Open source Pipeline for Espadons Reduction and Analysis

Eder Martioli[a], Doug Teeple[a], Nadine Manset[a], Daniel Devost[a], Kanoa Withington[a], Andre Venne[b], Megan Tannock[c]

[a]Canada France Hawaii Telescope, 65-1238 Mamalahoa Hwy, Hawaii, USA;
[b]Université de Sherbrooke, Québec, Canada;
[c]University of Victoria, British Columbia, Canada.

ABSTRACT

OPERA is a Canada France Hawaii Telescope (CFHT) open source collaborative software project currently under development for an ESPaDOnS echelle spectro-polarimetric image reduction pipeline. OPERA is designed to be fully automated, performing 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 open-ended, permitting users of OPERA to extend the pipeline capabilities. 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. The pipeline is hosted on SourceForge under the name "opera-pipeline".

This article concentrates on the design and software development of OPERA, whereas a forthcoming article to be published in an astronomical journal will present the details of the scientific methods employed by OPERA.

Keywords: CFHT, ESPaDOnS, OPERA, spectropolarimetry, echelle spectroscopy, data reduction

1. INTRODUCTION

The Open-source Pipeline for Espadons Reduction and Analysis (OPERA) is an open-source software reduction pipeline for echelle spectro-polarimetric data. OPERA will be used at CFHT for ESPaDOnS data reduction and replace the existing Upena pipeline based on Libre-Esprit software (Donati et al., 1997[1]).

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.

There have been a number of echelle spectrum pipelines developed in the past (e.g. REDUCE Package[2]). 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 is, by definition and design, extensible software, which is fully configurable and portable to GNU/Linux platforms. The main features of OPERA proposed in the design phase are presented in Section 2 as the scientific, operational, and technical requirements. Many of the design decisions address issues we see in other pipelines that preclude the software from being expanded to any other scientific use of the software. The software design for OPERA is presented in Section 3. Section 4 focusses on the actual reduction process as performed by OPERA to calibrate and extract flux information from

Further author information: (Send correspondence to E.M. or D.T.)
E.M.: E-mail: eder@cfht.hawaii.edu, Telephone: 1 808 885 3167
D.T.: E-mail: teeple@cfht.hawaii.edu, Telephone: +1 808 885 7944

echelle spectral images. OPERA produces calibrated spectra using four different extraction methods: standard extraction with and without background subtraction, Optimal Extraction introduced by Horne, K. (1986),[3] and a new optimal extraction algorithm using a two-dimensional instrument profile. Section 5 summarizes the work presented here and gives some concluding remarks.

Below we define the meaning of certain terms that will be used consistently throughout this article.

- 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.

2. REQUIREMENTS

Since the early phases of the development of OPERA, we established a set of requirements to define the goals of the project. OPERA, as a production pipeline, will be routinely used at CFHT and will serve a vast community of ESPaDOnS users. Each user, however, has a particular requirement based on their science. Therefore, a common sense basic scientific requirement is that this pipeline must produce satisfactory and stable results for the potential science that can be done with ESPaDOnS. Also, OPERA design should not preclude support for an eventual new scientific use of the instrument. Bearing this in mind, together with the current system of operations at CFHT and the available resources, we formulated scientific, technical and operation requirements as presented in detail below.

2.1 SCIENTIFIC REQUIREMENTS

2.1.1 Support for Different Modes of Operation

OPERA-core must be able to take 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 with any of the detector modes (EEV1e, EEV1, Olapa in 1-amp mode, Olapa in 2-amp mode), at any time since ESPaDOnS was made available to the CFHT communities in queue mode.

2.1.2 Minimum Calibration

OPERA-core must be able to reduce any ESPaDOnS science spectro-polarimetric data, meeting all scientific requirements, for all configurations described in Item 1, by making use of the following set of exposures:

- Calibration: 1 bias , 5 flat-field, 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.

2.1.3 OPERA Products Format

OPERA-core must provide reduced data as OPERA-core products in a standard format suitable for analysis by Principal Investigators (PIs): standard FITS and ascii tables. The OPERA-core products for each respective observing mode must contain 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. 1997[1]), and errors.

2.1.4 Removal of Detector Signatures

OPERA-core reduction must detect and correct the following CCD detector effects:

- Bias should be 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 should be identified and masked out.
- Pixel-to-pixel sensitivity variations should be 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 should be corrected if its amplitude is greater than the uncertainty in the flux for combined flat-field calibration exposures.

2.1.5 Removal of the Spectrograph Signatures

OPERA-core reduction must detect and correct the following instrumental signatures specific to ESPaDOnS.

- Order curvatures should be traced for all extracted orders to an RMS precision better than 0.1 pixel.
- Pseudo-slit shape should be measured for all extracted orders to an RMS precision better than 0.1 pixel.
- Pixel-to-wavelength calibration. All orders should be calibrated with an average accuracy better than 150 m/s. The wavelength calibration should use a Th-Ar lamp exposure as comparison and it should also provide 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 must be capable of using an archival solar spectra taken from an exposure of the Moon to perform the flux calibration with accuracy determined by the archival solar spectra. OPERA-core is not required to account for long-term variations in the instrumental effects mentioned above.

2.1.6 Extraction

OPERA-core must be able to extract the two-dimensional data and reduce them into one-dimensional spectra from at least 40 spectral orders. OPERA-core must offer the optimal extraction algorithm by Horne, K. (1986)[3] and further revised by Marsh, T.R. (1989).[4] A different approach for the extraction may be used as an option, but it must be proven to be reliable and more efficient than the optimal algorithm.

2.1.7 Errors

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

2.1.8 Continuum Normalization

OPERA-core must normalize the spectra by the continuum. The OPERA-core normalization algorithm should be 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 are not required to 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.

2.1.9 Signal-to-Noise Ratio

OPERA-core must calculate and retain as a byproduct the signal-to-noise ratio (SNR) for all orders.

2.1.10 Spectral Resolution

OPERA-core must calculate and retain as a byproduct the spectral resolution for all orders.

2.2 OPERATIONAL REQUIREMENTS

2.2.1 Platforms

The OPERA Core Reduction modules OPERA must execute on existing CFHT server hardware. As such it must execute on 32 or 64 bit hardware running a GNU/Linux.

2.2.2 Reduction Speed

OPERA Core Reduction modules should process 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 We currently advertise that ESPaDOnS PIs can get reduced data by noon HST the day following a night of observations.

2.2.3 Instruments

OPERA Core Reduction must reduce data taken from the ESPaDOnS EEV1 and OLAPA devices in either one or two amplifier device modes. OPERA must handle device mode changes within a given night. OPERA must associate calibrations with science data taken in the same instrument device mode. OPERA provides an open source framework that should not preclude use by other instruments. All parameterization should be localized to header files or definition files. Interfaces to data should be 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 should be configurable so that reduction modules can be added or inserted reasonably easily.

2.2.4 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.

2.2.5 Reliability and Availability

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

2.2.6 Autonomous Operation

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

2.3 TECHNICAL REQUIREMENTS

2.4 Programming and System Requirements

The OPERA Core Reduction modules should compile and run on any platform with a recent version of the GNU toolchain (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 must be open source libraries and freely available. OPERA modules should not require that users purchase licensed software in order to use OPERA.

2.5 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 must not preclude parallel module execution and may support module execution on multiple machines. The harness should be command-line driven. It should be abort-able, but 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.

2.6 Parameterization

OPERA pipeline execution should be parameterized, based on instrument, detector and other characteristics. The parameters may be stored in a data table or database, but use of a database is not mandatory. Access to parameters should be through a parameter access layer provided as a software library.

2.7 Software Modules and Software Libraries

Software modules must take all inputs and the names of all outputs as command line arguments. In addition, the harness should pass, as standard arguments, a location to store temporary byproducts and modules must not arbitrarily store temporaries in a hard-coded directory. Nothing in any module should preclude multiple instances of a module executing simultaneously. Common software libraries should be constructed and should be used by modules where possible.

2.8 External Dependencies

External dependencies should be kept to a minimum. The OPERA Core Reduction modules should not link to proprietary software libraries or data.

3. SOFTWARE ARCHITECTURE

3.1 Harness

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 GNU/Linux utility program “make” was chosen as the harness vehicle, since it supports the important requirements os clean abort-ability (any targets that were only partially completed when a user issues an abort are deleted automatically), restart-ability (since make rules are dependency-driven, when processing restarts after an abort, processing continues to create only those products that have yet to complete). Many pipelines use scripts as drivers for pipeline processing, but scripts lack these important characteristics.

The main entry point to customization lies the root Makefile in the OPERA harness:

```
harness/Makefile
```

This makefile contains the key variable instrument:

```
################################################################
# IMPORTANT to define the instrument here, selects the set of Makefiles to use...
################################################################

instrument      := espadons
```

This selects the set of Makefiles to use in the harness that contain execution rules for a particular instrument. There are two further kingpins to OPERA customization. These are the parameters to modules, and configuration. The parameters are stored in:

```
harness/Makefile.parameters.espadons
```

Parameters are intended to be module-specific processing parameters. In the design of OPERA we separate processing parameters from being hard-coded inside the module source code. The harness reads parameters particular to each module and passes them the module when it executes.

The parameters are different for each module and identified as in the sample below:

```
################################################################
# wavelength / telluric correction parameters
################################################################

wcal_threshold                := 0.5
wcal_sigma                    := 2.0
FourierFilterWidth            := 4.0
thorium_argon_atlas           := thar_MM201006.dat
# -1 means all orders
wcal_ordertoplot              := 52

################################################################
# instrument profile
################################################################
ipDimensions                  := 20 1 1 1
```

Second is the configuration. It contains location-dependent global definitions such as file paths. For example:

```
#################################################################
#
# CFHT-specific paths
#
#################################################################
ifeq ($(observatory),CFHT)

ifeq ($(OSTYPE),darwin)
        queuedir        := /data/espadons/
        outdir          := $(homedir)/opera/
else
        queuedir        := /data/niele/espadons/
        outdir          := /data/uhane5/opera/
endif

DATADIR         := $(queuedir)/$(NIGHT)/
ROOTDIR         := $(homedir)
NIGHT           := $$(basename $(DATADIR))
specdir         := $(outdir)/spectra/$$(basename $(DATADIR))/
reductiondir    := $(outdir)/reductions/$$(basename $(DATADIR))/
configdir       := $(operahomedir)/config/
calibrationdir  := $(outdir)/calibrations/$$(basename $(DATADIR))/
byproductsdir   := $(outdir)/byproducts/$$(basename $(DATADIR))/
processeddir    := $(outdir)/processed/$$(basename $(DATADIR))/
approveddir     := $(outdir)/approved/$$(basename $(DATADIR))/
tmpdir          := /tmp/$$(basename $(DATADIR))/
logdir          := $(outdir)/logs/$$(basename $(DATADIR))/
visualsdir      := $(outdir)/visuals/$$(basename $(DATADIR))/
endif
```

These definitions are given as an example only and will be determined for the particular instrument. These definitions mimic the Upena pipeline setup. Note the extensive use of conditional definitions and the use of Make variables.

An important feature of OPERA is scalability to production level capacity. Scalability by default means execute a single thread on the machine where OPERA is launched. However OPERA scales to execute multiple processes on multiple pieces of hardware simultaneously for production-level performance. Scalability is controlled as in the configurations shown below:

```
######################################################################
#
# CFHT-specific reduction machine definitions. Modify these entries to implement
# various machine use configurations.
#
######################################################################

# this runs just on the local machine, launching 1 process.
MACHINES                := $${HOSTNAME}
LOADAVERAGES            := 1

# this runs just on the local machine, launching 4 simultaneous processes.
#MACHINES               := $${HOSTNAME}
#LOADAVERAGES           := 4

# this runs on a machine called "remoteserver", launching 12 simultaneous processes
# use this to launch from your desktop and do the heavy work on a server.
#MACHINES               := remoteserver
#LOADAVERAGES           := 12

# this shows how to run on multiple machines, give a name and loadaverage for each,
# here we use 3 machines, polena having 8 cpus gets a heavier load...
# MACHINES              :=  hukilau     naio    polena
# LOADAVERAGES          :=  4           4       8
```

The user may uncomment the sample definitions (using site machine names) above to control execution in the site environment.

3.2 Module

Conceptually, a module executes a single, specific processing step in the pipeline, producing a single process step product.

Generally speaking a module is responsible for a single executable processing step. Each module should take all input parameters on the command line and produce a single output. 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.

A module must 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.

A module must not use shared memory. This design decision stems from the requirement to support multiple simultaneous execution instances of the pipeline.

All modules accept these common arguments:

```
--verbose
```

Which means that modules are directed to print lots of information about the processing they are doing. The content of verbosity depends on the particular module.

```
--trace
```

This means to show the actual call to each module as it is being executed. Trace is handy when you are starting to debug modules as you can copy and paste the trace to the command line to debug a particular module. All the configuration and module execution parameters are instantiated.

```
--debug
```

This is used to print out large amounts of debug information from each module. Useful in early stages of development.

```
--plot
```

OPERA modules are capable of generating plots (either PNG or GNUPLOT eps) files in the visuals directory. These are very useful when porting to new instruments or new environments. Module writers are encouraged to generate profuse plots.

```
--noexecute
```

This command instructs OPERA to show the commands that it would execute, in the order that it would execute them, with all configuration and module parameters instantiated. This is handy for those who like the comfort of script-like execution, but the script is not suitable for execution in a production environment.

3.3 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 `NaN`s 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:

1. nondestructive function call
2. inline function call (for speed)
3. 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.

3.4 Major Classes

Classes have been developed to handle images:

- `operaFITSImage` - a container for FITS-format images.
- `operaEspadonsImage` - an extension of `operaFITSImage` that also handles mode, speed, detector, and datasec.
- `operaFITSSubImage` - a container containing only pixels and dimensions (no headers) that contains an arbitrary sub-window of an `operaFITSImage`.

THe following classes, while not strictly required for ESPaDOnS images, have been developed to support instruments in future adaptations of OPERA.

- `operaMultiExtensionDITSImage` - a contained for images stored in the FITS MEF format.
- `operaMultiExtensionsFITSCube` - a contained for the FITS Image type MEF Cube. This is the format chosen for the CFHT WIRCam instrument.
- `operaFITSCube` - a FITS cube that is not in MEF format.

3.4.1 The `operaFITSImage` Class

An `operaFITSImage` is most easily created by definition, from an existing FITS file:

```
operaFITSImage anImage(filename, 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 float. `tfloat` is borrowed from CFITSIO terminology.

You may also create in-memory `operaFITSImages`:

```
operaFITSImage outputImage(filename, anImage.getnaxis1(), anImage.getnaxis2(), tfloat);
```

Here an image is created, which may be populated with data and saved to disk. There are other useful methods such as:

```
outputImage.operaFITSImageSave();
outputImage.operaFITSImageSaveAs("foo.fits");
outputImage.operaFITSImageClose();
```

One particularly useful method is `transpose`(). It transposes row for columns. This is handy since the orders lie along columns in ESPaDOnS images:

```
outputImage.transpose();
```

C++ has a feature called operator overloading. You cannot create new operators but you can define a meaning of existing operators for any given class. OPERA defines operators for each of the major classes that have specific meaning for images. For example, the assignment operator = is overloaded:

```
anImage = 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.

```
anImage = anotherImage; // copy the pixels from anotherImage to anImage
```

Here the pixels are copied from one image to another. The operators +=, -=, *=, /= also have meaning in the context of an image:

```
flatImage -= biasImage;          // remove the bias from the flat

anImage /= flatImage;            // divide anImage by the flat
anImage *= badpixelImage;        // mask anImage with the badpixel mask

anImage += 300.0;                // add a constant bias to anImage
```

The binary operators $+, /, +, -$ also have meaning in the context of an image:

```
anImage = flat1 - flat2; // anImage is the pixel-by-pixel difference between the flats
anImage = anImage * badpixelImage; // mask anImage with the badpixel mask
```

The operators $<, >, <=, >=$ also have meaning:

```
anImage *= anImage > 0.0;        // zero any negative values
```

Here `anImage` > `0.0` creates an in-memory pixel map of 1s and 0s of the dimensionality of anImage, where there is a 1 wherever anImage has a positive pixel values, else 0. This map is then multiplied by anImage, effectively masking all negative pixels in anImage. Take care that this expression:

```
anImage = anImage > 0.0;         // save the mask in anImage
```

is very different, because here anImage takes the value of the mask itself, becoming all 1s and 0s. Note that binary operators create in-memory temporary images that are reclaimed after use automatically. It is also possible to get the value of a pixel by using the [] operator:

```
// print some pixel values from the center of the image
// Note that y is the first dimension
for (unsigned y=2048; y<2056; y++) {
        for (unsigned x=1024; x<1032; x++) {
                cout << anImage[y][x] << ' ';   // output the pixel value
                anImage[y][x] = 0.0;    // zero the pixel value
        }
        cout << endl;
}
```

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 an `ImageIndexVector` type defined that is created by the "`where`" function, which has similar semantics to the where function in IDL:

```
// here we actually store the null-terminated vector of indices into vector
// operaImageVector are one-based image positions
operaImageVector vector = where(flatImage>0.0);
```

An `operaImageVector` is defined as a vector of image indices, where the image is treated as being singly dimensioned. The "`where`" function is useful for creating bins of image pixel values:

```
// here we create a subImage,which is a vector (a bin) of pixel values from the
// FITSImage, that fit the bin constraints the vector of indices is discarded
operaFITSSubImage subImage(anImage, where(anImage >= 400.0 && anImage <= 800.0));
```

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 zero, else 1. The operator || means: for every pixel if the either the corresponding lhs is non-zero or the rhs is non-zero, then 1, else zero. 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 one-dimensional `operaFITSSubimage` can be created from the pixel values of `operaFITSImage` of the length of the `operaImageVector`. The operation is essentially creating a bin of pixel values that satisfy the input constraint - all the pixel values from `anImage` between 400.0 and 800.0 ADU. The unary ! operator is also defined. It has the same meaning as ! in C++, except that it creates a map of every pixel.

```
anImage *= !badpixelImage; // mask anImage with the inverse of badpixel mask
```

Here is an interesting example 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 anImage(filename, tfloat); // create an operaFITSImage
anImage -= biasImage;           // remove the bias from the image
anImage /= flatImage;           // divide the image by the flat
anImage *= badpixelImage;       // mask the image with the badpixel mask
float maxvalue = anImage.max(); // find the maximum pixel value
anImage /= maxvalue;            // normalize the image
```

The C++ compiler produces fast inline executable code. There is a small test program in `opera − 1.0/test` called `operaAsmTest.cpp`. This example illustrates the implementation of certain operators for tutorial purposes. Ordinarily, the scientist using these high-level operators would not need to know the implementation details.

```
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
```

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.

3.4.2 The operaFITSSubImage Class

An operaFITSSubImage creates an in-memory 2 dimensional sub-window of an operaFITSImage. Sub-windows are used extensively in OPERA to speed processing. A "subImage" contains only the pixels and dimension, no header information. It is used primarily for doing image statistics. An operaFITSSubImage is most easily created by definition, from an existing operaFITSImage:

```
operaFITSSubImage aSubImage(anImage, 21, 0, 2000, 4096);
```

Here a sub-window is created from anImage, from pixel x=21, y=0, of size 2000x4096 pixels. All of the operators defined for operaFITSImages also work for operaFITSSubImages:

```
aSubImage = flat1SubImage - flat2SubImage; // subtract the flats
```

The where function also can be used to create sub-sub-windows:

```
// we can also use the [] operator to re-dimension a subImage
operaFITSSubImage *subImage2 = subImage[where((operaFITSImage*)(subImage >= 300.0 && subImage <= 350.0))];
```

Note in this example, that the [] operator is also defined to take an ImageIndexVector. Since all of the other operators have identical meanings for operaFITSSubImages, I will not go in to further detail. The transpose() method is also available for the subImage class.

3.4.3 The operaEspadonsImage Class

An operaEspadonsImage is an extension of operaFITSImage. It is most easily created by definition, from an existing FITS file:

```
operaEspadonsImage *In = new operaEspadonsImage(filename, 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= " << In->getspeed() << ' ' << In->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 << setw(8) << setprecision(4) << (float)(*subImage)[y][x] << ' ';  // output the pixel value
        }
        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 of the image
// we need to set it back in to the espadons image
Out.operaFITSImageSetData(*subImage);
```

3.4.4 The Matrix

The indexing operator [] defined for the `operaFITSImage` and `operaFITSSubImage` classes are 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.

```
// 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:

```
// print some pixel values from the center of the untransposedimage
for (unsigned x=1024; x<1032; x++) {
        for (unsigned y=2048; y<2056; y++) {
                cout << matrix[y][x] << ' ';    // output the pixel value
        }
        cout << endl;
}
```

You can also use the `transpose()` method:

```
// Transpose the image, now orders lie along the x direction
anImage.transpose;
matrix = anImage.getmatrix();           // be sure to get the new matrix
unsigned newmaxx = anImage.getnx();     // be sure to get the maxx
unsigned newmaxy = anImage.getny();     // be sure to get the maxy

void fooInC(float **matrix) {
        for (unsigned y=0; y<newmaxy; y++) {
                for (unsigned x=0; x<newmaxx; x++) {
                        cout << matrix[y][x] << ' ';    // output the pixel value
                }
                cout << endl;
        }
}
```

However the transposed version would be faster than simply switching the for loops if a row pointer was obtained and the x values iterated.

```
// get a column from the transposed image
float *column0 = matrix[0];
for (unsigned x=1024; x<1032; x++) {
        cout << column0[x] << ' ';      // output the pixel value
}
cout << endl;
```

3.4.5 Tiling

Reading small parts of an image rather than the whole image is useful for deep stacking image, where data storage for the number of images to be stacked would exceed the capacity of the computer. The `operaFITSSubImage` class has an interface that reads in a subimage and then has an interface to copy that subimage in to a location in the master image. Here is an example:

```
// open an input file
operaFITSImage *In = new operaFITSImage(image, tfloat, READONLY);
// clone to the output file
operaFITSImage *Out = new operaFITSImage(output, In->getnaxis1(), In->getnaxis2(), tfloat, READWRITE);
Out->operaFITSImageCopyHeader(In);
// create a subImage based on the input pixel subrange
operaFITSSubImage subIm(*In, 1024, 1024, 512, 512);
// copy the subimage into the output at 1024,1024
Out->operaFITSImageSetData(subIm, 1024, 1024);
// write the output file
Out->operaFITSImageSave();
```

3.4.6 The `operaMultiExtensionFITSImage` Class

This class extends the `operaFITSImage` to handle the multi extension FITS image. This type of image is used for cameras that have multiple detectors. The extensions add a dimension for indexing. For example:

```
operaMultiExtensionFITSImage anImage(filename , tfloat);

for (unsigned extension=1; extension<=image.getNExtensions(); extension++) {
  for (unsigned y=2048; y<2056; y++) {
    for (unsigned x=1024; x<1032; x++) {
       cout << anImage[extension][y][x] << ' '; // output the pixel value
       anImage[extension][y][x] = 0.0;  // zero the pixel value
    }
  }
}
cout << endl;
```

3.4.7 The `operaMultiExtensionFITSCube` Class

This class extends the `operaMultiExtensionFITSImage` class to handle multi-extension FITS cubes. Cubes add another dimension for indexing:

```
for (unsigned slice=1; slice<=image.getZDimensions); slice++) {
  for (unsigned extension=1; extension<=image.getNExtensions(); extension++) {
    for (unsigned y=2048; y<2056; y++) {
      for (unsigned x=1024; x<1032; x++) {
         cout << anImage[slice][extension][y][x] << ' '; // output the pixel value
         anImage[slice][extension][y][x] = 0.0;          // zero the pixel value
      }
    }
  }
}
cout << endl;
```

In this example, the image is indexed in four dimensions, the slice of the cube (time), the extension (detector) and the y and x coordinates. Notably, this class adds the `medianCollapse()` method, which collapses a cube into a single slice. The WIRCam infrared instrument at CFHT is an example of an instrument that generates MEF cubes.

3.4.8 The `operaFITSCube` Class

This class extends the `operaFITSImage`. The FITS Cube is not required for ESPaDOnS, but is implemented in OPERA to support future instruments which generate FITS cube data.

3.4.9 The `operaFluxVector` Class

The `operaFluxVector` class encapsulates vectors of flux, variance pairs. It is intended for use in polarimetry. Operators are defined that operate on whole vectors, element-by-element, most importantly also calculating the resultant variances. so that variances may be propagated through the various vector operations. A test case shows an example from polarimetry steps below:

```
const unsigned length = 5;

double i1EFluxes[] = {2000.0, 1450.0, 123.0, 4324.9, 1235.0};
double i1EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i1AFluxes[] = {2500.0, 1050.0, 1230.0, 434.9, 235.0};
double i1AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

double i2EFluxes[] = {2400.0, 1650.0, 103.0, 4424.9, 1035.0};
double i2EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i2AFluxes[] = {2200.0, 1250.0, 1030.0, 494.9, 225.0};
double i2AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

double i3EFluxes[] = {2200.0, 1430.0, 163.0, 4524.9, 1295.0};
double i3EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i3AFluxes[] = {2560.0, 1050.0, 1230.0, 434.9, 235.0};
double i3AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

double i4EFluxes[] = {2300.0, 1420.0, 173.0, 4624.9, 1265.0};
double i4EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i4AFluxes[] = {2520.0, 1550.0, 1238.0, 404.9, 205.0};
double i4AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

/*
 * Populate the E/A vectors
 */
operaFluxVector i1E(i1EFluxes, i1EVariances, length);
operaFluxVector i1A(i1AFluxes, i1AVariances, length);
operaFluxVector i2E(i2EFluxes, i2EVariances, length);
operaFluxVector i2A(i2AFluxes, i2AVariances, length);
operaFluxVector i3E(i3EFluxes, i3EVariances, length);
operaFluxVector i3A(i3AFluxes, i3AVariances, length);
operaFluxVector i4E(i4EFluxes, i4EVariances, length);
operaFluxVector i4A(i4AFluxes, i4AVariances, length);
/*
 * 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;
        /*
         * 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);
        R = Sqrt(Sqrt((r1 * r4) / (r2 * r3)));
        /*
         * Now print out the flux (first) and variance (second)
         */
        printf("r1=");
        for (unsigned i=0; i<length; i++) {
                printf("%.2f %.2f, ", r1[i]->first, r1[i]->second);
        }
        printf("\n");
        /*
         * Finally print out the errors
         */
        printf("R errors=");
        for (unsigned i=0; i<length; i++) {
                printf("%.2f, ", R.geterror(i));
        }
        printf("\n");
```

Note that as the calculations are done, variances are propagated to the final product. The `operaFluxVector` class stores booth flux and variance for each point in the vector. The operators are defined in such a way as to propagate the variances. The error can be obtained at any point using the "`geterror`" function call. The user of the `operaFluxVector` class need not know the details of the `operaFluxVector` class, but just how to use it. For tutorial purposes, to help understand the concept, an example of one definition in the class is given. As an example, the operator = is defined as:

```
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;
        while (n--) { *afluxes++ = *bfluxes++; *avariances++ = *bvariances++; }
        if (b.istemp) delete &b;
                return *this;
};
```

Note that both the flux and variance is retained. Each operator uses the statistical definition of variance propagation relevant to the particular operator. This example shows the definition of the operator *, meaning multiply two flux vectors:

```
/*!
 * \brief operator *
 * \brief multiply operator.
 * \param b - operaFluxVector*
 * \note usage:
                operaFluxVector a = operaFluxVector b * operaFluxVector c;
                multiplies the fluxes and error values  b * c and assigns to a
 * \return operaFluxVector&
 */
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, true);
                tfluxes = (double *)t->fluxes;
                tvariances = (double *)t->variances;
        }
        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;
};
```

The relevant line is the "while" statement, where you can see the formula used to propagate variance through multiplication. Each `operaFluxVector` operator is a binary operator operating on a left hand and a right hand side. Complex expressions are built up pairwise according to C++ operator precedence rules. "`geterror`" is defined as:

```
double geterror(unsigned index) { return sqrt(variances[index]); };
```

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 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.

4.1.1 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/average 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 χ^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 at 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`.

4.2 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.

4.2.1 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), Optimal Extraction[3, 4] , and Opera Optimal Extraction. The details for each of these methods is out of the scope of this document. The OPERA team is preparing a science paper on OPERA where the implementation and performance results for these methods will be discussed in greater detail.

- `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.

- `operaTelluricCorrection`. 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.

- `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. CONCLUSIONS

In this paper we presented the CFHT OPERA project, an open-source software project for reduction of spectro-polarimetric ESPaDOnS data. OPERA is currently in the development/prototyping phase.

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.

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.

Many features presented in this paper have been prototyped in a preliminary opera-1.0 release. OPERA is hosted on Source Forge under the name “opera-pipeline”, where one can obtain the latest distribution of opera-1.0.

ACKNOWLEDGMENTS

We would like to acknowledge the many reviewers who participated in the OPERA reviews and provided us with invaluable insights and contributions. The reviewers are mainly CFHT staff, scientists of the Gemini Observatory, and astronomers from France, Canada, Taiwan, Brazil, and University of Hawaii. We would also like to note that this work would not have been possible without the support and incentive of our former director Dr. Christian Veillet, who also gave important contributions in the reviews. In addition, we would like to thank Dr. J.-F. Donati, who developed the current ESPaDOnS pipeline running at CFHT and who has provided us with valid inputs for OPERA. Finally, we would like to acknowledge the members of the CFHT Scientific Advisory Committee, for their continued support in the development of OPERA.

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).