operaArrayMedian(redfits.getnpixels(),
     (float *)redfits.getpixels());
*redfits[where(!badpixfits)] = imageMedian;
*redfits[new operaImageVector(red.getGuideWindow(extension),1)] = imageMedian;
...
```

Finally copy the red, the green, and the blue images to the three axes of the output cube and save it:

```
// copy each color into the cube
out[1] = redfits;
```

```
        out[2] = greenfits;
        out[3] = bluefits;

        out.operaFITSImageSave();
```

We are done! We manipulated various opera FITS Image classes to create the final Cube product.

7.26 Matrix

No, not that matrix... The indexing operator [] defined for the operaFITSImage class is useful only to C++ programs. However many of the OPERA image and statistics libraries are written in C. There is a type *Matrix*, type-defined as a float **, that allows C functions to access the FITS images using the [y][x] notation. The matrix base is created every time a class image is created. The first dimension is a vector of length naxis2, which is a vector of float* to the address of each row in the image. It is used thusly:

```
// get a Matrix to pass to a C function
        Matrix matrix = anImage.getmatrix;
             fooInC(matrix);

        void fooInC(float **matrix) {
             for (unsigned y=0; y<maxy; y++) {
                   for (unsigned x=0; x<maxx; x++) {
                         cout << matrix[y][x] << ' ';     // output the pixel value
                   }
                   cout << endl;
             }
        }
```

Here is an example of iteration by column by reversing the for loops in the image:

```
for (unsigned x=1024; x<1032; x++) {
        for (unsigned y=2048; y<2056; y++) {
             matrix[y][x] = matrix[x][y];
        }
}
```

7.27 operaEspadonsImage

An operaEspadonsImage is an extension of operaFITSImage. It is most easily created by definition, from an existing FITS file:

```
operaEspadonsImage *in = new operaEspadonsImage("foo.fits", tfloat);
```

While an operaEspadonsImage contains extended information unique to espadons images, the most important attribute is that the image pixels are from the datasec, and the overscan pixels are not included in the image. Here are some examples of the extended information:

```
cerr << "IMTYPE= " << in->getimtype() << ' ' << in->getimtypestring() << '\n';
cerr << "DETECTOR= " << in->getdetector() << ' ' << in->getdetectorstring() << '\n';
cerr << "AMPLIFIER= " << in->getamplifier() << ' ' << in->getamplifierstring() << '\n';
cerr << "MODE= " << in->getmode() << ' ' << in->getmodestring() << '\n';
cerr << "SPEED= " << v->getspeed() << ' ' << v->getspeedstring() << '\n';
cerr << "STOKES= " << in->getstokes() << ' ' << in->getstokesstring() << '\n';
cerr << "POLAR QUAD= " << in->getpolarquad() << ' ' << in->getpolarquadstring() << '\n';
```

The mode, speed, detector, enums are defined in the header file and will not be further detailed here.

Note that these examples use an operaEspadonsImage pointer, so access to the operators is a little different:

```
// Note the fairly tricky *Out notation here, as Out is a pointer
*Out -= 100.0;                  // remove bias
*Out *= *Out < 35000.0;         // remove saturated pixels
// print some pixel values from the center of the image
for (unsigned y=2048; y<2056; y++) {
     for (unsigned x=1024; x<1032; x++) {
          cout << (*Out)[y][x] << ' ';    // output the pixel value
          (*Out)[y][x] = 0.0;             // zero the pixel value
     }
}
cout << endl;
```

The datasec portion of the image is stored in the private member datasecSubImage which has a getter:

```
/*! * operaFITSSubImage *getDatasecSubImage()
 * \brief get the datasec subImage
 * \return subImage pointers */
operaFITSSubImage *getDatasecSubImage();
```

This is the preferred access method to the espadons image data, for example:

```
operaFITSSubImage *subImage = In.getDatasecSubImage();
*subImage += 100.0;                    // add bias
*subImage *= *subImage < 35000.0;      // remove pixels above 35000 ADU
for (unsigned y=2048; y<2056; y++) {
     for (unsigned x=1024; x<1032; x++) {
          cout << (float)(*subImage)[y][x] << '';
     }
     cout << endl;
}
cout << endl;
```

Note that the subImage is a copy of the espadons image, and must be set back into the full espadons image if the subImage pixels are changed:

```
// now, if we ever change the subImage, which is a copy
// we need to set it back in to the espadons image
Out.operaFITSImageSetData(*subImage);
```

7.28 operaWIRCamImage

The operaWIRCamImage class describes an object that has specific WIRCam pieces. It is derived from the operaMultiExtensionFITSCube, but adds:

- exposure time
- chip bias
- sky levels
- guide cube boxes

There are also a number of methods available for WIRCam images:

```
/*
* void detrend(operaWIRCamImage &image, operaWIRCamImage &dark, operaWIRCamImage &badpixelmask, operaWIRCamImage *flat)
* \brief detrend a WIRCam image. all extensions in parallel
*/
void operaWIRCamImage::detrend(operaWIRCamImage &image, operaWIRCamImage &dark, operaWIRCamImage &badpixelmask, operaWIRCamImage *flat);
/*
* void weightmap(operaWIRCamImage &image, operaWIRCamImage &badpixelmask)
* \brief create a weightmap of bad and saturated pixxels of a WIRCam image. all extensions in parallel
```

```
        */
        void operaWIRCamImage::weightmap(operaWIRCamImage &image, operaWIRCamImage &badpixel-
mask);

         /*
         * void skySubtraction(operaWIRCamImage &image, operaWIRCamImage &flat, operaWIRCamImage
&dark)
         * \brief sky subtract a WIRCam image
         */
        void operaWIRCamImage::skySubtraction(operaWIRCamImage &image, operaWIRCamImage &sky);

         /*
         * void maskGuideWindows(operaWIRCamImage &image)
         * \brief mask the entire rows and columns at the guide window positions in a
WIRCam image
         */
         void operaWIRCamImage::maskGuideWindows(operaWIRCamImage &image);
         /*
         * void masterFlat(operaWIRCamImage &image[], operaFITSImage *weight, count)
         * \brief create a master flat from a medianCombined cube
         * \brief optionally xreating a weight map
         */
         void operaWIRCamImage::masterFlat(operaWIRCamImage *images[], operaWIRCamImage *weight,
unsigned count);
         /*
         * void masterDark(operaWIRCamImage &image)
         * \brief create a master dark by median combining the stack
         */
         void operaWIRCamImage::masterDark(operaWIRCamImage *images[], unsigned count);
         /*
         * void subtractReferencePixel(operaWIRCamImage &image)
         * \brief subtract reference pixels from WIRCam image
         */
         void operaWIRCamImage::subtractReferencePixels(operaWIRCamImage &image);
         /*
         * void calculateSkyBackground(operaWIRCamImage &image)
         * \brief Calculate the Sky Background - the median / exposure time
         * Could also be gotten from the headers SKYLVL, SKYDEV -- if they exist
         */
         void operaWIRCamImage::calculateSkyBackground(operaWIRCamImage &image);

         /*
         * void calculateQERatio(operaWIRCamImage *images[], unsigned count)
         * \brief Calculate the QE Ratio - the median sky rate over the images
         * Could also be gotten from the headers SKYLVL, SKYDEV -- if they exist
         */
         void operaWIRCamImage::calculateQERatio(operaWIRCamImage *images[], unsigned count);
         /*
         * void createSky(operaWIRCamImage *images[], unsigned count)
         * \brief Create a master sky image
         */
         void operaWIRCamImage::createSky(operaWIRCamImage *images[], unsigned count);
         /*
         * void correctSkyLevel(operaWIRCamImage *images[], unsigned count)
         * \brief Correct sky level
         */
         void operaWIRCamImage::correctSkyLevel(operaWIRCamImage *images[], unsigned count);
```

Details may be found in the doxygen documentation.

8. Preliminary Results

9.1 Extraction Aperture Calibration

Here we present some of the results obtained with the prototypes developed for the aperture calibration module.

Figure 1 below shows the instrument profile for ESPaDOnS star-only mode. The sub-pixels covered by the aperture are set to zero so one can visualize the aperture position with respect to the illumination profile. The aperture in this case is a single beam and the gap was chosen to be zero between the beam and the background windows.

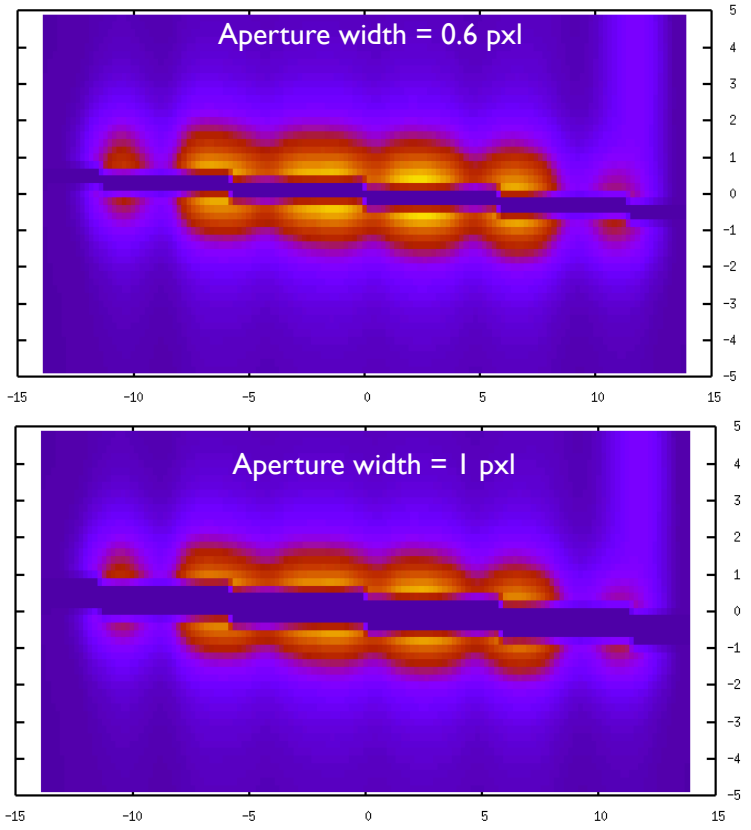


Figure 1 - Two examples of single beam (star-only) extraction apertures.

Top panel in Figure 2 shows an aperture with 2 beams, which can be used for either polarimetry or star+sky modes. In this example we used an aperture with 1-pixel wide gap, aperture width of 26 pixels and height of 0.6 pixel. The background aperture width is 2 pixels. In middle panel the gap width was set as 3 pixels, and the bottom panel shows an aperture with height = 1.2 pixel.

Figure 3 presents a plot of extraction apertures for all spectral orders.

Figure 4 shows the measurement of the tilt angle for the maximum flux fraction for each order. The median value and uncertainty limits are also shown. The final tilt measurement for this data set is -2.4 +/- 0.2 degrees.

Figure 5 presents the flux fraction measured inside the aperture versus order number. The aperture has a constant area whereas the illumination profile changes with wavelength. The reason for this change is due to the change in the pixel-to-wavelength scale, which affects the amount of encircled energy collected by an aperture of fixed size (in pixel units). These results show that an ideal aperture that collects constant amount of flux per spectral element should vary in size with wavelength in order to keep a constant sampling with respect to the resolution element. The plot in Figure 4 also shows that the dependence with wavelength is smooth for ESPaDOnS and therefore could be easily modeled and provide an adaptable aperture size.

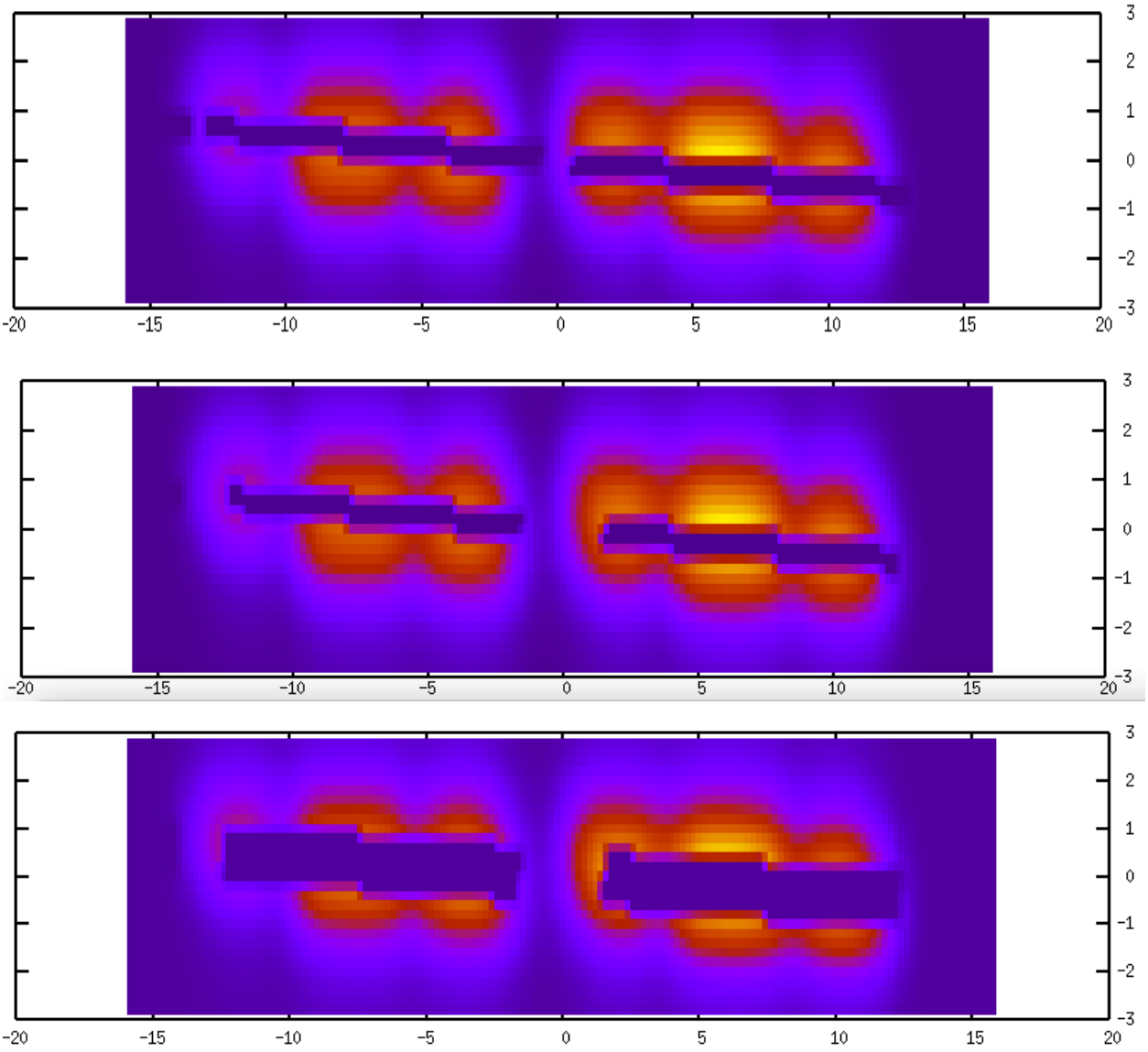


Figure 2 - Extraction Apertures. Top panel: 28 x 0.6 pixel aperture, with 2 beams, and 2-pixel background windows with gap of 1 pixel. Middle panel: same configuration as in top panel but with gap of 3 pixels. Bottom panel: same configuration as in middle panel but with height of 1.2 pixel.

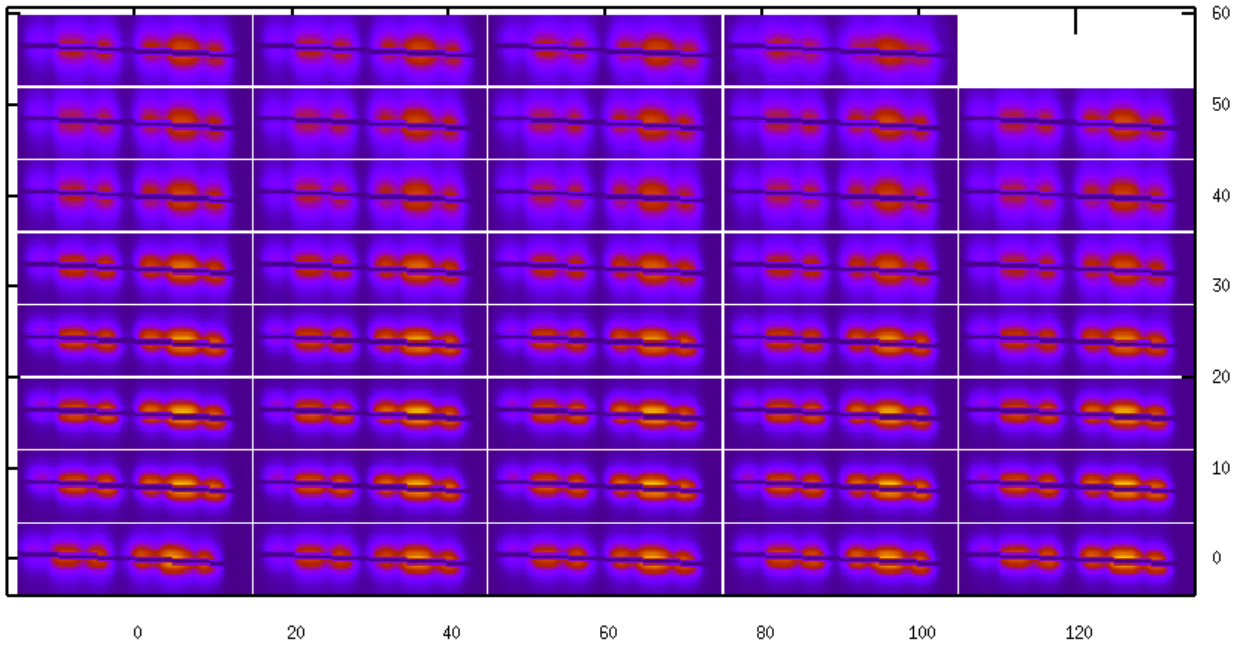


Figure 3 - Extraction Apertures for orders 22 to 60. The aperture configuration is 2 beams, aperture dimensions of 28 x 0.6, with no gap, and background of 2 pixels.

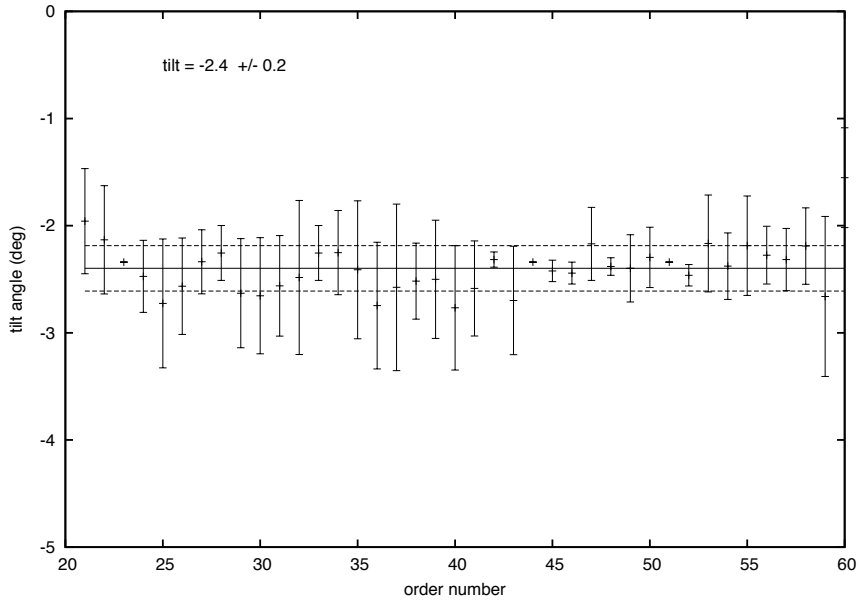


Figure 4 - Tilt measurements.

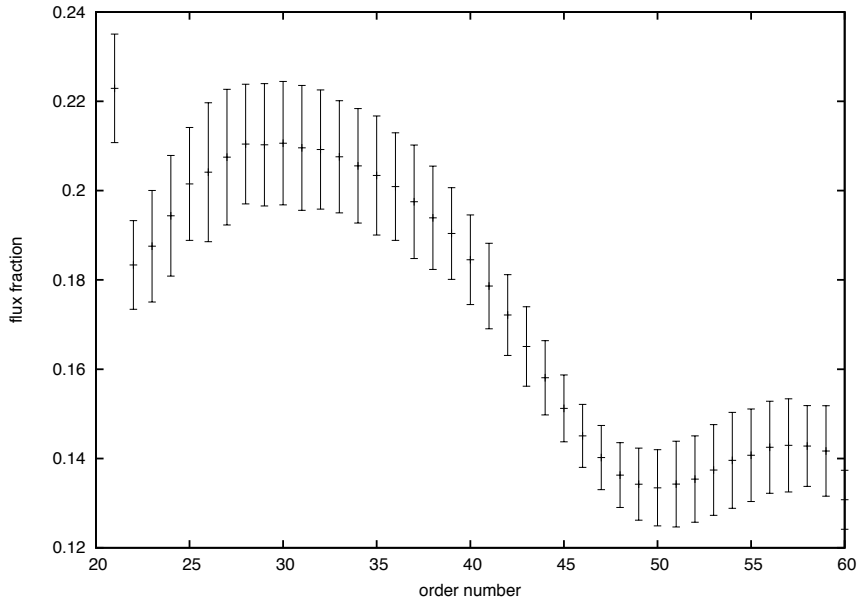


Figure 5 - Flux fraction measured inside the aperture for all orders.

8.2 Wavelength Calibration

In this section we present some of the preliminary results obtained with OPERA for the wavelength calibration module.

Figure 6 is a screen shot of information printed out on the screen during calibration of order number 56. One can see the information obtained from geometry calibration, which gives the pixel range and also the wavelength initial guess solution, which defines the range of wavelength covered by the order and therefore the approximate range within which to select atlas data points. Note that the line detection algorithm has found 617 lines in the raw uncalibrated comparison spectrum and 825 lines in the atlas spectrum. Note that the atlas has a higher resolution than the observed data.

Also in Figure 6 one may notice the result for the initial solution, which is the first guess improved by the cross-correlation search method. This al-

lowed the matching algorithm to identify about 57% of all lines in the atlas and 77% of all lines in the comparison. The final solution and other additional information like radial velocity precision, wavelength precision, and spectral resolution are presented at the bottom.

Figure 7 presents the radial velocity precision versus order number. Note that the first two orders in the red side (left) and the last in the blue present relatively worse precision. The reason for that is because the number of lines is not as high as for other orders due to the high vigneting at the borders of the detector. Another interesting case to note is of order 35, which presents a considerably worse calibration compared to other orders. This order is the most affected by highly saturated lines at that region of the detector. Other orders around the same region suffer from the same effect, but less noticeable. This problem could be minimized if the saturation level could be minimized.

Figure 8 shows the measured spectral resolution for each order. The resolution is measured as the central wavelength divided by the median line width.

```
operaWavelengthCalibration: Order 56: [geom] ymin = 3.00 ymax = 4600.00 dmin = 0.00 dmax = 4613.22
operaWavelengthCalibration: Order 56: [wave] wavelength selected range: wl0 = 398.62 wlc = 411.70 wlf = 423.41
operaWavelengthCalibration: Order 56: [Comparison] 617 lines in comparison between wl0 = 402.76 and wlf = 419.26.
operaWavelengthCalibration: Order 56: [Atlas] 825 lines detected in input atlas between wl0 = 398.62 and wlf = 423.41 .

operaWavelengthCalibration: Order 56: Initial Solution:
operaWavelengthCalibration: Order 56: par[0]=403.906 par[1]=0.00419323 par[2]=-1.29853e-07  chisqr=0.000202228
operaWavelengthCalibration: Order 56: 475 lines matched between wl0 = 403.91  wlf = 420.49.
operaWavelengthCalibration: Order 56: [Atlas]    matched 57.58 % of detected lines.
operaWavelengthCalibration: Order 56: [Comparison] matched 76.99 % of detected lines.
operaWavelengthCalibration: Order 56: Final Solution:
operaWavelengthCalibration: Order 56: par[0]=403.896 par[1]=0.00423179 par[2]=-1.36275e-07  chisqr=3.85966e-08
operaWavelengthCalibration: Order 56: 27 lines matched between wl0 = 403.90  wlf = 420.52.
operaWavelengthCalibration: Order 56: [Atlas]    matched 3.27 % of detected lines.
operaWavelengthCalibration: Order 56: [Comparison] matched 4.38 % of detected lines.
operaWavelengthCalibration: ----------------------------------------------------------------
operaWavelengthCalibration: Order 56: Radial velocity precision = 134.48 m/s. Using 27 spectral lines.
operaWavelengthCalibration: Order 56: Wavelength RMS precision = 0.0001852245 nm  Median Precision = 0.0001086108 nm.
operaWavelengthCalibration: Order 56: [Comparison Lines] median sigma = 0.82 +/- 0.26.
operaWavelengthCalibration: Order 56: Spectral Resolution = 61335.43 +/- 9854.20.
operaWavelengthCalibration: ----------------------------------------------------------------
```

Figure 6 - Screenshot of the information displayed during wavelength calibration for order 56.

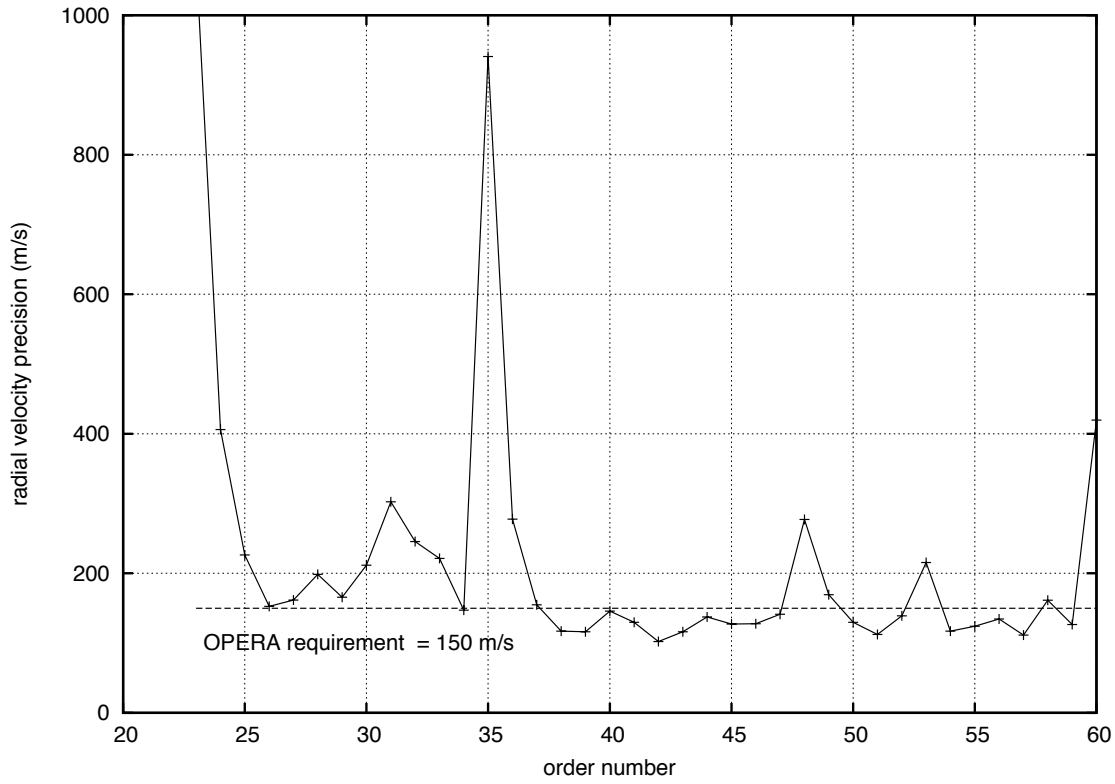


Figure 7 - Radial velocity precision.

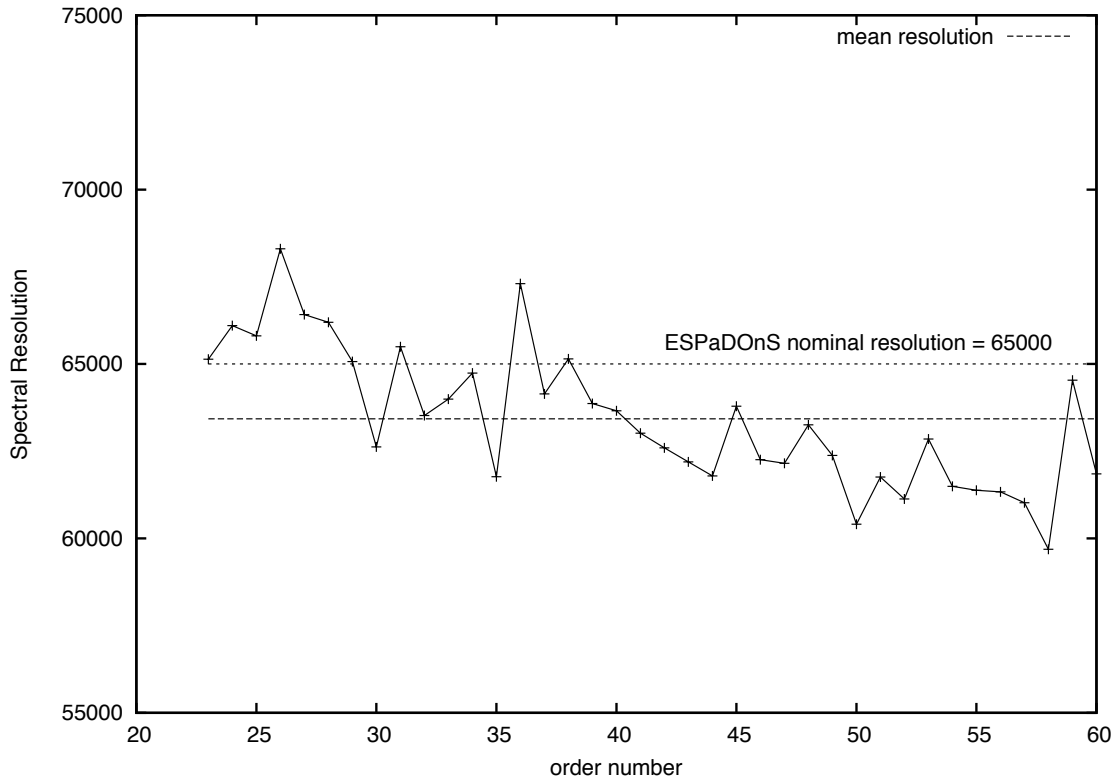


Figure 8 - Spectral resolution versus order number.

8.3 Normalization

In this section we present some of the results obtained with the normalization module.

Figure 9 shows two extracted spectra for orders 34 and 35. Also in the same plot we present the sample points obtained with the algorithm that detects the continuum. The full continuum is modeled as an interpolation of these points. In Figure 10 we present the same spectra normalized by this continuum.

Note that order 34 (red) presents a considerably large number of absorption lines and order 35 (in green) contains a wide Hydrogen-alpha absorp-

tion line at ~656.28 nm. Those two cases were carefully chosen as examples to show the robustness of the normalization algorithm. A choice of a larger bin size would probably provide a better estimate of the continuum but for some lower orders this bin size wouldn't be ideal. So, this is compromise of choosing a single bin size that works relatively well for all orders. A future improvement to the algorithm would be to define a variable bin size which would give an optimal value for all orders.

Another point to be noted in Figure 10 is the presence of a sharp cosmic ray in order 34 (red). This is ignored by the algorithm, as expected.

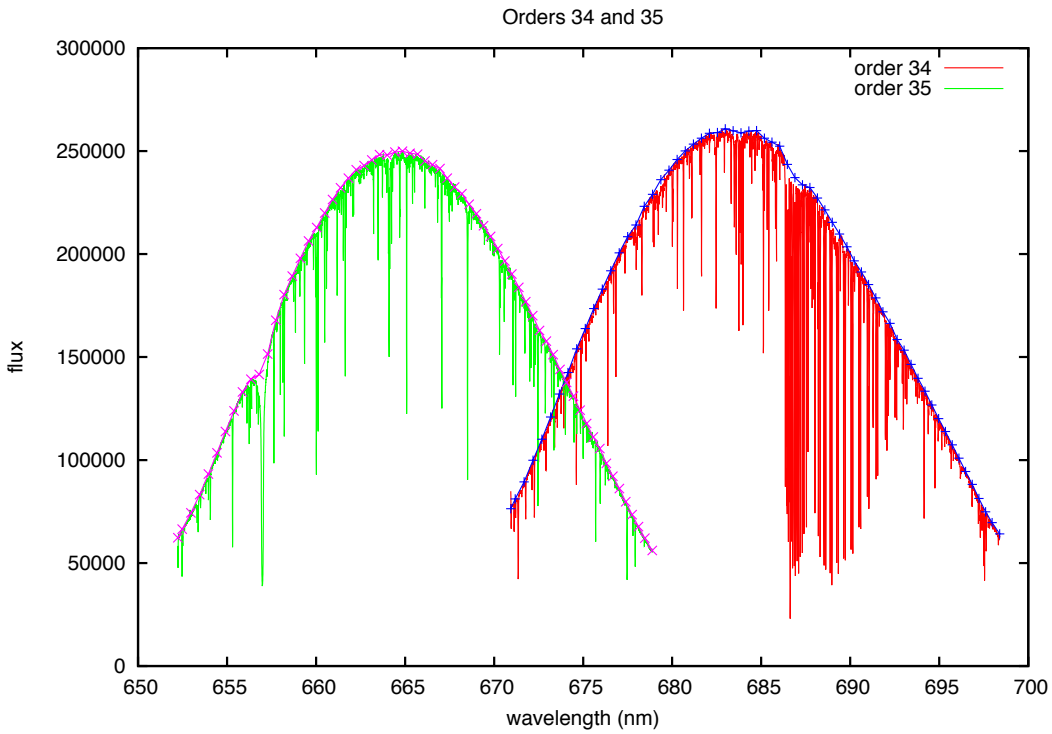


Figure 9 - Two extracted spectra for orders 34 and 35. Magenta and blue crosses show the sample points representing the detected continuum and lines connecting those points are the continuum model.

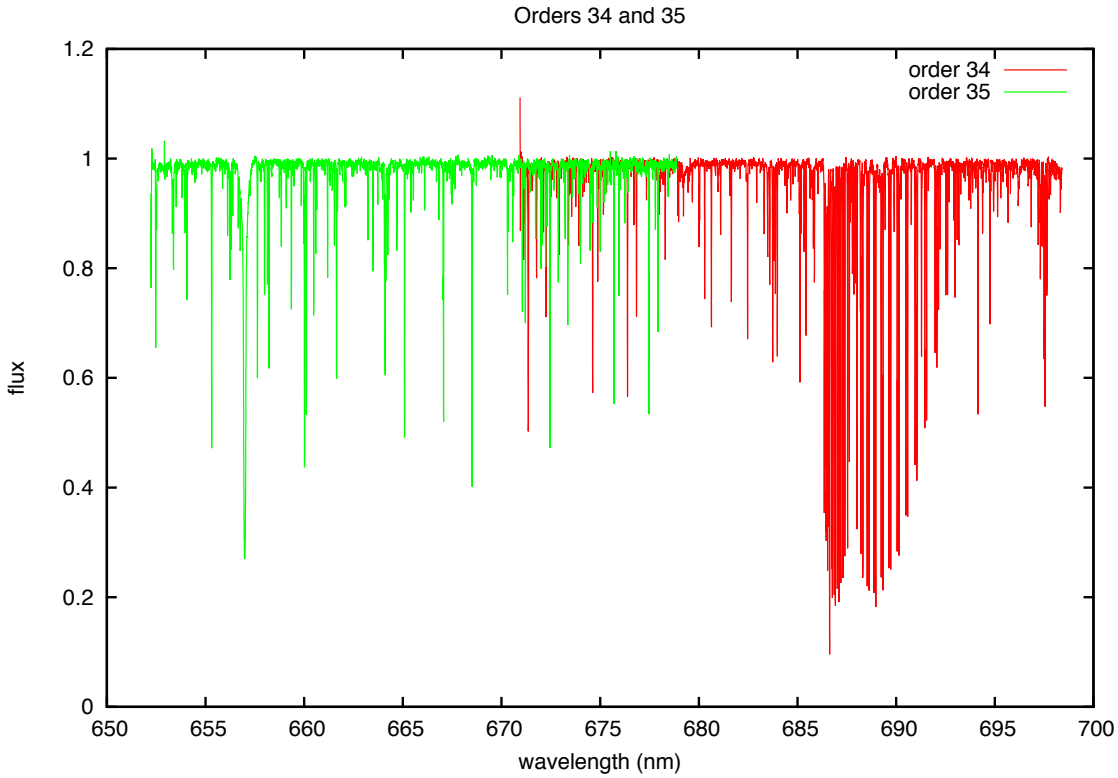


Figure 10 - Normalized spectra to the continuum.

8.4 Flux Calibration

This module has not been prototyped yet.

8.5 Polarimetry

In this section we present the results obtained using the operaPolar module.

In order to test the polarimetric module we developed some simulation tools for polarimetry. These tools allow us to produce simulated data for polarimetric spectrum, which can be used as input to the operaPolar module, then the results can be checked against the input polarization.

Figure 11 shows the degree of polarization measured for a simulated spectrum. The intensity input beam is simulated as a black body spectrum at 5000K from 300nm to 700nm, with degree of polarization in Q set to 10%. There is a gaussian dispersion of 10% introduced to simulate errors in the intensities. The input beam is represented by a Stokes vector. The vector then goes through a simulation of ESPaDOnS polarimetric module using Mueller matrices. The output beam is then analyzed using the polarimetric module of OPERA. As shown on Figure 11, the measured degree of polarization is 10%, which demonstrate that the Mueller matrices can effectively simulate optical components and that the polarimetric module calculates the same polarization as the one in the input beam.

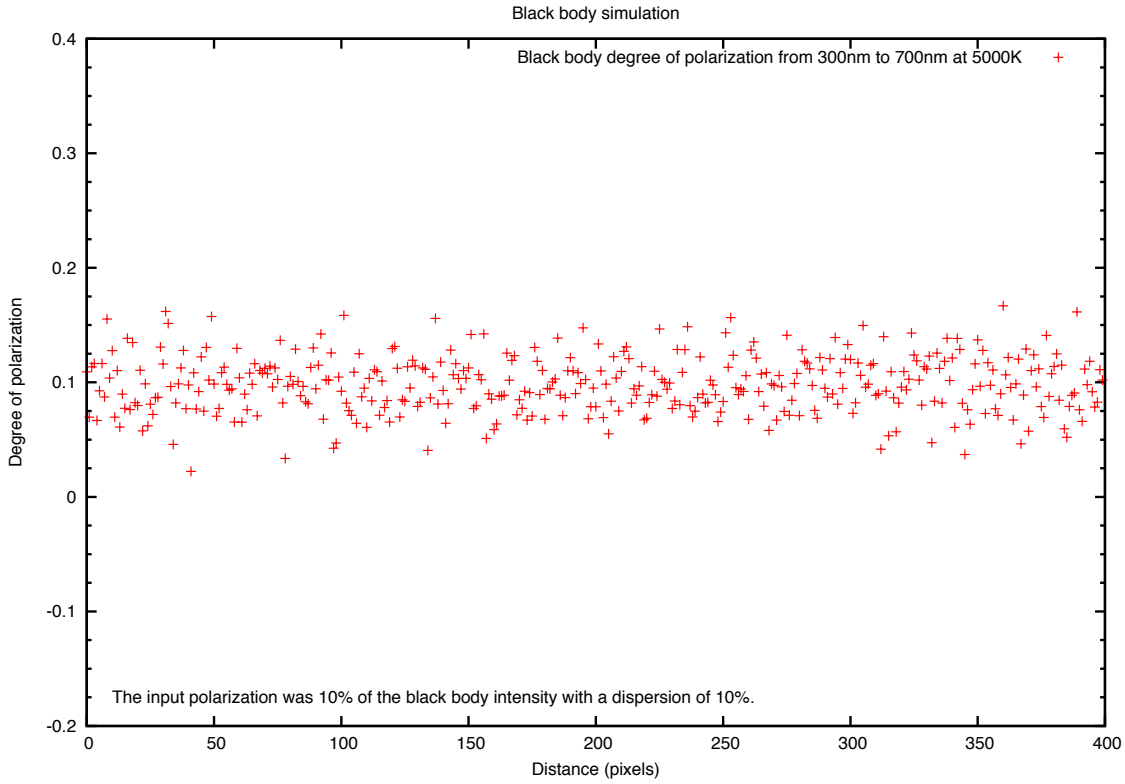


Figure 11 - OPERA polarization measurements of a simulated black body spectrum. The input polarization was 10% of the black body intensity with a gaussian dispersion of 10%.

Figure 12 shows the Probability Density Function (PDF), which represents the dispersion of the degree of polarization of data presented in Figure 11. The standard deviation is about 0.025%.

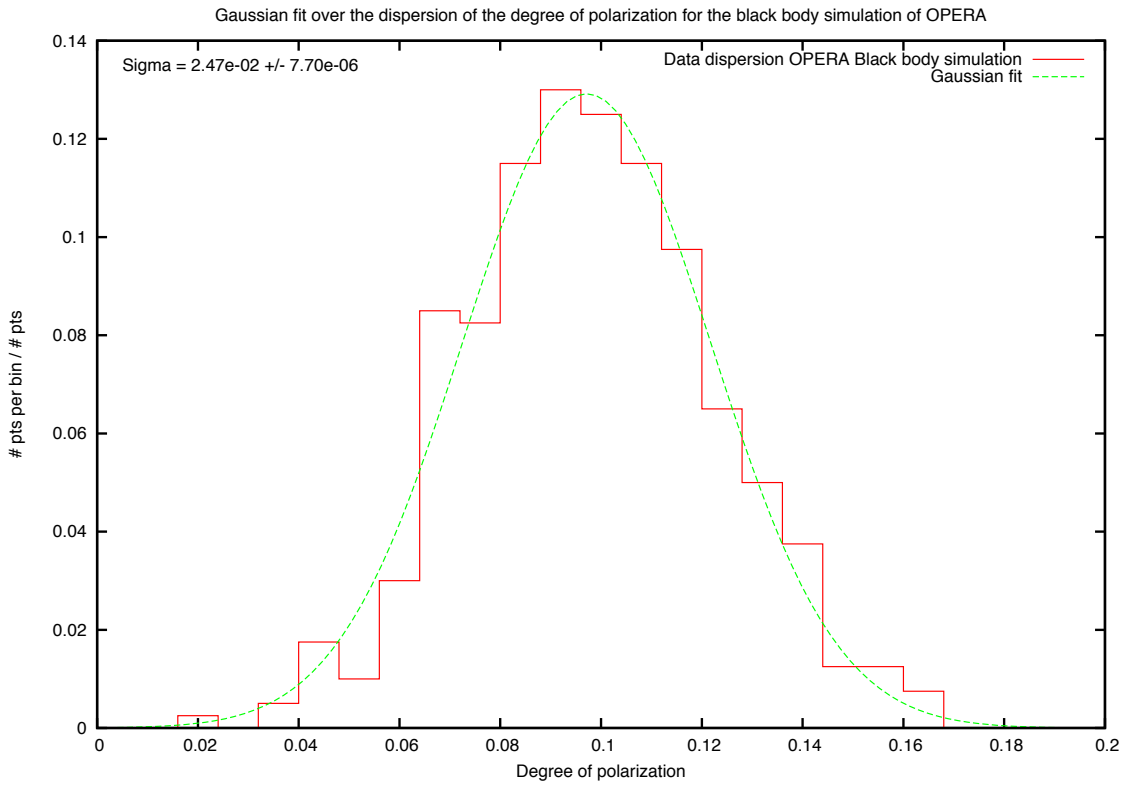


Figure 12 - Dispersion of the degree of polarization of Figure 11.

Figure 13 shows the degree of linear polarization measured for a star that presents a feature that can be detected in a single spectral line using the Libre-Esprit reduction pipeline. As one can notice in this plot, the OPERA results are comparable to Libre-Esprit presenting relatively good levels of noise. As shown in this example, this precision allows one to detect a feature with polarization amplitude of ~0.005%. On Figure 14 we present the null polarization spectra calculated by OPERA and Libre-Esprit. In this example, the high levels of variations in the null spectra indicate that the feature on the left seen in Figure 13 is probably not a real polarization feature.

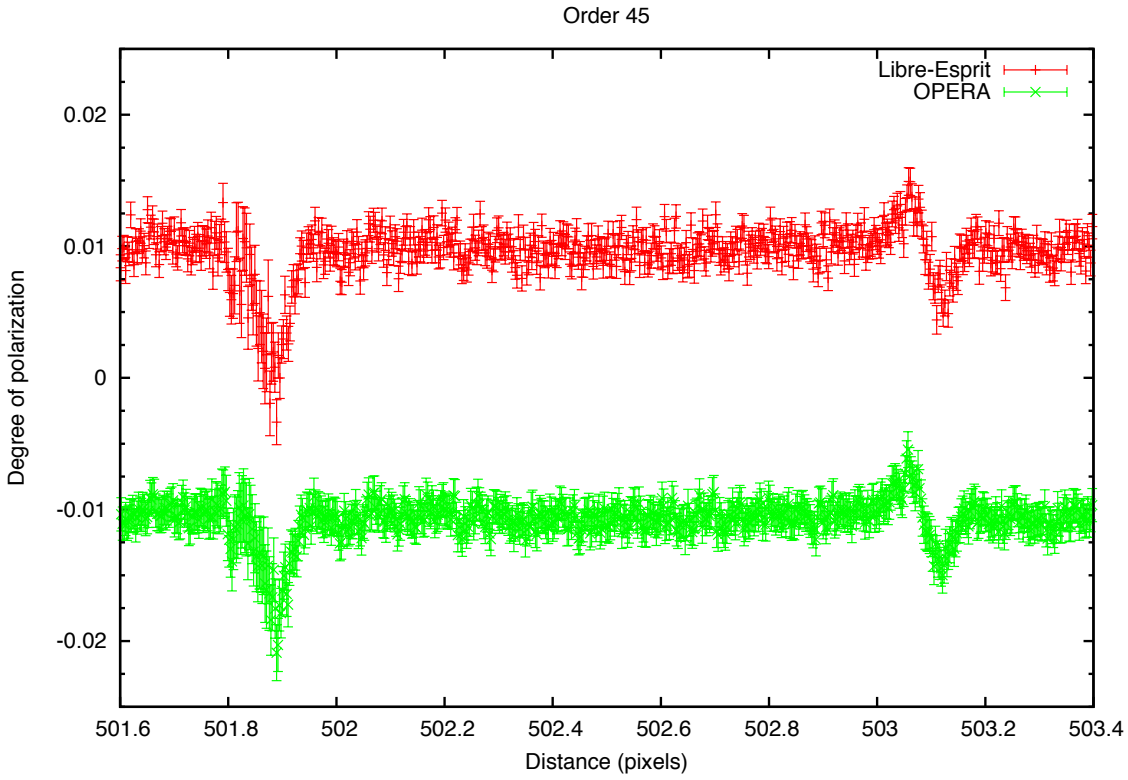


Figure 13 - OPERA polarization measurements (green) compared to Libre-Esprit measurements (red) for a star presenting a detectable polarization feature.

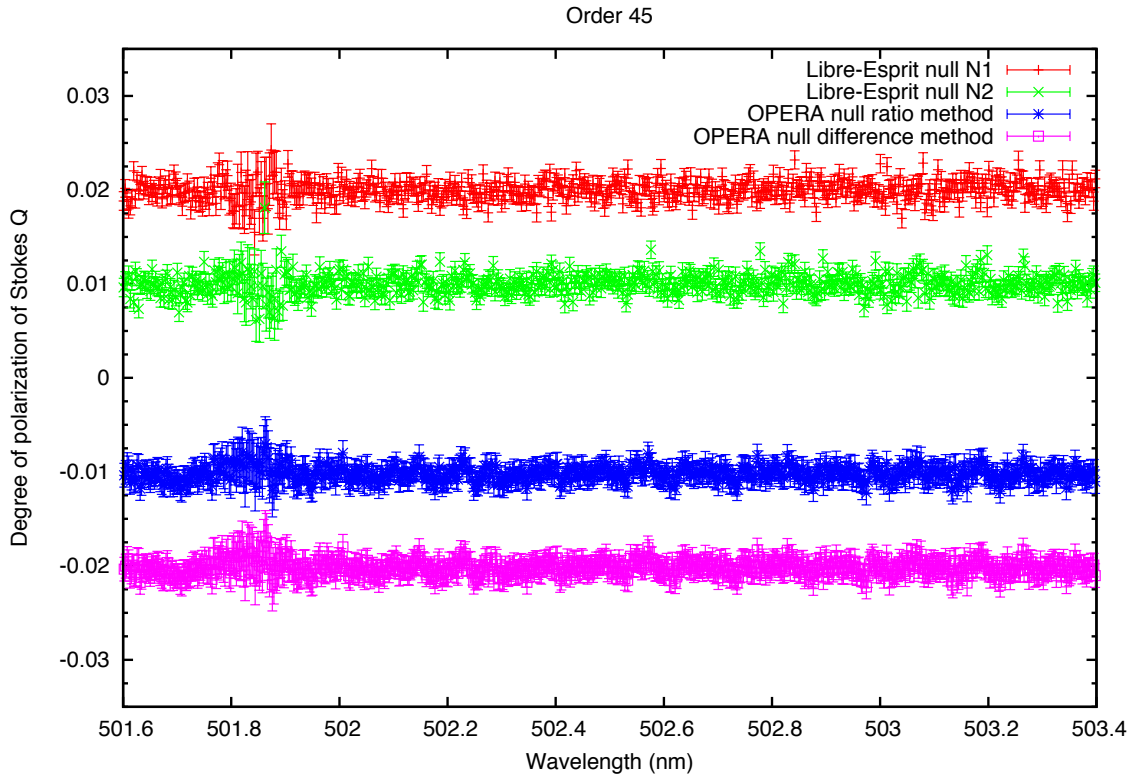


Figure 14 - OPERA NULL polarization spectrum (blue and magenta) compared to Libre-Esprit measurements (red and green) for the same data presented on Figure 13.

Figure 15 and 16 present the dispersion in the degree of polarization in order 45. Since the degree of polarization and the null polarization spectrum have the same levels of noise, the dispersion is measured on the null polarization spectrum. That way, the dispersion is not affected by the polarization features. Furthermore, the edge of the order are noisier due to lower intensities, so the order has been divided into 3 segments for our error analysis. The middle segment presents better precision. Figure 15 shows the PDF for OPERA results. A gaussian fit returns a sigma of about 0.0007%. Figure 16 presents the dispersion in the middle segment for OPERA and Libre-Esprit for comparison. The gaussian fit to the PDF shows

that data reduced with Libre-Esprit has a sigma dispersion of about 0.0011%, which is slightly higher than OPERA.

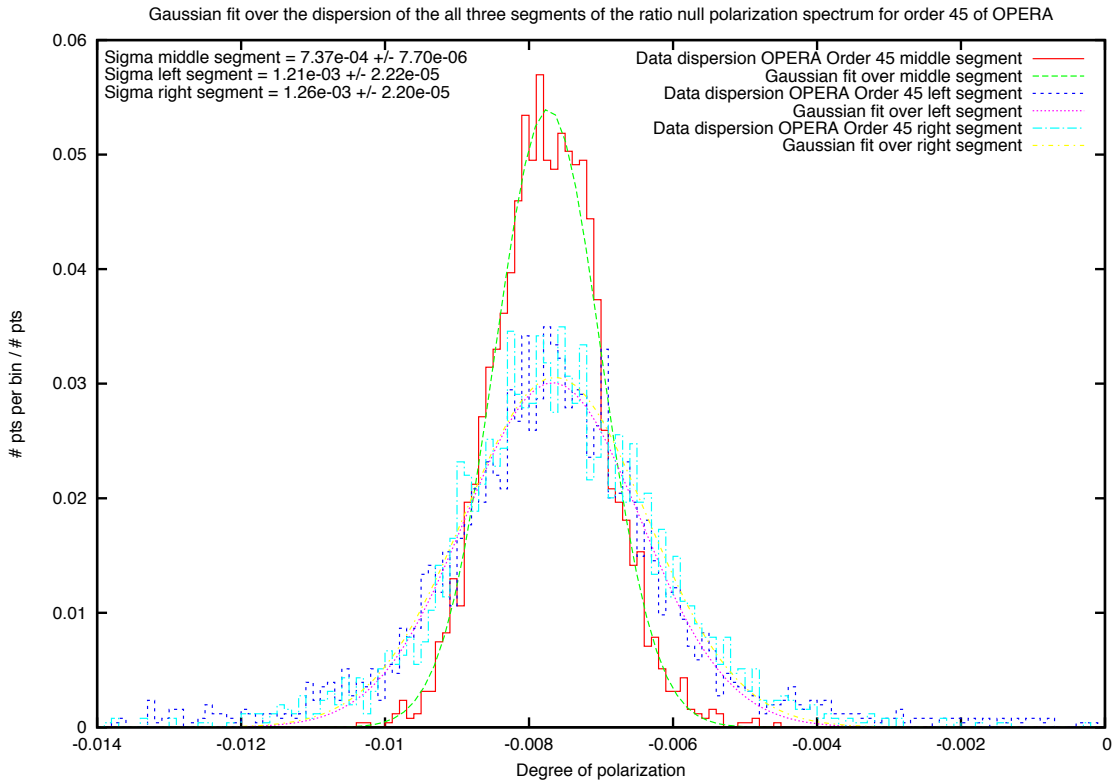


Figure 15 - Dispersion of the degree of polarization of OPERA order 45. The order was divided in 3 segments. The middle segment (in red and green) presents less noise than the edges of the order.

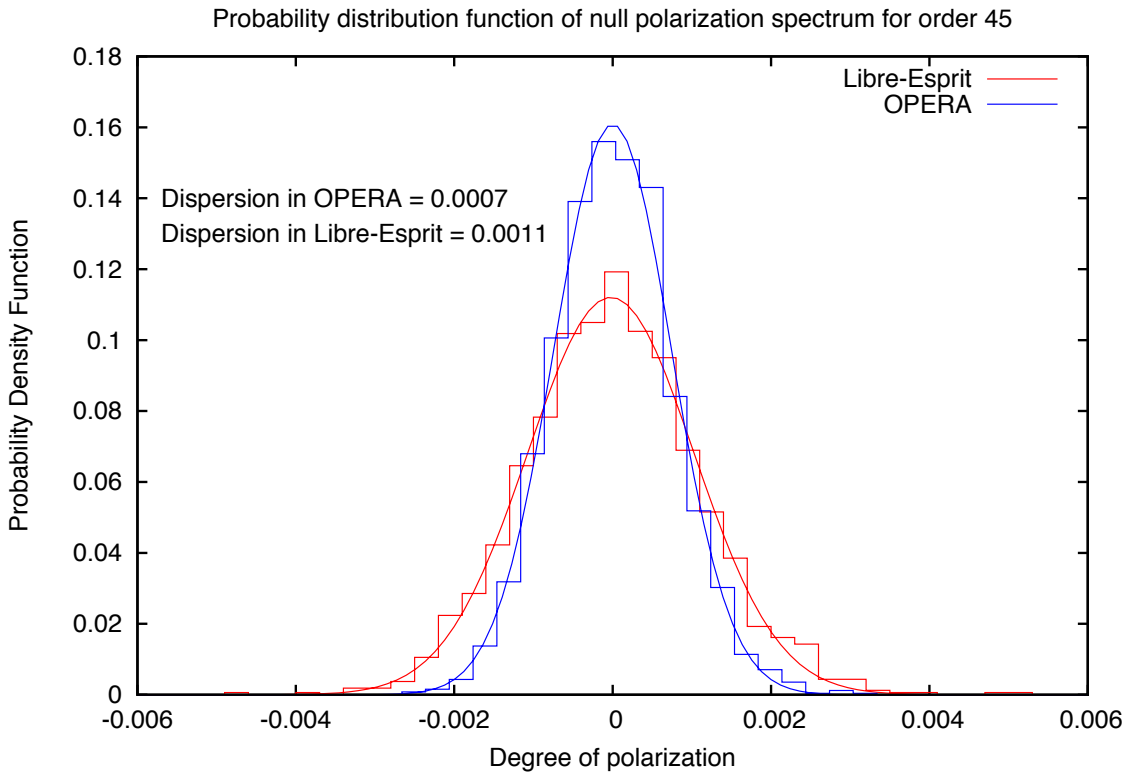


Figure 16 - Probability density function for the degree of polarization of middle segment of order 45 for null polarization spectrum data reduced by OPERA (red line) and Libre-Esprit (blue line).

The aperture calibration module meets the requirement of modeling an extraction aperture that accounts for the tilt in ESPaDOnS pseudo-slit. The module also meets the requirement of using an over-sampled aperture width per spectral bin, which in the case of Libre-Esprit is chosen as 0.6 pixel.

The main requirement for the wavelength calibration module was that it should provide an overall precision better than 150 m/s. Our results with OPERA shows that we are close to this precision for most of the orders, with some orders presenting degraded results due to high levels of saturation. Another requirement that is accomplished with this module is the cal-

culation of the instrument spectral resolution, which presents to be in good agreement with nominal instrument values.

The normalization module main requirement was that it should be able to detect and normalize the continuum of an object stellar spectrum that is not dominated by either broad band absorption lines or emission lines. As it was presented in our results, the algorithm is robust even for a relatively large number of absorption features, which is expected to be found in typical stellar objects.

A prototype for OPERA that implements the polarimetric module has also been tested and as shown in Figure 11 presents a polarimetric precision comparable to Libre-Esprit reduction. The requirement for this module was that it should perform as good or better than Libre-Esprit.

9. Conclusion

This document described the major classes, data structures and algorithms used in calibration and reduction ini OPERA, specifically, geometry calibration, aperture calibration, wavelength calibration, normalization, flux calibration, polarimetry, telluric wavelength calibration, heliocentric calibration. and extraction. The OPERA classes and libraries define high level operations that the scientist may use to perform processing. At the same time, the implementation is a very efficient compiled language. The efficiency of the OPERA implementation has two benefits. Firstly, large quantities of data may be reduced, as required by a production environment. Secondly, the scientist is free to implement advanced algorithms which otherwise may be considered too slow to be of practical use. So, OPERA presents the best of both worlds: the level of abstraction of a high level object-oriented language and the efficiency of a compiled language.

Operationally, the pipeline efficiently performs the required calibration steps at both a high level if desired, or at the level of individual steps. OPERA performs extraction and polarimetry as required for the ESPaDOnS instrument.The OPERA prototype has shown to be fast and robust enough to meet the demands of a production environment, reducing the data of many Principal Investigators well within the time constraints of the Canada France Hawaii Telescope environment.The design of OPERA is firmly based on clear scientific and operational requirements. From its inception, it was envisioned as open source software, that would benefit from the contributions of experts in numerous institutions. It is extensible by design, clearly separating instrument parameters and site configuration from the modules and libraries themselves.

The design addresses the stringent requirements of a production pipeline that must deliver products for many users on a daily basis. It is robust, fast, failure-resilient, scalable from single user to multiple simultaneous server operation. The design is highly modularized and new modules may easily be added to extend the pipeline capabilities. It is designed to be portable to GNU/Linux and MacOSX platforms.

Appendix A - Product Formats

This appendix shows samples of calibration products. The data in each product is self identifying as given by the marker on the first line. The data represents a serialization of the content of the corresponding class. The OPERA class `operaSpectralOrderVector` contains serialization and de-serialization methods for all types of OPERA calibration data.

1. Aperture

```
#!aper
######################################################################
# Extraction Aperture format is:
#
# <number of orders>
# <order number> <number of beams> <measured tilt> <tilt error>
# <order number> <leftBackgroundIndex> <xsampling> <ysampling> <lb height> <lb width> <lb slope> <lb xcenter>
<lb ycenter> <lb fluxFraction>
# <order number> <rightBackgroundIndex> <xsampling> <ysampling>  <rb height> <rb width> <rb slope> <rb xcenter>
<rb ycenter> <rb fluxFraction>
# <order number> <beam> <xsampling> <ysampling>  <beam height> <beam width> <beam slope> <beam xcenter> <beam
ycenter>  <beam fluxFraction>
# each beam here...
#
# Note that leftBackgroundIndex = 0.
# Note that rightBackgroundIndex = 1.
# Note that <beam> is zero-based.
#
######################################################################
44
21 2 -1.9583 0.4906
21 0 5 5 0.600000 2.000000 -0.041880 -14.987739 0.627692 0.005018
21 1 5 5 0.600000 2.000000 -0.041880 14.987739 -0.627692 0.114560
21 0 5 5 0.600000 14.000000 -0.041880 -6.993869 0.292906 0.087266
21 1 5 5 0.600000 14.000000 -0.041880 6.993869 -0.292906 0.131925
22 2 -2.1322 0.5054
22 0 5 5 0.600000 2.000000 -0.041880 -14.987739 0.627692 0.001255
22 1 5 5 0.600000 2.000000 -0.041880 14.987739 -0.627692 -0.000012
22 0 5 5 0.600000 14.000000 -0.041880 -6.993869 0.292906 0.080094
22 1 5 5 0.600000 14.000000 -0.041880 6.993869 -0.292906 0.101891
23 2 -2.3402 0.0000
...
```

2. Wavelength

```
#!wave
######################################################################
# Wavelength Calibration, the format is:
# <number of orders>
# <order number> <number of coefficients> <polynomial coefficients>
# ...
#
######################################################################
44
22 4
979.149 0.010321 -3.02306e-07 -1.57731e-12
23 4
944.326 0.00987524 -2.90112e-07 -1.262e-12
24 4
902.525 0.00946668 -2.78391e-07 -1.16577e-12
25 4
851.884 0.00909062 -2.67426e-07 -1.12601e-12
...
```

3. Instrument Profile

```
#!prof
########################################################################
# Instrument Profile Calibration the format is::
# <number of orders> <number of columns i> <number of rows j> <xsize> <xsampling>
<ysize> <ysampling>
# <order number> <i> <j> <number of coefficients> <ndatapoints> <polynomial coeffi-
cients> <chisqr>
# ...
#
########################################################################
44 150 30 30 5 6 5
22 0 0 1 0 0 0
22 1 0 1 0 -2.49179e-07 4.03458e-09
22 2 0 1 0 -5.2636e-06 1.50845e-08
22 3 0 1 0 -7.10612e-06 2.99461e-08
```

4. Order Spacing Polynomial

```
#!ordp
########################################################################
# Order Spacing Polynomial, the format is:
# <number of orders> <minorder> <maxorder> <chisqr>
# <number of coefficients> <polynomial coefficient> <polynomial coefficienterror> ...
#
########################################################################
44 18 61 0.0965637
3 2.5988546371e+01 9.4846338034e-01 2.2815797478e-02 2.2439078894e-03 3.2395435028e-06
1.0811994571e-06
```

5. Geometry Polynomial

```
#!geom
########################################################################
# Geometry Calibration, the format is:
# <number of orders>
# <order number> <number of coefficients> <ndatapoints> <polynomial coefficient>
<polynomial coefficienterror>... <chisqr> <YBinning> <miny><maxy>
# ...
#
########################################################################
44
18 3 4597 7.0399336830e+01 0.0000000000e+00 -1.0914863902e-01 0.0000000000e+00
2.2718632850e-05 0.0000000000e+00 0.1245 20 3.0000 4600.0000
19 4 4597 9.8833277181e+01 0.0000000000e+00 -1.1280984522e-01 0.0000000000e+00
2.4035242364e-05 0.0000000000e+00 -1.2165472230e-10 0.0000000000e+00 0.1988 20 3.0000
4600.0000
20 4 4597 1.2746104503e+02 0.0000000000e+00 -1.1400426967e-01 0.0000000000e+00
2.4026208298e-05 0.0000000000e+00 -6.5963536156e-11 0.0000000000e+00 0.2094 20 3.0000
4600.0000
```

6. Gain/Bias/Noise

```
#!gain
######################################################################
# Gain Noise format is:
# <number of amps>
# <amp> <gain> <noise> <gainerror> <bias>
# ...
# Note that <amp> is zero-based.
# Note that gain is in units e/ADU, noise in e and bias in ADU.
#
######################################################################
2
0         1.1963 3.6962 0.0033 388.0000
1         1.1787 3.6194 0.0021 389.0000
```

7. BeamSpectrum

```
#!standardbeamspectrum
######################################################################
# Standard Beam Spectrum format is:
# <number of orders>
# <order number> <nElements> <nBeams> <elementindex> <SpectralElements photoCenterX>
<SpectralElements photoCenterY> <SpectralElements dist> <SpectralElements flux> <Spec-
tralElements flux variance> <XCorrelation>
# <order number> <nElements> <nBeams> <beam> <BeamElements[beam] photoCenterX> <BeamE-
lements[beam] photoCenterY> <BeamElements[beam] flux> <BeamElements[beam] flux vari-
ance>
# ...
#
######################################################################
43
22 7652 2 0 205.51 3.30036 0.302583 233.51 1055.07 0.377375
22 7652 2 0 199.017 3.61902 118.838 529.333
22 7652 2 1 212.002 2.9817 114.672 525.733
22 7652 2 1 205.436 3.90108 0.907746 245.19 1075.2 0.364972
22 7652 2 0 198.944 4.21974 113.376 530.881
22 7652 2 1 211.929 3.58243 131.813 544.322
22 7652 2 2 205.363 4.50181 1.51291 237.892 1061.41 0.412503
```

8. SNR

```
#!SNR
######################################################################
# SNR format is:
# <number of orders> <cols>
# <order number> <nElements> <Center SNR> <wavelength> <SNR>
# ...
#
######################################################################
0 4
22 7658 5.7256 0.0000 -0.3263
22 7658 5.7256 0.0000 -0.2456
22 7658 5.7256 0.0000 -0.0879
```

9. The operaFITSProduct

The spectra above are packaged into a FITS image for archival storage.Please see the companion document: OPERA-DistributionProduct for details.

Appendix B - Terms

Below we define the meaning of certain terms that are used consistently throughout this document.

• ESPaDOnS - Echelle Spectro-Polarimetric Device for the Observation of Stars

• OPERA Core Reduction - consist of the main steps to produce reduction products of the CFHT-based pipeline for ESPaDOnS.

• Upena - current ESPaDOnS core reduction pipeline that calls Libre-Esprit software.

• OPERA Analysis And Post Reduction - consists of the reduction steps that are optional and open ended.

• OPERA Reduction Products - output files of the Core Reduction steps. The files are in FITS format and will be distributed to an archiving center.

• OPERA Analysis Products - OPERA Analysis Products are the output files of the optional OPERA Analysis steps.

• Observing Mode - ESPaDOnS at CFHT is offered in three different observing modes: Polarimetry, Star + Sky, Star only.

• Polarimetry (pol) - observing mode that provides spectra of the degree of polarization through any of the three Q, U, or V Stokes parameters, plus the intensity Stokes I. The spectrograph is operated in this mode with two fiber channels, one for each orthogonal polarization state, and three slices produced by the Bowen-Walraven image slicer. The resolution for this mode is about 65,000.

• Star + sky (sp1) - observing mode that provides sky subtracted intensity spectra. The spectrograph is operated with two fiber channels, one for object and another for sky, and three slices produced by the Bowen-Walraven image slicer. The resolution for this mode is about 65,000.

• Star only (sp2) - observing mode that provides intensity spectra. The spectrograph is operated with one fiber channel, and six slices produced by the Bowen-Walraven image slicer. The resolution for this mode is about 80,000.

• Readout Mode - Readout mode refers to the speed at which pixels are read from the detector. The speeds are: Fast, Normal, Slow. The readout mode has an impact on the detector noise and gain.

- EEV1 - backside-illuminated, 2K x 4.5K CCD detector, which was used by ESPaDOnS until 2011. • Olapa - deep-depletion E2V, 2K x 4.5K CCD detector, currently in use by ESPaDOnS since 2011.

Appendix C - References

[1] Donati, J.-F., Semel, M., Carter, B. D., Rees, D. E., and Cameron, A. C., "Spectropolarimetric observations of active stars," MNRAS 291, 658–682 (1997).

[2] Piskunov, N. E. and Valenti, J. A., "New algorithms for reducing cross-dispersed echelle spectra," A&A 385, 1095–1106 (2002).

[3] Horne, K., "An optimal extraction algorithm for ccd spectroscopy," PASP 98, 609–617 (1986).

[4] Marsh, T. R., "The extraction of highly distorted spectra," PASP 101, 1032–1037 (1989).

[5] Lovis, C. and Pepe, F., "A new list of thorium and argon spectral lines in the visible.," A&A 468, 1115–1121(2007).