EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

HDRL Pipeline Developer Manual

ESO-299492

Issue 1.1.0

Date April 2018

Prepared:	ESO HDRL Team	April 2018	
	Name	Date	Signature
Approved:	P.Ballester		
	Name	Date	Signature
Released:	M. Peron		
	Name	Date	Signature

Change record

Issue/Rev.	Date	Section/Parag. affected	Reason/Initiation/Documents/Remarks
0.0.1	26/04/2013	All	Overscan Computation & Correction
0.0.2	09/08/2013	All	Master Bias & Master Dark
0.0.3	29/11/2013	All	First Release
0.1.0	17/09/2014	4.5	Bad Pixel Detection
0.1.5	15/12/2014	All	Clipping Algorithm: IQR $\rightarrow$ MAD, Library Integration
0.2.0	01/05/2015	4.4, 4.6	Master Flatfield, Strehl Ratio
0.3.0b1	26/11/2015	4.4, 4.7	Master Flatfield, Master Fringe
1.0.0	01/04/2017	4.8	Object catalogue generation
1.1.0	01/04/2018	4.6, 4.9, 4.10 4.11 and H, 4.12	Strehl Ratio, Spectral Efficiency, Spectral Response, Differential Atmospheric Refraction, Effective air mass

Contents

1 Introduction **13**

1.1 Motivation and Purpose . . . 14

1.2 Stakeholders . . . 14

1.3 Acknowledgements . . . 14

1.4 Scope and references . . . 14

2 Software Infrastructure **16**

2.1 SVN usage and library integration . . . 17

2.2 Releases . . . 17

2.2.1 Release version 1.1.0 . . . 18

2.2.2 Release version 1.0.0 . . . 18

2.2.3 Release version 0.3.0b1 . . . 19

2.2.4 Release version 0.2.0 . . . 19

2.2.5 Release version 0.1.5 . . . 19

2.2.6 Release version 0.1.0 . . . 19

2.2.7 Release version 0.0.3 . . . 19

2.3 Dependencies . . . 20

3 Core Objects **21**

3.1 Images . . . 22

3.1.1 Example usage . . . 22

3.2 Image Lists . . . 23

3.2.1 Views . . . 23

3.2.1.1 Example . . . 23

3.2.2 Collapse Operations . . . 24
3.3 Spectrum . . . 26
3.3.1 Resampling . . . 27
3.3.2 Example usage . . . 27

4 Functionalities **28**
4.1 Overscan . . . 29
4.1.1 Overscan Computation . . . 29
4.1.1.1 Inputs . . . 29
4.1.1.2 Outputs . . . 30
4.1.1.3 Algorithm . . . 31
4.1.2 Overscan Correction . . . 33
4.1.2.1 Inputs . . . 33
4.1.2.2 Outputs . . . 33
4.1.2.3 Algorithm . . . 34
4.1.3 Examples . . . 34
4.1.3.1 Overscan Computation . . . 34
4.1.3.2 Overscan Correction . . . 36
4.1.4 Overscan Parameters in a Recipe . . . 36
4.2 Bias . . . 38
4.2.1 Algorithm - short description . . . 38
4.2.2 Functions . . . 38
4.2.3 Inputs . . . 38
4.2.4 Outputs . . . 39
4.2.5 Examples . . . 39
4.2.6 Master Bias Parameters in a Recipe . . . 39
4.3 Dark . . . 41
4.3.1 Algorithm - short description . . . 41
4.3.2 Functions . . . 41
4.3.3 Inputs . . . 41
4.3.4 Outputs . . . 41

4.3.5 Examples . . . 41
4.3.6 Master Dark Parameters in a Recipe . . . 42
4.4 Flat . . . 43
4.4.1 High frequency master flatfield . . . 43
4.4.1.1 Algorithm - short description . . . 43
4.4.1.2 Functions . . . 43
4.4.1.3 Inputs . . . 44
4.4.1.4 Outputs . . . 44
4.4.2 Low frequency master flatfield . . . 45
4.4.2.1 Algorithm - short description . . . 45
4.4.2.2 Functions . . . 45
4.4.2.3 Inputs . . . 45
4.4.2.4 Outputs . . . 45
4.4.3 Examples . . . 45
4.4.4 Master Flat Parameters in a Recipe . . . 46
4.5 Bad-pixel detection . . . 48
4.5.1 Bad-pixel detection on a single image . . . 48
4.5.1.1 Algorithm - short description . . . 48
4.5.1.2 Functions . . . 48
4.5.1.3 Inputs . . . 48
4.5.1.4 Outputs . . . 49
4.5.2 Bad-pixel detection on a stack of identical images . . . 49
4.5.2.1 Algorithm - short description . . . 49
4.5.2.2 Functions . . . 50
4.5.2.3 Inputs . . . 50
4.5.2.4 Outputs . . . 50
4.5.3 Bad-pixel detection on a sequence of images . . . 50
4.5.3.1 Algorithm - short description . . . 50
4.5.3.2 Functions . . . 51
4.5.3.3 Inputs . . . 51

4.5.3.4 Outputs . . . 51
4.5.4 Cosmic Ray Hits detection . . . 52
4.5.4.1 Algorithm - short description . . . 52
4.5.4.2 Functions . . . 52
4.5.4.3 Inputs . . . 52
4.5.4.4 Outputs . . . 52
4.5.5 Examples . . . 52
4.5.6 Bad-Pixel Parameters in a Recipe . . . 53
4.5.6.1 Bad-pixel detection on a single image . . . 53
4.5.6.2 Bad-pixel detection on a set of images . . . 53
4.5.6.3 Cosmic Ray Hits detection . . . 53
4.6 Strehl . . . 55
4.6.1 Algorithm - short description . . . 55
4.6.2 Functions . . . 55
4.6.3 Inputs . . . 55
4.6.4 Outputs . . . 56
4.6.5 Examples . . . 56
4.6.6 Strehl Parameters in a Recipe . . . 57
4.7 Fringing . . . 58
4.7.1 Master fringe computation . . . 58
4.7.1.1 Algorithm - short description . . . 58
4.7.1.2 Functions . . . 60
4.7.1.3 Inputs . . . 60
4.7.1.4 Outputs . . . 61
4.7.2 Master fringe correction . . . 61
4.7.2.1 Algorithm - short description . . . 61
4.7.2.2 Functions . . . 61
4.7.2.3 Inputs . . . 62
4.7.2.4 Outputs . . . 62
4.8 Object catalogue generation . . . 63

4.8.1 Algorithm - short description . . . 63
4.8.2 Functions . . . 64
4.8.3 Inputs . . . 64
4.8.4 Outputs . . . 65
4.8.5 Examples . . . 66
4.8.6 Catalogue Parameters in a Recipe . . . 67
4.9 Efficiency . . . 69
4.9.1 Algorithm - short description . . . 69
4.9.2 Functions . . . 70
4.9.3 Inputs . . . 70
4.9.4 Outputs . . . 70
4.9.5 Examples . . . 70
4.9.6 Notes . . . 71
4.10 Response . . . 72
4.10.1 Algorithm - short description . . . 72
4.10.1.1 Telluric Correction . . . 72
4.10.1.2 Response Calculation . . . 73
4.10.2 Functions . . . 74
4.10.3 Inputs . . . 74
4.10.4 Outputs . . . 75
4.10.5 Examples . . . 75
4.10.6 Notes . . . 76
4.11 Differential Atmospheric Refraction (DAR) . . . 77
4.11.1 Algorithm - short description . . . 77
4.11.1.1 Owens saturation pressure . . . 77
4.11.1.2 Filippenko refractive index . . . 78
4.11.2 Function . . . 78
4.11.3 Inputs . . . 79
4.11.4 Outputs . . . 80
4.11.5 Example . . . 80

4.11.6 DAR Parameters in a Recipe . . . 80
4.12 Effective air mass . . . 82
4.12.1 Algorithm - short description . . . 82
4.12.1.1 Zenith distance . . . 82
4.12.1.2 Hardie approximation (1962) . . . 83
4.12.1.3 Young & Irvine approximation (1967) . . . 83
4.12.1.4 Young approximation (1994) . . . 83
4.12.2 Functions . . . 83
4.12.3 Outputs . . . 84
4.12.4 Example . . . 84

5 Cookbook **86**
5.1 Handling large datasets . . . 86
5.1.1 Examples . . . 87

A Abbreviations and acronyms **88**

B Statistical Estimators **89**
B.1 Arithmetic mean - Arithmetic weighted mean . . . 90
B.2 Median . . . 91
B.3 Kappa-sigma clipping . . . 93

C Bad Pixel Determination **94**
C.1 Detailed Description of the various Algorithms . . . 95
C.1.1 Bad-pixel detection on a single image . . . 95
C.1.2 Bad-pixel detection on a stack of images . . . 95
C.1.3 Bad-pixel detection on a sequence of images . . . 98
C.1.4 Cosmic Ray Detection . . . 100
C.1.5 Testing the Cosmic Ray Detection . . . 105

D Masterfringe Computation and Removal **109**
D.1 Detailed Description of the de-fringing Algorithm . . . 110

E Object Catalogue Generation **111**

E.1 Detailed Description of the Object Catalogue Generation Algorithm . . . 112

E.2 Detailed Description of the Object Catalogue Table . . . 113

F Efficiency Calculation **114**

F.1 Detailed Description of the Efficiency Computation . . . 115

G Response Calculation **116**

G.1 Detailed Description of the Response Computation . . . 117

H Differential Atmospheric Refraction **118**

H.1 Introduction . . . 119

H.2 Testing HDRL DAR . . . 120

H.3 Testing the *hdrl_dar* Differential Atmospheric Refraction Routines with SINFONI Data . . . 121

H.3.1 I. Method and Qualitative Results . . . 121

H.3.2 II. Quantitative Results . . . 122

H.3.3 The Strehl Ratios in *hdrl_dar*-Corrected Images . . . 125

Chapter 1

Introduction

1.1 Motivation and Purpose

The ESO Pipeline System Group (PPS) is developing and maintaining a large variety of data reduction pipelines for the different instruments. In order to decrease the implementation and maintenance efforts of the single pipelines the *High Level Data Reduction Library* projects was launched.

The *High Level Data Reduction Library* project aims to identify and concentrate cross-pipeline algorithms in a single place, verify and homogenize algorithms, implement error propagation and extensively test the different functionality. Moreover, this should not only accelerate the development of new pipelines, but all pipelines would instantaneously benefit from bug-fixes and improvements in *High Level Data Reduction Library* .

The various consortia are also encouraged to propose new functionality/algorithms for the library - the PPS and SDP department will then evaluate the algorithm and decide if they should be part of the *High Level Data Reduction Library* .

The pipeline developer but also the consortia are highly encouraged to use the new library - if for some reason they want to implement another solution to an already existing algorithm they should inform PPS.

1.2 Stakeholders

The following actors may interact with the *High Level Data Reduction Library* project:

- SciOps
- SDP
- USD
- QC
- PPS
- Consortia

1.3 Acknowledgements

1.4 Scope and references

The modules implemented in the *High Level Data Reduction Library* are high-level functionalities that are intended to be used by several VLT/VLTI instrument pipelines.

The *Common Pipeline Library* plays exactly this role for low-level functionalities.

[1] Anirban DasGupta. *Asymptotic Theory of Statistics and Probability*. Springer, New York, 2008. 91

[2] M. J. Irwin. Automatic analysis of crowded fields. *Monthly Notices of the Royal Astronomical Society*, 214:575–604, June 1985. 18, 63

[3] J. S. Maritz and R. G. Jarrett. A note on estimating the variance of the sample median. *Journ. Am. Stat. Assoc.*, 73:194–196, mar 1978. 91

[4] Paul R. Rider. Variance of the median of small samples from several special populations. *Journ. Am. Stat. Assoc.*, 55:148–150, mar 1960. 91, 92

[5] P. G. van Dokkum. Cosmic-ray rejection by laplacian edge detection. *PASP*, 113:1420–1427, nov 2001. 100

Chapter 2

Software Infrastructure

2.1 SVN usage and library integration

Th *High Level Data Reduction Library* library should be included into the pipeline library during the svn check-out by using the svn::external concept. The external should point to a given library release in the tags and not to the trunk, i.e. to a subdirectory of `http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl` (see also sect. 2.2). In order to build the library one has to add the *hdrl.m4* from the common *m4macros* repository (`Pipelines/common/m4macros` in SVN) to the pipeline sources. Then one has to modify the *configure.ac* and the top-level *Makefile.am* as follows:

- *configure.ac*:

```
# mark hdrl folder as containing a configurable package
AC_CONFIG_SUBDIRS([hdrl])

# check and define all required variables to build and
# link hdrl external located in the hdrl folder
HDRL_CHECK([hdrl])
```

- *Makefile.am*:

```
SUBDIRS = hdrl ...
```

Moreover, the variable `$(HDRL_LIBS)` must also be added to the link variable, `$(HDRL_LDFLAGS)` to the linker flag variable, and `$(HDRL_INCLUDES)` to the `AM_CPPFLAGS` variable in the *Makefile.am* of any folder making use of HDRL. As HDRL is currently a static library it also needs has to be added as a dependency of objects using it so these are relinked when HDRL changes. For example:

```
hdrldemo_bias_la_LDFLAGS = $(HDRL_LDFLAGS) ...
hdrldemo_bias_la_LIBADD = $(HDRL_LIBS) ...
hdrldemo_bias_la_DEPENDENCY = $(LIBHDRL) ...
```

In the source files, the only include needed is:

```
#include <hdrl.h>
```

2.2 Releases

There is no fixed release cycle for the *High Level Data Reduction Library* library (as e.g. for CPL), but new releases are feature-driven, i.e. if there are new functionality/algorithms available and carefully tested a new release will be announced and the pipeline developer can change the svn::external to this new release. This has the advantage that the pipeline developer has more freedom to decide when to update the pipeline. On the other hand it allows the developer to incorporate the new *High Level Data Reduction Library* release to his pipeline on a very short timescale and prepare a new Paranal/public release. This is only possible as the library is not installed in the Data Flow System environment in Paranal (like CPL) but delivered inside each pipeline.

2.2.1 Release version 1.1.0

The *High Level Data Reduction Library* release version 1.1.0 can be included from
http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-1.1.0

In this release we added five new algorithms:

- Computation of the Strehl ratio. The Strehl ratio is defined as the ratio of the peak image intensity from a point source compared to the maximum attainable intensity using an ideal optical system limited only by diffraction over the telescope aperture. The Strehl ratio is very frequently used to perform the quality control of the scientific data obtained with the AO assisted instrumentation.
- Computation of the *spectral efficiency* as a function of wavelength: The efficiency is used to monitor the system performance and health. It is calculated from observing flux standard stars (in photometric conditions). Then, the observed 1D spectrum is compared with the reference spectrum, as it would be observed outside the Earth's atmosphere. The reference spectrum is provided by the user, usually via a catalog of standard stars. See section 4.9 for detailed information.
- Computation of the *spectral response* as a function of wavelength: The algorithm is divided in two parts: *Telluric correction* and *Response calculation*. In the provided implementation the *Telluric correction* is optional and can be disabled by the user. See section 4.10 for detailed information.
- Computation of the *Differential Atmospheric Refraction* as a function on wavelength. The differential atmospheric refraction is calculated according to the algorithm from Filippenko (1982, PASP, 94, 715) using the Owens formula which converts relative humidity in water vapor pressure. See section 4.11 for detailed information.
- Computation of the *effective air mass* of an observation. See section 4.12 for detailed information.

2.2.2 Release version 1.0.0

The *High Level Data Reduction Library* release version 1.0.0 can be included from
http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-1.0.0

In order to provide astrometric and photometric calibration information, the *High Level Data Reduction Library* implements in this release a functionality to generate a catalogue of detected objects (i.e. stars, galaxies).

A high-level summary of the implemented data reduction sequence is:

- estimate the local sky background over the image and track any variations at adequate resolution to eventually remove them,
- detect objects/blends of objects and keep a list of pixels belonging to each blend for further analysis (see [2] for details)
- parametrise the detected objects, i.e. perform astrometry, photometry and a shape analysis.

For details see section 4.8.

2.2.3 Release version 0.3.0b1

The *High Level Data Reduction Library* release version 0.3.0b1 can be included from

http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-0.3.0b1

In this release we added an algorithm to do fringe correction. In a first step the algorithm creates a master-fringe image using a Gaussian mixture model. A properly scaled version of the master-fringe image is then used to remove the fringes from the single images (see section 4.7).

2.2.4 Release version 0.2.0

The *High Level Data Reduction Library* release version 0.2.0 can be included from

http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-0.2.0

In this release we added two algorithms to derive a master flatfield (see section 4.4) and one algorithm to compute the Strehl ratio (see section 4.6)

2.2.5 Release version 0.1.5

The *High Level Data Reduction Library* release version 0.1.5 can be included from

http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-0.1.5

The sigma clipping algorithm has been changed. It now uses a scaled Median Absolute Deviation (MAD) to derive a robust RMS for the clipping and not anymore the interquartile range (IQR). The MAD gives better results for the case for low number statistics and a high fraction of pixels affected by e.g. cosmic ray hits. Furthermore, the library integration in the pipeline slightly changed - see section 2.1 for details.

2.2.6 Release version 0.1.0

The *High Level Data Reduction Library* release version 0.1.0 can be included from

http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-0.1.0

Various methods for bad pixel detection are added in this release (see section 4.5)

2.2.7 Release version 0.0.3

The *High Level Data Reduction Library* release version 0.0.3 can be included from

http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-0.0.3

It is the first prototype release.

2.3 Dependencies

Relationship with *Common Pipeline Library* and other libraries:

The latest hdrl library depends on

- The Common Pipeline Library (CPL) version 7.0 or higher. **Please note that CPL must be compiled with wcs functionality available**
- The GSL - GNU Scientific Library version 1.16 or higher.

Chapter 3

Core Objects

3.1 Images

In order to handle simple error propagation tasks HDRL provides an image class `hdrl_image` which is similar to `cpl_image`. It carries both the data values and their associated errors and also contains a bad pixel mask in form of a binary cpl_mask. When crating a new `hdrl_image` from two existing cpl images (*image* and *error*) by using the function `hdrl_image_create(image, error)`, the bad pixel mask of the passed error-image (*error*) is completely ignored. The bad pixel mask associated with the passed image (*image*) becomes the only relevant bad pixel mask. Moreover, `hdrl_image_create` creates the relevant `hdrl_image` by copying the cpl images. Basic arithmetic operations, like addition, subtraction, multiplication, division, and collapse operations (mean, median, sum, ...) involving `hdrl_image` propagate the errors according to linear error propagation theory. Correlations between data is currently not accounted for.

Some operations, e.g division can add new bad pixels when the value can not be calculated, e.g. $x \div 0$. Moreover, if a value can be calculated but the corresponding propagated error can not (e.g. the propagated error would be $0 \div 0$), the tuple value-error is still considered valid, i.e. not marked as a bad pixel.

The API of `hdrl_image` is similar to `cpl_image`, please refer to the API reference for details.

3.1.1 Example usage

```
hdrl_image * my_function(cpl_image * input_image,
                         cpl_image * input_errors
                         const hdrl_image * bias1,
                         const hdrl_image * bias2,
                         double scale)
{
    /* create new hdrl_image from input images */
    hdrl_image * data = hdrl_image_create(input_image, input_errors);

    /* take mean with error propagation
       equivalent to hdrl_imagelist_collapse */
    hdrl_image * master_bias = hdrl_image_duplicate(bias1);
    hdrl_image_add_image(master_bias, bias2);
    hdrl_image_div_scalar(master_bias, 2., 0.);

    /* subtract bias with error propagation */
    hdrl_image_sub_image(data, master_bias);

    /* reject a pixel */
    hdrl_image_reject(data, 1, 1);
    hdrl_image_delete(master_bias);

    return data;
}
```

3.2 Image Lists

Similar to CPL that provides `cpl_imagelist` to store collections of equally dimensioned `cpl_image` objects, HDRL provides an equivalent `hdrl_imagelist`. The API is again similar but, like with images, the simple arithmetic operations and collapse operations propagate errors according to the linear error propagation theory. For details please refer to the reference.

3.2.1 Views

In addition to handling simple error propagation automatically `hdrl_imagelist` allows creating multiple views of the list which point to the same data buffers. These views can be used in any place regular `hdrl_imagelist` are used and they are also deleted with `hdrl_imagelist_delete` (in contrast to `cpl_image` views which must be deleted with `cpl_image_unwrap`). Also, any `hdrl_image` extracted from a view can be deleted using `hdrl_image_delete`.

Views allow applying operations on subsets of the imagelists without creating copies. This is especially useful when dealing with large swap or disk backed files.

Due to technical limitations in CPL only currently views of full rows of images are available.

3.2.1.1 Example

```
void process(hdrl_imagelist * list)
{
    hdrl_imagelist_add_scalar(list, 5. , 1.);
    hdrl_imagelist_mul_scalar(list, 2., 0.);
}

void process_big_data(hdrl_imagelist * large_list)
{
    size_t ny = 100;
    /* process the large list in chunks of several rows,
       this can be very beneficial if the images contained in the
       imagelist are allocated from a disk backed memory map as it
       performs more operations per load */
    for (size_t i = 0; i < ny; i += 10) {
        /* create a view of the large imagelist only containing 10 rows */
        hdrl_imagelist * view;
        view = hdrl_imagelist_row_view(large_list, i, i + 10);
        process(view);
        hdrl_imagelist_delete(view);
    }
    /* now the full input imagelist has been processed */
}
```

If the user has constraints on the amount of RAM that can be allocated, and need to chunk an imagelist, HDRL provides an iterator interface to row views of `hdrl_imagelist`.

```
/* create an iterator providing views containing blocksize rows
   the iterator owns the view and will take care of deallocating
   it on each iteration */
hdrl_iter * it = hdrl_imagelist_get_iter_row_slices(himlist, blocksize,
                                                    HDRL_ITER_OWN_OUTPUT);

/* hdrl_iter_next returns a view on the data or NULL when its done */
for (hdrl_imagelist * v = hdrl_iter_next(it);
     v != NULL;
     v = hdrl_iter_next(it)) {
    process(v);
}
hdrl_iter_delete(it);
```

3.2.2 Collapse Operations

Collapse operations like sum, mean or standard deviations can be performed with `hdrl_imagelist`. For operations were the error propagation formula is well defined, the errors will be accounted for to determine the result. In most cases a collapse operation only has one or two outputs, the result image and a integer contribution map counting how many values contributed to each pixel of the image. These collapse operations are called via `hdrl_imagelist_collapse`:

```
cpl_error_code hdrl_imagelist_collapse(
        const hdrl_imagelist     *   himlist,
        const hdrl_parameter     *   param,
        hdrl_image               **  out,
        cpl_image                **  contrib)
```

The `out` and `contrib` pointers are filled with allocated result images which must be deleted by the user when not required anymore.

The parameter `param` is a `hdrl_parameter` who's type defines the exact collapse method which is applied. Currently available are:

- `hdrl_collapse_mean_parameter_create()`
- `hdrl_collapse_median_parameter_create()`
- `hdrl_collapse_weighted_mean_parameter_create()`
- `hdrl_collapse_sigclip_parameter_create(kappalow, kappahigh, niter)`
- `hdrl_collapse_minmax_parameter_create(nlow, nhigh)`

Note that these parameters are dynamically allocated and must be deleted again. For convenience HDRL provides preallocated parameters for collapses which do not require any additional parameter. These can be used anywhere a regular parameter can be used and deletion with `hdrl_parameter_delete` has no effect on them:

- `HDRL_COLLAPSE_MEAN`
- `HDRL_COLLAPSE_MEDIAN`
- `HDRL_COLLAPSE_WEIGHTED_MEAN`

The sigma clipping and minmax collapse method can additionally return the low and hight rejection thresholds used to calculate the mean. These results can only be obtained with `hdrl_imagelist_collapse_sigclip` and `hdrl_imagelist_collapse_minmax` which takes two additional output pointer arguments which will be filled with the allocated results.

```
cpl_error_code hdrl_imagelist_collapse_sigclip(
        const hdrl_imagelist    *   himlist,
        double                      kappa_low,
        double                      kappa_high,
        int                         niter,
        hdrl_image              **  out,
        cpl_image               **  contrib,
        cpl_image               **  reject_low,
        cpl_image               **  reject_high);
```

```
cpl_error_code hdrl_imagelist_collapse_minmax(
        const hdrl_imagelist    *   himlist,
        double                      nlow,
        double                      nhigh,
        hdrl_image              **  out,
        cpl_image               **  contrib,
        cpl_image               **  reject_low,
        cpl_image               **  reject_high);
```

3.3 Spectrum

The struct `hdrl_spectrum1D` provides a convenient abstraction representing a 1D spectrum. The 1D spectrum is composed by two elements: the *wavelength* and the *flux*. The *wavelength* contains the wavelengths the flux is defined on. The *flux* contains the values the spectrum reaches for each wavelength. Therefore, a spectrum can be visualized as two sequences of values, wavelengths and fluxes. The `hdrl_spectrum1D` defines also a third component, the *wavelength scale*, which can be logarithmic or linear.

The wavelengths are considered error-free, therefore wavelength manipulation functions do not support error propagation. The functions manipulating fluxes, on the other hand, support error propagation. The flux contains also a bad pixel map to signal flux values considered unreliable.

The `hdrl_spectrum1D` does not require the wavelength to be sorted in a strictly monotonic increasing fashion. However, some operations (e.g. resampling) require the wavelengths to be sorted. The routines having this requirement will detected whether the wavelengths are sorted or not. If they are not sorted a copy of the spectrum's data will be silently sorted and then the operation will be performed on the sorted copy. The original spectrum will not be changed. It is therefore more efficient to provide the flux and wavelengths sorted in order to have better performance.

For an in-depth explanation of the `hdrl_spectrum1D` please refer to the API reference. Here, we provide the general guidelines. The API mainly consists of:

- Constructors: the key difference between them is in the way they handle the errors values on the flux;
- Getters: the functions allow to extract flux, wavelength, scale or the value of the i-th elements of the flux or of the wavelength;
- Vectorial flux manipulators: given two spectra, an operation (e.g. multiplication) is performed between corresponding elements of the two fluxes. Error propagation is used. The functions can mutate one of the provided spectra to be the output or they can output a new spectrum. We refer to the first as mutator operators. We refer to the latter as creator operators, they do not modify the input spectra, and they end with `_create`. These operations can be executed only if the two spectra are defined on the same frequencies, sorted in the same order. If this is not the case, an appropriate error code is set;
- Scalar flux manipulators: given a spectrum and a scalar, an operation (e.g. multiplication) is performed between each element of the spectrum and the scalar. Error propagation is used. Also here there are two versions of the operators, like in the vectorial case: mutator and creator operators.
- Wavelength manipulators: a set of functions that shift, multiply or divide the wavelength values by a scalar. Also here mutator and creator operators are available;
- Wavelength conversions: a set of functions that convert the scale of the wavelength, from logarithmic to linear and vice versa. Also here mutator and creator operators are available;
- Wavelength selectors: a subset of the flux is selected based on wavelengths values;
- Conversion functions to and from `cpl_table`.

3.3.1 Resampling

The spectrum resampling routine supports three modes: *interpolation*, *fitting* and *integration*. To switch between the modes a different `hdrl_parameter` has to be provided. Interpolation supports 5 interpolation algorithms, and fitting is provided in two variants: normal and windowed. Windowed fitting is experimental: therefore the API is not stable, it is recommended that windowed fitting is not used in production code. Please refer to the APIs documentation for the details.

3.3.2 Example usage

```
hdrl_spectrum1D * s1 = /* get first spectrum from the outside */
hdrl_spectrum1D * s2 = /* get second spectrum from the outside */

const hdrl_spectrum1D_wavelength spec_wav =
                    hdrl_spectrum1D_get_wavelength(s1);

/* Resample s2 on s1 wavelengths */
hdrl_parameter * params =
            hdrl_spectrum1D_resample_interpolate_parameter_create
            (hdrl_spectrum1D_interp_akima);


hdrl_spectrum1D * s3 =
                hdrl_spectrum1D_resample(s2, &spec_wav, params);

hdrl_parameter_delete(params);

/* Multiply every flux value by 0.4, and write the result in s3 */
hdrl_spectrum1D_mul_scalar(s3, (hdrl_value){0.4, 0.0});

/* Multiply every flux value by 1.4, and write the result in s4, s3 is
 * unchanged */
hdrl_spectrum1D * s4 = hdrl_spectrum1D_mul_scalar_create
                                (s3, (hdrl_value){1.4, 0.0});

/* Covert s4 to a table, having 4 columns: flux, wavelength, error, bpm
 * respectively.*/
cpl_table * table = hdrl_spectrum1D_convert_to_table(s4, "FLUX_COL",
                                        "WAV_COL", "FLUX_ERROR_COL",
                                        "FLUX_BPM_COL");
```

Chapter 4

Functionalities

4.1 Overscan

The overscan (or prescan) region on a CCD can consist of physical pixels on the detector that are not illuminated, or it can consist of a set of virtual pixels created by reading out the serial register either before or after transferring the charge from the CCD for each column/row. A detector may have multiple regions (amplifiers) and each one usually has an associated overscan region.

This module is intended to be applied to a single overscan region.

In HDRL the overscan correction value for each row/column of pixels is computed from a rectangular running sub-region of the full overscan region of the detector.

A small sub-region will increase the uncertainty of the correction but can account for rapid spatial changes in the bias level, while a large sub-region will decrease the uncertainty but smooth out spatial changes.

The actual correction value can be computed by estimating the *location/first moment* of the pixel value distribution within the sub-region. Several estimation methods, like mean, median or sigma/minmax-clipped[1] mean, are available.

A typical overscan correction may be essentially independent of row/column, or it may vary smoothly or with abrupt gradient changes. Hence the implementation of a location estimation method as opposed to the fitting of an analytical function.

When choosing the overscan region, note that the first few rows/columns that are read out after the image area will have an artificially high bias level due to the charge transfer efficiency not being 100%. It is recommended therefore to not include the 2-3 rows/columns right next to the image region in the overscan region.

An interesting phenomenon is that a very bright star/object may increase the bias level for the few rows/columns that it covers. Hence it is not advisable to set the size of the sub-region to a value that is much larger than the image point-spread-function FWHM.

This module allows the computation of the overscan correction for an image from a predefined overscan region. It also provides a function that applies the overscan correction to the image, using the result of the overscan computation.

4.1.1 Overscan Computation

The Overscan is computed using the following function call:

```
hdrl_overscan_compute_result * oscan_result = hdrl_overscan_compute(
        const cpl_image          *   source,
        const hdrl_parameter     *   params) ;
```

4.1.1.1 Inputs

The input parameters that need to be passed to the function are created with:

[1]For convenience we denote the minmax clipping algorithm and the kappa sigma clipping algorithm as sigma/minmax-clipped whenever the statement applies to both algorithm

```
hdrl_parameter * params =
    hdrl_overscan_parameter_create(hdrl_direction correction_direction,
                                   double ccd_ron,
                                   int box_hsize,
                                   hdrl_parameter collapse,
                                   hdrl_parameter rect_region) ;
```

The collapsing parameters are created with one of the following function calls:

```
hdrl_parameter * collapse =
    hdrl_collapse_mean_parameter_create(void) ;
    hdrl_collapse_median_parameter_create(void) ;
    hdrl_collapse_weighted_mean_parameter_create(void) ;
    hdrl_collapse_sigclip_parameter_create(double kappa_low,
                                           double kappa_high,
                                           int niter) ;
    hdrl_collapse_minmax_parameter_create(double nlow,
                                          double nhigh) ;
```

The Overscan region specification is done with:

```
hdrl_parameter * rect_region =
    hdrl_rect_region_parameter_create(cpl_size llx,
                                      cpl_size lly,
                                      cpl_size urx,
                                      cpl_size ury) ;
```

4.1.1.2 Outputs

The results of the Overscan computation (mostly 1D CPL or HDRL images of type DOUBLE) are contained in a structure that can be accessed with the following functions:

```
hdrl_image * correction =
    hdrl_overscan_compute_result_get_correction(oscan_result) ;
cpl_image * contribution =
    hdrl_overscan_compute_result_get_contribution(oscan_result) ;
cpl_image * chi2 =
    hdrl_overscan_compute_result_get_chi2(oscan_result) ;
cpl_image * red_chi2 =
    hdrl_overscan_compute_result_get_red_chi2(oscan_result) ;
cpl_image * sigclip_reject_low =
    hdrl_overscan_compute_result_get_sigclip_reject_low(oscan_result) ;
cpl_image * sigclip_reject_high =
    hdrl_overscan_compute_result_get_sigclip_reject_high(oscan_result) ;
```

```
cpl_image * minmax_reject_low =
    hdrl_overscan_compute_result_get_minmax_reject_low(oscan_result) ;
cpl_image * minmax_reject_high =
    hdrl_overscan_compute_result_get_minmax_reject_high(oscan_result) ;
```

4.1.1.3 Algorithm

Figure 4.1.1.1 gives a general description of the overscan computation method.

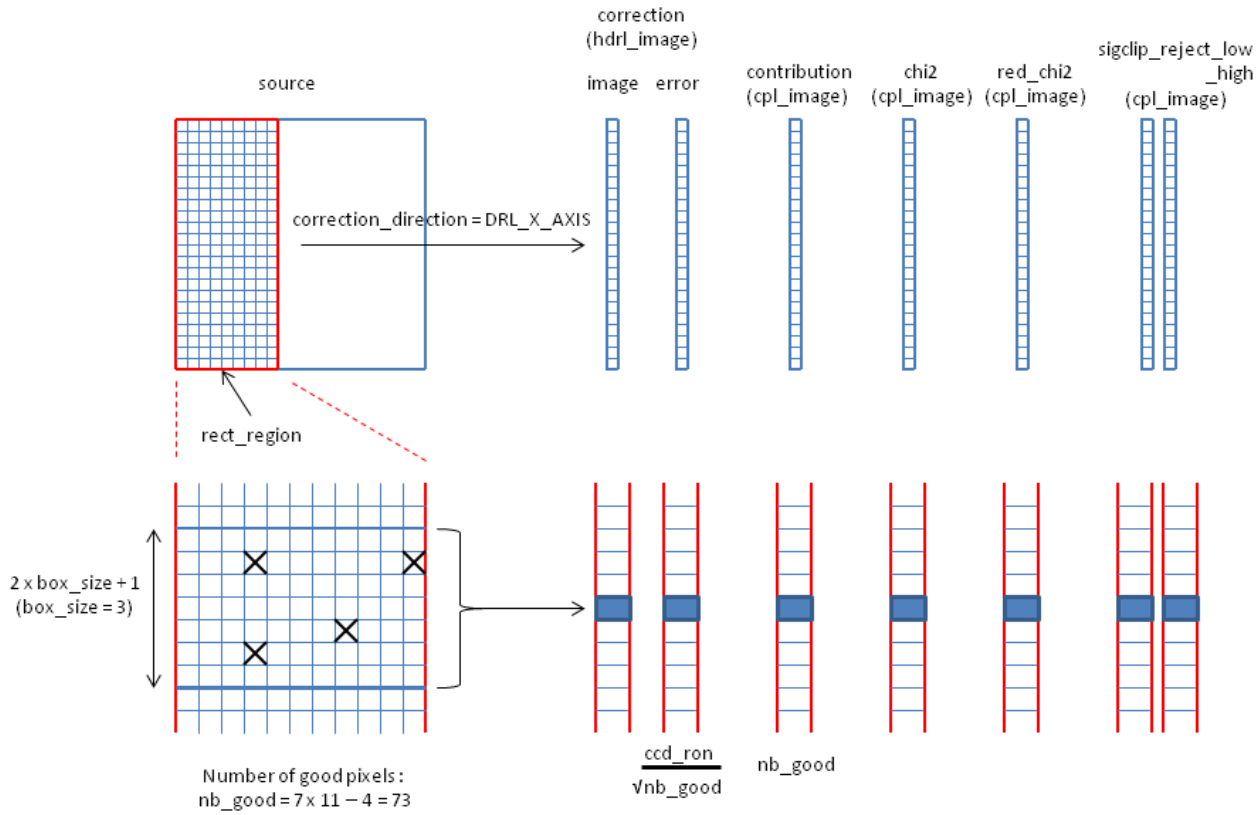

Figure 4.1.1.1: *Overscan Computation Algorithm description.*

The module extracts the pixel data for the overscan region from the input image, including the bad pixel map. The source image may contain more than the overscan region that is actually needed by the computation.

`rect_region` defines the overscan region in the source image. The bad pixels that might be present in the image are taken into account (i.e. excluded from the computations).

Each pixel of the resulting 1D images are computed from a running sub-window of the overscan region (see Figure 4.1.3.1). `box_hsize` defines the half size of the sub-window used for the computation (in the direction orthogonal to `correction_direction`). If `box_hsize` value is `HDRL_OVERSCAN_FULL_BOX`, the calculation is done on the whole overscan region instead of a running sub-window. In this case, all pixels of the resulting 1D images will be identical.

The running sub-window defined by `box_hsize` is used to compute the output pixel of the different 1D images. When reaching the edges, the sub-window is shrinking such that the current pixels stays in the center of the sub-window.

`correction_direction` is `HDRL_X_AXIS` (resp. `HDRL_Y_AXIS`) if the overscan region has to be collapsed along the X (resp. Y) axis in order to create the 1D resulting images (correction, error, contribution, the χ^2 (chi2) and the reduced χ^2 (`red_chi2`), additionally `sigclip_reject_low` and `_high` if the collapsing method is sigma/minmax-clipping rejection).

The possible collapsing methods can be mean, median, weighted mean, sigma clipping, or minmax rejection. The appropriate function need to be used at the creation of the collapse parameters.

In case the collapsing method used is sigma-clipping, an iterative (niter iterations) rejection is applied in the sub-window before the computation of the results. `kappa_low` and `kappa_high` are used to define what pixels need to be rejected.

The uncertainty on each pixel value is assumed to be `ccd_ron`. `ccd_ron` is the CCD read-out noise. The parameter is mandatory, must be strictly non-negative. It is used for the error, the chi2 and the `red_chi2` computation.

The output `hdrl_overscan_compute_result` object contains the Overscan computation results that can be accessed with dedicated accessor functions (see the 'Output' section):

correction is a 1D HDRL image of type double.

Its image part contains the Overscan correction values computed from the good pixels of the running sub-window (mean, weighted mean, median or mean after rejection, depending on the collapsing method used).

Its error part contains $\frac{ccd\_ron}{\sqrt{contribution}}$ for mean, weighted mean and sigma/minmax-clipping collapsing methods. In case of the median collapsing, it contains $\sqrt{\frac{\pi}{2}} \times \frac{ccd\_ron}{\sqrt{contribution}}$ if contribution is strictly greater than 2 pixels and $\frac{ccd\_ron}{\sqrt{contribution}}$ when the contribution is 1 or 2 pixels.

`contribution` is a 1D CPL image of type integer. It contains the number of good pixels of the input running sub-window in case the collapsing method is mean, weighted mean or median, and the remaining pixels after the rejection in case the sigma/minmax-clipping method is used.

chi2 is a 1D CPL image of type double. It contains $\sum_i (\frac{source_i - correction}{ccd\_ron})^2$ where *i* is running over the good pixels of the running sub-window.

`red_chi2` is a 1D CPL image of type double. It contains the reduced χ^2, i.e the χ^2 divided by the number of contributing pixels.

`sigclip_reject_low` and `_high` are 1D CPL images of type double. They are only returned in case the

sigma/minmax-clipping collapsing method is used. They indicate the final thresholds of the siglicp/minmax rejection.

4.1.2 Overscan Correction

The Overscan is corrected using the following function call:

```
hdrl_overscan_correct_result * oscan_correct = hdrl_overscan_correct(
        const hdrl_image                    *   source,
        const hdrl_parameter                *   region,
        const hdrl_overscan_compute_result  *   oscan_result) ;
```

4.1.2.1 Inputs

The `oscan_result` parameter is obtained by the Overscan computation (see 4.1.1).

The `region` defines the region in source that needs to be corrected. It is created with:

```
hdrl_parameter * region =
    hdrl_rect_region_parameter_create(cpl_size llx,
                                      cpl_size lly,
                                      cpl_size urx,
                                      cpl_size ury) ;
```

4.1.2.2 Outputs

The results of the Overscan correction are contained in a structure that can be accessed with the following functions:

```
hdrl_image * corrected =
    hdrl_overscan_correct_result_get_corrected(oscan_correct) ;
cpl_image * badmask =
    hdrl_overscan_correct_result_get_badmask(oscan_correct) ;
```

`hdrl_overscan_correct_result_get_corrected` returns a copy of the input image with the data row or column wise corrected by the overscan values.

`hdrl_overscan_correct_result_get_badmask` returns an integer image with all pixels considered bad by the algorithm set to 1 and all good pixels set to 0. Pixels of the input image will be considered bad if there is no overscan value available to correct it (which is defined by the badpixel mask of the overscan values image from `hdrl_overscan_compute`)

4.1.2.3 Algorithm

`source` is the input HDRL image that needs to be corrected. Usually the image part is the one passed to `hdrl_overscan_compute()` to compute the overscan correction parameters.

`region` specifies which region of the `source` image must be corrected. If NULL, the whole image is corrected. The size of `region` must fit the contents of `oscan_result`.

If the `correction_direction` value is `HDRL_X_AXIS` (resp. `DRL_Y_AXIS`), the correction will be applied row by row (resp. column by column) to the input image, ie correction value number i will be subtracted to row (resp. column) number i.

The error part of the source HDRL image is used for error propagation. Uncertainties in the overscan computation result, if present, are added in quadrature to the uncertainties on the input image pixel values.

`oscan_result` contains all the parameters for the overscan correction. It has been produced by `hdrl_overscan_compute()`.

The output `hdrl_overscan_correct_result` object contains the following members:

corrected is a HDRL image of type double of the same size as source.

Its image part had all its pixels within the specified region subtracted the proper correction. The pixels outside the specified region remain unchanged.

Its error part had all the pixels within the specified region set to $\sqrt{overscan\_computation.error^2 + source\_error^2}$, which is the standard linear error propagation. The pixels outside the specified region remain unchanged.

baddmask is a CPL image identifying the bad pixels.

4.1.3 Examples

4.1.3.1 Overscan Computation

In the following example, we are using a UVES image whose left 24 pixel columns are used for the overscan computation. The image size is 1074x1500.

The Overscan region is defined with:

```
hdrl_parameter * os_region =
    hdrl_rect_region_parameter_create(1, 1, 24, 1500) ;
```

We would like for the collapsing method to use a sigma-clipping algorithm with 5 iterations and with 3.0 for the low and high rejection kappa values:

```
hdrl_parameter * os_collapse =
    hdrl_collapse_sigclip_parameter_create(3.0, 3.0, 5) ;
```

The specified Overscan region needs to be collapsed along the X axis, the CCD RON is 10.0, and the running box half size is 5 pixels.

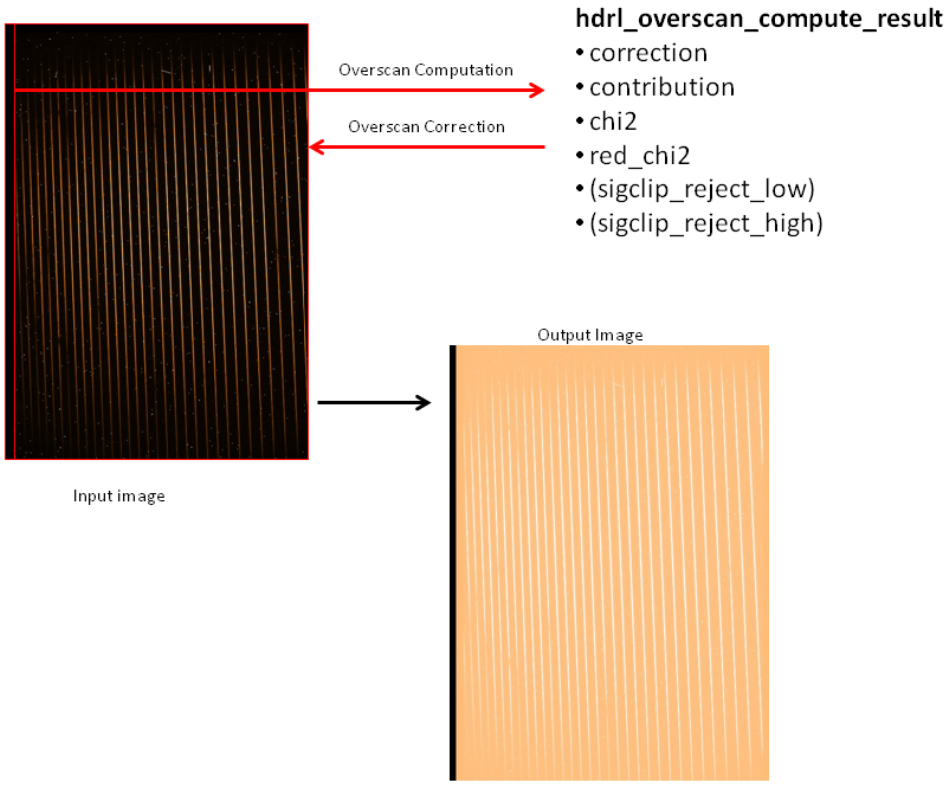

Figure 4.1.3.1: *Overscan Computation Example.*

In these conditions, the Overscan computation parameter is created with:

```
hdrl_parameter * os_params =
    hdrl_overscan_parameter_create(HDRL_X_AXIS, 10.0, 5,
                                   os_collapse, os_region);
```

The actual computation occurs here:

```
hdrl_overscan_compute_result * os_computation =
    hdrl_overscan_compute(image, os_params);
```

Let's not destroy `os_computation` as we need it to apply the correction in the next section. What we do not need any more are the parameters. They are deleted with:

```
hdrl_parameter_delete(os_region) ;
hdrl_parameter_delete(os_collapse) ;
hdrl_parameter_delete(os_params) ;
```

For convenience, a `_destroy` function is also provided, that deletes the parameter and recursively deletes the contained parameters:

```
hdrl_parameter_destroy(os_params) ;
```

4.1.3.2 Overscan Correction

The Overscan Computation result needs now to be used to correct the input image. We only want to correct the non-overscan part of the image.

In order to do that, we specify the image region we want the correction to be applied on:

```
hdrl_parameter * apply_region =
    hdrl_rect_region_parameter_create(25, 1, 1074, 1500) ;
```

The wished bad pixel mask code is 2, the correction function call is:

```
hdrl_overscan_correct_result * os_corrected =
    hdrl_overscan_correct(image, apply_region, 2, os_computation) ;
```

A number of accessor function give access to the elements of the `os_corrected` result object.

Finally, the used parameters, or the created objects need to be deleted:

```
hdrl_overscan_compute_result_delete(os_computation) ;
hdrl_parameter_delete(apply_region) ;
hdrl_overscan_correct_result_delete(os_corrected) ;
```

4.1.4 Overscan Parameters in a Recipe

In the context of a recipe, all input parameters needed by the Overscan computation may be provided as input parameters. There is a convenient interface for doing this. This avoids to update all recipes whenever a new parameter is offered by the Overscan computation.

A function call allows to generate a parameter list, another one allows to parse it in order to generate the input Overscan computation parameters:

```
/* Create Wished Default Parameters for the Overscan region */
hdrl_parameter * rect_region_def =
    hdrl_rect_region_parameter_create(1, 1, 20, 0);
/* Create Wished Default for the Sigma-Clipping Parameters */
hdrl_parameter * sigclip_def =
    hdrl_collapse_sigclip_parameter_create(3., 3., 5);

/* Create Overscan Computation Parameters List */
cpl_parameterlist * os_comp_parlist =
```

```
    hdrl_overscan_parameter_create_parlist("hdrldemo_bias", "oscan",
            "alongX", 0, 0., rect_region_def, "MEDIAN", sigclip_def);
hdrl_parameter_delete(rect_region_def);
hdrl_parameter_delete(sigclip_def);
```

The created parameter list `os_comp_parlist` simply needs to be fully or partly appended to the recipe parameter list to make those parameters available to the user:

```
esorex --man-page hdrldemo_bias
...
--oscan.correction-direction        : Correction Direction. <alongX | alongY> [alongX]
--oscan.box-hsize                   : Half size of running box in pixel
--oscan.ccd-ron                     : Readout noise in ADU. [10.0]
--oscan.calc-llx                    : Lower left x pos. (FITS) defining the region. [1]
--oscan.calc-lly                    : Lower left y pos. (FITS) defining the region. [1]
--oscan.calc-urx                    : Upper right x pos. (FITS) defining the region. [20]
--oscan.calc-ury                    : Upper right y pos. (FITS) defining the region. [0]
--oscan.collapse.method             : Method used for collapsing the data. [MEDIAN]
--oscan.collapse.sigclip.kappa-low  : Low kappa factor. [3.0]
--oscan.collapse.sigclip.kappa-high : High kappa factor. [3.0]
--oscan.collapse.sigclip.niter      : Maximum number of clipping iterations. [5]
...
```

If parlist is the recipe parameters list, we also offer a function to generate the Overscan parameters directly from the recipe parameters:

```
/* Parse the Overscan Parameters */
hdrl_parameter * os_params =
    hdrl_overscan_parameter_parse_parlist(parlist, "hdrldemo_bias");
```

4.2 Bias

Bias frames give the read out of the CCD detector for zero integration time with the shutter closed. That means that any frame acquisition with shutter open (either a calibration or a science observation) need to be bias corrected. Bias frames are acquired in optical (e.g UVB or VIS) domain. In order to have a more accurate estimation of the bias, they are typically taken as a set of (usually five) bias frames that have to be combined into a master to increase the S/N level.

4.2.1 Algorithm - short description

Combining multiple images to one master image is handled by `hdrl_imagelist_collapse` (see section 3.2.2). In order to be robust against outliers (like cosmic ray hits) it is recommended to use the median or sigma clipping collapse methods. Both methods are robust against outliers but the median has a low statistical efficiency so its result will have a higher uncertainty than collapsing images with few outliers via sigma clipping.

4.2.2 Functions

The master bias is computed by using the general `hdrl_imagelist_collapse` function:

```
cpl_error_code hdrl_imagelist_collapse(
        const hdrl_imagelist     *   himlist,
        const hdrl_parameter     *   param,
        hdrl_image               **  out,
        cpl_image                **  contrib)
```

The `out` and `contrib` pointers are filled with allocated result images which must be deleted by the user when not required anymore.

4.2.3 Inputs

The input parameter `param` that need to be passed to the collapse function is a `hdrl_parameter` who's type defines the exact collapse method which is applied. Currently available are:

- `hdrl_collapse_mean_parameter_create()`
- `hdrl_collapse_median_parameter_create()`
- `hdrl_collapse_weighted_mean_parameter_create()`
- `hdrl_collapse_sigclip_parameter_create(kappalow, kappahigh, niter)`
- `hdrl_collapse_minmax_parameter_create(nlow, nhigh)`

Note that these parameters are dynamically allocated and must be deleted again. For convenience HDRL provides preallocated parameters for collapses which do not require any additional parameter. These can be used anywhere a regular parameter can be used and deletion with `hdrl_parameter_delete` has no effect on them. See section 3.2.2 for more details.

4.2.4 Outputs

In most cases a collapse operation only has one or two outputs, the resulting master-image (`out`) and a integer contribution map (`contrib`) counting how many values contributed to each pixel of the image. The sigma clipping and minmax collapse method can additionally return the low and hight rejection thresholds used to calculate the mean. These results can only be obtained with `hdrl_imagelist_collapse_sigclip` and `hdrl_imagelist_collapse_minmax` See section 3.2.2 for the usage of these functions.

4.2.5 Examples

```
hdrl_image* master;
cpl_image* contrib_map;
/* initialize the parameters of a sigma clipping combination */
hdrl_parameter * sigclip_par;
sigclip_par = hdrl_collapse_sigclip_parameter_create(3., 3., 5)

/* do the combination */
hdrl_imagelist_collapse(data, sigclip_par ,&master, &contrib_map);

/* cleanup */
hdrl_parameter_delete(sigclip_par) ;
hdrl_imagelist_delete(data);
```

4.2.6 Master Bias Parameters in a Recipe

In the context of a recipe, all input parameters needed by the master bias computation may be provided as input parameters. The creation of `cpl_parameter` objects for the recipe interface is facilitated by:

- `hdrl_collapse_parameter_create_parlist`
- `hdrl_collapse_parameter_parse_parlist`

The former provides a `cpl_parameterlist` which can be appended to the recipe parameter list, the latter parses the recipe parameter list for the previously added parameters and creates a ready to use `hdrl_parameter`.

```
/* prepare defaults for recipe */
hdrl_parameter * sigclip_def =
    hdrl_collapse_sigclip_parameter_create(3., 3., 5);
```

```
/* create parameters with context RECIPE_NAME under the
   collapse hierarchy */
cpl_parameterlist * pbiascollapse =
    hdrl_collapse_parameter_create_parlist(
            RECIPE_NAME, "collapse", "MEDIAN", sigclip_def);
/* add to recipe parameter list, change aliases,
   disable not required ones etc.
 */
hdrl_parameter_delete(sigclip_def);
```

This will create a esorex man-page as follows:

```
esorex --man-page hdrldemo_bias
...

--collapse.method             : Method used for collapsing the data.
                                <MEAN | WEIGHTED_MEAN | MEDIAN | SIGCLIP | MINMAX> [MEDIAN]
--collapse.sigclip.kappa-low  : Low kappa factor for kappa-sigma clipping algorithm. [3.0]
--collapse.sigclip.kappa-high : High kappa factor for kappa-sigma clipping algorithm. [3.0]
--collapse.sigclip.niter      : Maximum number of clipping iterations
                                for kappa-sigma clipping. [5]
...
```

4.3 Dark

The dark current of a CCD detector can be measured by using frames with an exposure time larger then zero (usually it is in the same order of magnitude as the science data) during which the shutter remains closed. Various dark frames are then combined into a master dark in order to increase the signal to noise of the dark current. Before doing this master frame combination, depending on the detector and the wavelength regime, the overscan and/or master bias has to be subtracted from the single frames.

As the master dark creation algorithm is essentially the same as the master bias algorithm (with the possible difference that a master bias image and a multiplicative scaling for the exposure time has to be taken into account) we point the user to the master bias section (see Section 4.2)

4.3.1 Algorithm - short description

See Section 4.2.1

4.3.2 Functions

See Section 4.2.2

4.3.3 Inputs

See Section 4.2.3

4.3.4 Outputs

See Section 4.2.4

4.3.5 Examples

Following code is a example how one can use HDRL to produce a master dark from a set of raw dark frames and their associated errors.

```
/* input raw darks as hdrl_images, so they can have an error */
hdrl_image * raw_darks[ndarks] = ...;
/* multiplicative scaling factor (e.g. exposure time) */
double scaling_factor[ndarks] = ...;
hdrl_imagelist  * ilst_os_cor_raw_darks = hdrl_imagelist_new();

/* overscan correct all raw darks */
for (size_t i = 0; i < ndarks; i++) {
    hdrl_overscan_compute_result * os_com_res;
```

```
    hdrl_overscan_correct_result * os_cor_res;
    hdrl_bitmask_t rejectcode = 256;
    os_com_res = hdrl_overscan_compute(img, os_parameters);
    os_cor_res = hdrl_overscan_correct(raw_darks[i], region_parameter,
                                       rejectcode, os_com_res);
    /* take ownership of the corrected images from result object */
    hdrl_image * cor_dark =
        hdrl_overscan_correct_result_unset_corrected(os_cor_res);
    /* scale images */
    hdrl_image_mul_scalar(cor_dark, scaling_factor[i], 0.);
    hdrl_imagelist_set(ilst_os_cor_raw_darks, cor_dark, i);

    hdrl_overscan_compute_result_delete(os_com_res);
    hdrl_overscan_correct_result_delete(os_cor_res);
}

/* subtract the master bias from the darks */
hdrl_imagelist_sub_image(ilst_os_cor_raw_darks, master_bias);

/* collapse corrected darks to the master_dark */
hdrl_image * master_dark; cpl_image * contrib;
hdrl_imagelist_collapse(ilst_os_cor_raw_darks, HDRL_COLLAPSE_MEDIAN,
                        &master_dark, &contrib);
```

4.3.6 Master Dark Parameters in a Recipe

See Section 4.2.6.

4.4 Flat

Flat field frames give information on the response of the detector, allowing to measure variations in efficiency at small (pixel-to-pixel), intermediate (fringing) and large (the blaze function) scale. Whereas in the optical regime the flats are usually twilight or dome flats, in the NIR regime the flats sometimes consists of a pair of illuminated (denoted as ON-Flats) and not-illuminated (OFF-Flats) frames.

In this section we present algorithms to combine single flatfields into a master flatfield with increased SNR.

4.4.1 High frequency master flatfield

This algorithm derives the high frequency part of the master flatfield – often also denoted as pixel-to-pixel variation.

4.4.1.1 Algorithm - short description

The algorithm first divides each input image by the smooth image obtained with a median filter (the latter is controlled by the parameters `filter_size_x` and `filter_size_y`). Concerning the error propagation, the smoothed image is considered to be noiseless, i.e. the relative error associated to the normalised images is the same as the one of the input images. Then all residual images are collapsed into a single master flatfield. The collapsing can be done with all methods currently implemented in HDRL (see section 3.2.2 for an overview).

To distinguish between illuminated and not illuminated regions/pixels (i.e. orders in an echelle flat image) the user may provide an optional static mask `stat_mask` to the algorithm. In this case the smoothing procedure is done twice, once for the illuminated region and once for the blanked region. This ensures that the information of one region does not influence the other regions during the smoothing process.

4.4.1.2 Functions

The high frequency master flatfield is computed using the following function

```
cpl_error_code hdrl_flat_compute(
        hdrl_imagelist          *  hdrl_data,
        const cpl_mask          *  stat_mask,
        const hdrl_parameter    *  collapse_params,
        hdrl_parameter          *  flat_params,
        hdrl_image              ** master,
        cpl_image               ** contrib_map)
```

The output `master` and `contrib_map` products are filled with the resulting master flat and associated contribution map images. The corresponding object data has to be deleted by the user when not required any more.

4.4.1.3 Inputs

The list of flatfields that should be combined into a master flatfield is passed to the function with the `hdrl_imagelist` `hdrl_data` input function-parameter. If a static mask is needed to distinguish between illuminated and dark regions, this can be passed with the `cpl_mask stat_mask` input function-parameter. The static mask is an optional input function-parameter, i.e. one can pass NULL if no static mask is needed.

Note that the function will overwrite the input imagelist in order to conserve memory. Its contents after the call are undefined and it must be deleted by the caller.

The `collapse_params` controls the collapsing algorithm and the `flat_params` controls the smoothing algorithm.

The input `flat_params` parameter is created by the function

```
hdrl_parameter * hdrl_flat_parameter_create(
        cpl_size                filter_size_x,
        cpl_size                filter_size_y,
        hdrl_flat_method        method)
```

In order to use the high frequency algorithm for master flatfield determination, the `method` parameter must be set to `HDRL_FLAT_FREQ_HIGH`. The `filter_size_x` and `filter_size_y` parameters defining the smoothing kernel must have an odd integer value greater then 0. If both values are set to 1, no smoothing is done by the algorithm.

The `collapse_param` parameter input of the collapse function is a `hdrl_parameter`. Its type defines the collapse method which is applied. Currently available are the following collapsing parameter creation utilities:

- `hdrl_collapse_mean_parameter_create()`
- `hdrl_collapse_median_parameter_create()`
- `hdrl_collapse_weighted_mean_parameter_create()`
- `hdrl_collapse_sigclip_parameter_create(kappalow, kappahigh, niter)`
- `hdrl_collapse_minmax_parameter_create(nlow, nhigh)`

Note that these parameters are dynamically allocated and must be deleted when not needed. For convenience HDRL provides preallocated parameters for collapsing operations (e.g. `HDRL_COLLAPSE_MEAN`) which do not require any additional parameters. These can be used anywhere a regular parameter can be used (deletion with `hdrl_parameter_delete` has no effect on them). See section 3.2.2 for more details.

4.4.1.4 Outputs

The result is a `hdrl_image` stored in the function-parameter `master` containing the master flatfield and its associated error as well as an integer contribution map (`contrib`) counting how many values contributed to each pixel of the image.

4.4.2 Low frequency master flatfield

This algorithm derives the low frequency part of the master flatfield – often also denoted as the shape of the flatfield.

4.4.2.1 Algorithm - short description

The algorithm multiplicatively normalizes the input images by the median (considered to be noiseless) of the image to unity. An optional static mask `stat_mask` can be provided to the algorithm in order to define the pixels that should be taken into account when computing the normalisation factor. This allows the user to normalize the flatfield e.g. only by the illuminated section. In the next step, all normalized images are collapsed into a single master flatfield. The collapsing can be done with all methods currently implemented in hdrl (see section 3.2.2). Finally, the master flatfield is smoothed by a median filter controlled by `filter_size_x` and `filter_size_y`. The associated error of the final masterframe is the error derived via error propagation of the previous steps, i.e. the smoothing itself is considered noiseless. Please note that, if the smoothing kernel is set to unity, i.e. `filter_size_x = 1` and `filter_size_y = 1` no final smoothing will take place but the resulting masterframe is simply the collapsed normalized flatfields.

4.4.2.2 Functions

The called functions are the same as for the high frequency masterflat (see section 4.4.1.2)

4.4.2.3 Inputs

The input parameters are the same as for the high frequency masterflat (see section 4.4.1.3) with the exception that in order to trigger the computation of the low frequency flatfield, the flatfield `method` must be set to `HDRL_FLAT_FREQ_LOW`.

4.4.2.4 Outputs

As for the high frequency masterflat algorithm, the result is a `hdrl_image` stored in the function-parameter `master` containing the master flatfield and its associated error as well as an integer contribution map (`contrib`) counting how many values contributed to each pixel of the image.

4.4.3 Examples

```
hdrl_image* master;
cpl_image* contrib_map;

/* initialize the parameters of a sigma clipping combination */
hdrl_parameter * collapse_params;
collapse_params = hdrl_collapse_sigclip_parameter_create(3., 3., 5)
```

```
/* initialize the parameters of the flatfield */
hdrl_parameter * flat_params;
flat_params = hdrl_flat_parameter_create(7, 7, HDRL_FLAT_FREQ_LOW) ;

/* Do the actual flatfield computation */
hdrl_flat_compute(input_imagelist, stat_mask, collapse_params,
                                    flat_params, &master, &contrib_map);

/* cleanup the parameter*/
hdrl_parameter_delete(collapse_params);
hdrl_parameter_delete(flat_params);
```

4.4.4 Master Flat Parameters in a Recipe

In the context of a recipe, all input parameters needed by the master flat computation may be provided as input parameters.

The creation of `cpl_parameter` objects for the collapse recipe interface is facilitated by:

- `hdrl_collapse_parameter_create_parlist`
- `hdrl_collapse_parameter_parse_parlist`

The former provides a `cpl_parameterlist` which can be appended to the recipe parameter list, the latter parses the recipe parameter list for the previously added parameters and creates a ready to use collapse `hdrl_parameter`.

The creation of `cpl_parameter` objects for the flat recipe interface is facilitated by:

- `hdrl_flat_parameter_create_parlist`
- `hdrl_flat_parameter_parse_parlist`

The former provides a `cpl_parameterlist` which can be appended to the recipe parameter list, the latter parses the recipe parameter list for the previously added parameters and creates a ready to use flat `hdrl_parameter`.

```
/* prepare defaults for recipe */


/* Create collapse default parameters */
hdrl_parameter * sigclip_def =
    hdrl_collapse_sigclip_parameter_create(3., 3., 5);

/* create collapse parameters with context RECIPE_NAME under the
   collapse hierarchy */
```

```
cpl_parameterlist * pflatcollapse =
    hdrl_collapse_parameter_create_parlist(
            RECIPE_NAME, "collapse", "SIGCLIP", sigclip_def);
/* add to recipe parameter list, change aliases,
   disable not required ones etc.
 */
hdrl_parameter_delete(sigclip_def);

/* Create flat default parameters */
hdrl_parameter * deflts = hdrl_flat_parameter_create(5, 5,
        HDRL_FLAT_FREQ_LOW);
/* create flat parameters with context RECIPE_NAME under the
   flat hierarchy */
cpl_parameterlist * flat_param = hdrl_flat_parameter_create_parlist(
            RECIPE_NAME, "flat", deflts) ;
/* add to recipe parameter list, change aliases,
   disable not required ones etc.
 */
hdrl_parameter_delete(deflts) ;
```

This will create a esorex man-page as follows:

```
esorex --man-page hdrldemo_flat_nir
...

--collapse.method:             Method used for collapsing the data. <MEAN | WEIGHTED_MEAN
                               | MEDIAN | SIGCLIP | MINMAX> [SIGCLIP]
--collapse.sigclip.kappa-low:  Low kappa factor for kappa-sigma  clipping algorithm. [3.0]
--collapse.sigclip.kappa-high: High kappa factor for kappa-sigma clipping algorithm. [3.0]
--collapse.sigclip.niter:      Maximum number of clipping iterations for kappa-sigma
                               clipping. [5]
...

--flat.filter-size-x  :        Smoothing filter size in x-direction. [5]
--flat.filter-size-y  :        Smoothing filter size in y-direction. [5]
--flat.method         :        Method to use for the master flatfield calculation.
                               <low | high> [low]
...
```

4.5 Bad-pixel detection

4.5.1 Bad-pixel detection on a single image

The recipe derives bad pixels on individual images.

4.5.1.1 Algorithm - short description

The algorithm first smoothes the image by applying the methods described below. Then it subtracts the smoothed image and derives bad pixels by thresholding the residual image, i.e. all pixels exceeding the threshold are considered as bad. To compute the upper and lower thresholds, it measures a robust rms (a properly scaled Median Absolute Deviation), which is then scaled by the parameters `kappa_low` and `kappa_high`. Furthermore, the algorithm is applied iteratively for a number of times defined by the parameter `maxiter`. During each iteration the newly found bad pixels are ignored. Please note, that the thresholding values are applied as median (residual-image) $\pm$ thresholds. This makes the algorithm more robust in the case that the methods listed below are not able to completely remove the background level, e.g due to an exceeding number of bad pixels in the first iteration.

Two methods are currently available to derive a smoothed version of the image:

- Applying a filter like e.g. a median filter to the image. The filtering can be done by all modes currently supported by cpl and is controlled by the filter-type `filter`, the border-type `border`, and by the kernel size in x and y, i.e. `smooth_x`, and `smooth_y`.
- Fitting a Legendre polynomial to the image of order `order_x` in x and `order_y` in y direction. This method allows you to define `steps_x` $\times$ `steps_y` sampling points (the latter are computed as the median within a box of `filter_size_x` and `filter_size_y`) where the polynomial is fitted. This substantially decreases the fitting time for the Legendre polynomial.

4.5.1.2 Functions

The bad pixels are computed using the following function call

```
cpl_mask * bpm_2d = hdrl_bpm_2d_compute(
        const hdrl_image        *   img_in,
        const hdrl_parameter    *   params)
```

4.5.1.3 Inputs

The input parameters that need to be passed to the function for smoothing the image by a filter are created by

```
hdrl_parameter * params = hdrl_bpm_2d_parameter_create_filtersmooth(
        double              kappa_low,
        double              kappa_high,
```

```
        int                 maxiter,
        cpl_filter_mode     filter,
        cpl_border_mode     border,
        int                 smooth_x,
        int                 smooth_y);
```

The input parameters that need to be passed to the function for smoothing the image by fitting a polynomial

```
hdrl_parameter * params = hdrl_bpm_2d_parameter_create_legendresmooth(
        double              kappa_low,
        double              kappa_high,
        int                 maxiter,
        int                 steps_x,
        int                 steps_y,
        int                 filter_size_x,
        int                 filter_size_y,
        int                 order_x,
        int                 order_y);
```

4.5.1.4 Outputs

The result is a mask of type cpl_mask containing the newly found bad pixels. Please note that already known bad pixels given to the routine will not be included in the output mask.

4.5.2 Bad-pixel detection on a stack of identical images

The recipe derives bad pixels on a stack of (identical) images

4.5.2.1 Algorithm - short description

This algorithm assumes that the mean level of the different images are the same, if this is not the case, the master-image as described below will be biased. The algorithm first collapses the stack of images by using the median to generate a master-image. Then it subtracts the master image from each individual image and derives the bad pixels on the residual-images by thresholding, i.e. all pixels exceeding the threshold are considered as bad.

Three methods are currently available to derive the bad pixels on the residual images:

- `method = HDRL_BPM_3D_THRESHOLD_ABSOLUTE`: It uses `kappa_low` and `kappa_high` as absolute threshold.
- `method = HDRL_BPM_3D_THRESHOLD_RELATIVE`: It scales the measured rms on the residual-image with `kappa_low` and `kappa_high` and uses it as threshold. For the rms a properly scaled Median Absolute Deviation (MAD) is used.

- `method = HDRL_BPM_3D_THRESHOLD_ERROR`: It scales the propagated error of each individual pixel with `kappa_low` and `kappa_high` and uses it as threshold.

4.5.2.2 Functions

The bad pixels are computed using the following function call

```
cpl_imagelist * bpm_3d = hdrl_bpm_3d_compute(
        const hdrl_imagelist     *   imglist_in,
        const hdrl_parameter     *   params)
```

4.5.2.3 Inputs

The input parameters that need to be passed to the function for are created by

```
hdrl_parameter * hdrl_bpm_3d_parameter_create(
        double               kappa_low,
        double               kappa_high,
        hdrl_bpm_3d_method   method);
```

4.5.2.4 Outputs

The result is a cpl_imagelist containing the newly found bad pixels for each input image with the same pixel coding as for a cpl bad pixel mask, i.e. 0 for good pixels and 1 for bad pixels. Please note that already known bad pixels given to the routine will not be included in the output mask.

4.5.3 Bad-pixel detection on a sequence of images

The recipe derives bad pixels on a sequence of images by fitting a polynomial along each pixel sequence of the images.

4.5.3.1 Algorithm - short description

Three options are available to convert the information from the fit into a bad pixel map.

- chi-method: Relative cutoff on the chi distribution of all fits. Pixels with chi values which exceed of `mean` $\pm$ `cutoff` $\times$ `standard deviation` are considered bad.
- fit-coefficient-method: Relative cutoff on the distribution of the fit coefficients. Pixels with fit coefficients which exceed of `mean` $\pm$ `cutoff` $\times$ `standard deviation` are considered bad.
- p-value-method: Pixels with low `p-value`. When the errors of the pixels are correct the `p-value` can be interpreted as the probability with which the pixel response fits the chosen model.

4.5.3.2 Functions

The bad pixels are computed using the following function call

```
cpl_error_code hdrl_bpm_fit_compute(
        const hdrl_parameter *  params,
        const hdrl_imagelist *  imglist_in,
        const cpl_vector     *  sample_position,
        cpl_image            ** out_mask)
```

4.5.3.3 Inputs

The input parameters that need to be passed to the function when using the chi-method are:

```
hdrl_parameter * hdrl_bpm_fit_parameter_create_rel_chi(
        int                 degree,
        double              rel_chi_low,
        double              rel_chi_high);
```

The input parameters that need to be passed to the function when using the fit-coefficient-method are:

```
hdrl_parameter * hdrl_bpm_fit_parameter_create_rel_coef(
        int                 degree,
        double              rel_coef_low,
        double              rel_coef_high);
```

The input parameters that need to be passed to the function when using the p-value-method are:

```
hdrl_parameter * hdrl_bpm_fit_parameter_create_pval(
        int                 degree,
        double              pval);
```

4.5.3.4 Outputs

The result is an integer cpl_image (denoted out_mask) containing the newly found bad pixels for each input image.
For the `rel_coef` parameter the value of out_mask encodes the coefficient that was outside the relative threshold as a power of two. E.g. if coefficient 0 and 3 of the fit where not within the threshold for a pixel, it will have the value $2^0 + 2^3 = 9$. All other parameters return an out_mask with nonzero values marking pixels outside the selection thresholds.

Please note that already known bad pixels given to the routine will not be included in the output mask.

4.5.4 Cosmic Ray Hits detection

This recipes determines cosmic-ray-hits and/or bad-pixels on a single image.

4.5.4.1 Algorithm - short description

This routine determines cosmic-ray-hits and/or bad-pixels following the algorithm (LA-Cosmic) described in van Dokkum, PASP,113,2001,p1420-27). HDRL does not use use error model as described in the paper but the error image passed to the function. Moreover it does several iterations with `max_iter` defining an upper limit for the number of iterations, i.e. the iteration stops if no new bad pixels are found or `max_iter` is reached. In each iteration this function replaces the detected cosmic ray hits by the median of the surroundings 5x5 pixels taking into account the pixel quality information. The input parameter `sigma_lim` and `f_lim` refere to σ_{lim} and f_{lim} as described in the paper mentioned above.

4.5.4.2 Functions

The bad pixels/cosmics are computed using the following function call:

```
cpl_mask * bpm_lacosmic = hdrl_lacosmic_edgedetect(
        const hdrl_image        *   img_in,
        const hdrl_parameter    *   params)
```

4.5.4.3 Inputs

The input parameters that need to be passed to the function for are created by:

```
hdrl_parameter * hdrl_lacosmic_parameter_create(
        double              sigma_lim,
        double              f_lim,
        int                 max_iter);
```

4.5.4.4 Outputs

The result is a mask of type cpl_mask containing the newly found bad pixels / cosmic-ray-hits. Please note that already known bad pixels given to the routine will not be included in the output mask.

4.5.5 Examples

To follow Overscan layout we need to make an example of how to define a bad pixel parameter creation for each supported method.

4.5.6 Bad-Pixel Parameters in a Recipe

4.5.6.1 Bad-pixel detection on a single image

In the following example, we are using bias or dark images where we set the kappa sigma parameters kappa_low and kappa_high to 3, 3, and maxiter to 2, the number of sampling coordinates in X and Y direction, step_x, step_y to 20, 20, the median filter X and Y sizes parameters filter_size_x, filter_size_y to 11, 11, the Legendre polynomial order parameters order_x, order_y to 3, 3.

```
hdrl_parameter * bpm_2d_param =
    hdrl_bpm_2d_parameter_create_legendresmooth(3., 3., 2, 20, 20,
    11, 11, 3, 3) ;
```

Then the bad pixel locations are stored in a mask by the following function which receives as inputs an hdrl image, himage, and the previously defined bad pixels:

```
cpl_mask * mask = hdrl_bpm_2d_compute(himage, bpm_2d_param) ;
```

4.5.6.2 Bad-pixel detection on a set of images

In the following example, we are using linearity flat images where we set the kappa sigma parameters kappa_low and kappa_high to 3,3, and maxiter to 2, the filter mode parameter to CPL_FILTER_MEDIAN, the filter border parameter to CPL_BORDER_FILTER, the smoothing kernel X and Y size parameters filter_size_x,filter_size_y to 3,3.

```
hdrl_parameter * bpm_2d_param =
    hdrl_bpm_2d_parameter_create_filtersmooth(3, 3, 2,
    CPL_FILTER_MEDIAN, CPL_BORDER_FILTER, 3, 3) ;
```

Then the newly detected bad pixel locations are stored in a cpl_imagelist by the following function which receives as inputs an hdrl image, himage, and the previously defined bad pixels:

```
imlist = hdrl_bpm_3d_compute(himlist, bpm_param) ;
```

4.5.6.3 Cosmic Ray Hits detection

In the following example, we are using raw frames representing deep exposures where we set the sigma_lim parameter to 20, the f_lim parameter to 2.0, and the max_iter parameter to 5.

```
hdrl_parameter * params =
            hdrl_lacosmic_parameter_create(20., 2.0, 5);
```

Then the cosmic ray hits locations are stored in a mask by the following function which receives as inputs an hdrl image, himage, and the previously defined bad pixels:

```
cpl_mask * mask = hdrl_lacosmic_edgedetect(image, params);
```

4.6 Strehl

The most commonly used metric for evaluating the AO correction is the Strehl ratio. The Strehl ratio is defined as the ratio of the peak image intensity from a point source compared to the maximum attainable intensity using an ideal optical system limited only by diffraction over the telescope aperture. The Strehl ratio is very frequently used to perform the quality control of the scientific data obtained with the AO assisted instrumentation.

4.6.1 Algorithm - short description

The implemented HDRL function assumes that the input raw image contains a single object and it is already pre-processed. This means that the instrument signature (bad pixels, CRH detection, bias level and dark current subtraction, flat fielding etc.) has been removed and the contribute from the sky background subtracted.

This function allows also the user to correct a residual local sky background evaluated in an annular region centred on the peak of the (expected single) object PSF, by setting the values of the corresponding parameters, which control the minimum and maximum radii of the annular region, `bkg_radius_low`, `bkg_radius_high` (in arcsec units).

The PSF is identified and its integrated flux, whose determination is controlled by the parameter `flux_radius`, in arcsec, is normalized to 1.

Next the PSF barycentre is computed and used to generate the theoretical normalised PSF. This depends on:

i) the telescope pupil characteristics (telescope radius and central obstruction parameters, `m1_radius`, `m2_radius`);

ii) the wavelength (parameter `wavelength`) at which the image has been obtained, all expressed in m units;

iii) the detector pixel scale on sky in arcsec (parameters `pixel_scale_x`, `pixel_scale_y`).

Then the Strehl ratio is obtained by dividing the maximum intensity of the image PSF by the maximum intensity of the ideal PSF. The associated error is finally computed.

4.6.2 Functions

The Strehl of an image and its error are computed by the following function call:

```
hdrl_strehl_result
hdrl_strehl_compute(const hdrl_image * himg, hdrl_parameter* params)
```

where `hima` is the HDRL input image on which the Strehl has to be computed, and `params` is a `hdrl_parameter` structure specifying the parameters controlling the Strehl computation.

4.6.3 Inputs

The input parameters to be passed to the function to compute the Strehl are created by the following function call:

```
hdrl_parameter * params = hdrl_strehl_parameter_create(
        double              wavelength,
        double              m1_radius,
        double              m2_radius,
        double              pixel_scale_x,
        double              pixel_scale_y,
        double              flux_radius,
        double              bkg_radius_low,
        double              bkg_radius_high);
```

4.6.4 Outputs

The function creates an output `hdrl_strehl_result` structure containing the following quantities:

```
typedef struct {
    /* computed strehl value and its propagated error */
    hdrl_value strehl_value;
    /* computed x and y position of the star peak */
    double star_x, star_y;
    /* star peak and its propagated error */
    hdrl_value star_peak;
    /* star flux and its propagated error */
    hdrl_value star_flux;
    /* star background and its propagated error */
    hdrl_value star_background;
    /* star background error estimated from image
     * on normal data sqrt(pi / 2) larger that star_background error as
     * it is estimated via a median */
    double computed_background_error;
    /* number of pixels used for background estimation */
    size_t nbackground_pixels;
} hdrl_strehl_result;
```

4.6.5 Examples

If the user would like to compute the Strehl of an image observed at 1.635 μm with a telescope of 5.08 m diameter with a central obstruction of 1.8288 m, and the detector pixel size on Sky is 33.1932 mas$\times$33.1932 mas (1 mas = 0.001 arcsec), integrating the observed image PSF over a radius of 1.5 arcsec where the sky background is computed on the annular region defined by the internal and external radii respectively of 1.5 and 2.5 arcsec, the following function calls would do the job:

```
  hdrl_strehl_result          strehl;
```

```
hdrl_parameter * params = hdrl_strehl_parameter_create(1.635e-6,
5.08, 1.8288, 0.0331932, 0.0331932, 1.5, 1.5, 2.5);

strehl = compute_strehl(hima, params);
```

4.6.6 Strehl Parameters in a Recipe

The computation of the Strehl is usually part of a more complex recipe to reduce standard stars or science objects. Concerning the parameters to be defined to compute this quantity the user has to create a `cpl_parameter` object as described in Section 4.6.3.

4.7 Fringing

The major causes for fringing are the night sky emission lines (mainly OH transition) in the upper atmosphere. If monochromatic lines are reflected within the CCD they interfere producing a fringe-pattern (usually variable over night). Some instruments show considerable fringing in certain wavebands like the z-band. To correct for the latter a master fringe image has to be created and subtracted before doing photometric measurements.

4.7.1 Master fringe computation

This algorithm extracts a master fringe from a sequence of images.

4.7.1.1 Algorithm - short description

The algorithm consists of two main steps: i) Using a Gaussian mixture model (see below) it estimates the background (scalar) and the fringe scaling-factor (amplitude - scalar) of each image. The background is then subtracted and the amplitude is used to multiplicatively normalize the background-subtracted image to the same scale. ii) The resulting images are then collapsed into a master-fringe image. The collapsing can be done with all methods currently implemented in HDRL (see section 3.2.2 for an overview). Moreover, different masks controlling the algorithm in different stages can be passed to the hdrl function.

- `ilist_obj`: One mask per input fringe image flagging the astronomical objects in the input image. Each step in the algorithm takes this mask into account. The images should have a value of 0 if a pixel does not belong to an object and unity otherwise.
- `stat_mask`: A optional single cpl mask used to exclude image regions with weak fringes. This mask is only taken into account in step i), i.e. when computing the scaling factors but ignored for step ii).

The algorithm combines the bad pixel map from `ilist_fringe`, the object mask `from ilist_obj` and the static mask from `stat_mask` for the fringe computation itself, but uses only the combined bad pixel map and object mask for the final collapsing. This ensures that the master fringe is also calculated in regions excluded by the static mask.

Please note, that the function directly works on the passed hdrl imagelist (`ilist_fringe`) in order to save memory thus modifying this imagelist. Moreover, the scaling factor derived and used in this function is considered to be noiseless, i.e. the associated error is supposed to be zero.

Gaussian mixture Model in detail

For the master-fringe estimation, the algorithm model the pixel intensity distribution in a given image as a mixture of two Gaussian distributions, whose means are the background and the fringe amplitudes, respectively:

Thus the density function $f(x)$ of the intensity of an individual pixel is modeled as follows

$$f(x) = c_1\, e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + c_2\, e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}}. \tag{4.1}$$

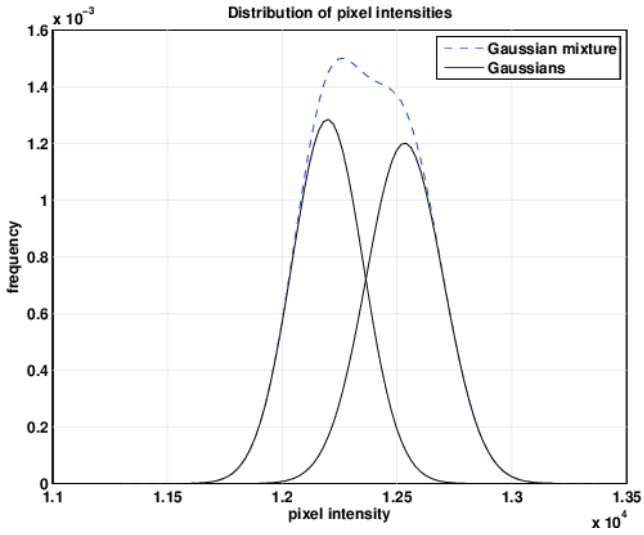

Figure 4.7.1.1: Density of a mixture of two normal distributions, whose means are the background and the fringe amplitudes, respectively.

The means μ_1 and μ_2 ($\mu_1 < \mu_2 = \mu_1 + a$) are proportional to the background amplitude and the fringe pattern amplitude, respectively. These values are used for normalization of the background and fringe amplitudes before stacking.

The parameters of the two Gaussian components are estimated from the density function of the pixel intensities by a nonlinear least squares fit algorithm. The algorithm requires as its input an estimated density function. Such an estimate is calculated in a preprocessing step as a truncated Hermite series:

$$f(x) \approx \sum_{n=0}^{p} c_n h_n \left(\frac{x-\mu}{\sigma} \right), \tag{4.2}$$

where h_n is the normalized Hermite function

$$h_n(x) = \pi^{-\frac{1}{4}} 2^{-\frac{n}{2}} (n!)^{-\frac{1}{2}} (1)^n e^{\frac{x}{2}^2} \frac{d^n}{dx^n} \left(e^{-x^2} \right), \tag{4.3}$$

μ and σ are, respectively, the sample mean and the sample standard deviation of pixel intensities in the given image. The truncation parameter p is found experimentally, $p = 20$ has been sufficient so far. The Hermite coefficients c_n are computed as follows

$$c_n = \frac{1}{\sigma N} \sum_{i=1}^{N} h_n \left(\frac{I_i - \mu}{\sigma} \right), \tag{4.4}$$

where the summation extends over all pixel intensities $I_1, \ldots, I_N$, N is the total number of pixels, and $n = 0, \ldots, p$.

A detailed description of the algorithm can be found in appendix D

4.7.1.2 Functions

The master fringe is computed using the following function

```
cpl_error_code hdrl_fringe_compute(
        hdrl_imagelist          *  ilist_fringe,
        const cpl_imagelist     *  ilist_obj,
        const cpl_mask          *  stat_mask,
        const hdrl_parameter    *  collapse_params,
        hdrl_image              ** master,
        cpl_image               ** contrib_map,
        cpl_table               ** qctable)
```

The output `master` and `contrib_map` products are filled with the resulting master fringe and associated contribution map images. The output `qctable` product contains the background level (in table column `Background_level`) and the fringe amplitude (in table column `Fringe_amplitude`) computed by the algorithm for each input image. The corresponding object data has to be deleted by the user when not required any more.
Please note that if the background level and the fringe amplitude can not be computed a warning is issued. Moreover, in this case a background level of 0 and a fringe amplitude of 1 is assumed, i.e. the original image is used in the following steps.

4.7.1.3 Inputs

The list of images affected by fringes that should be combined to form the master fringe is passed to the function with the hdrl imagelist `ilist_fringe` input function-parameter. If object masks are needed to distinguish between sky and object pixels (very important!) they can be passed with the cpl imagelist `ilist_obj` input function-parameter. Moreover, if a static mask is needed to distinguish between regions with prominent and weak fringes it can be passed with the cpl mask `stat_mask` input function-parameter.

Please note that the static mask `stat_mask` and the object masks `ilist_obj` are optional input function-parameters, i.e. one can pass NULL if no mask should be used.

The `collapse_params` parameter controls the collapsing algorithm of the master fringe computation and is a `hdrl_parameter`. Its type defines the collapse method which is applied. Currently available are the following collapsing parameter creation utilities:

- `hdrl_collapse_mean_parameter_create()`
- `hdrl_collapse_median_parameter_create()`
- `hdrl_collapse_weighted_mean_parameter_create()`
- `hdrl_collapse_sigclip_parameter_create(kappalow, kappahigh, niter)`
- `hdrl_collapse_minmax_parameter_create(nlow, nhigh)`

Note that these parameters are dynamically allocated and must be deleted when not needed. For convenience HDRL provides preallocated parameters for collapsing operations (e.g. `HDRL_COLLAPSE_MEAN`) which do not require any additional parameters. These can be used anywhere a regular parameter can be used (deletion with `hdrl_parameter_delete` has no effect on them). See section 3.2.2 for more details.

4.7.1.4 Outputs

The result is a `hdrl_image` stored in the function-parameter `master` containing the master fringe and its associated error as well as an integer contribution map (`contrib`) counting how many values contributed to each pixel of the image. Moreover, the output table `qctable` contains the derived background level in the table column `Background_level` and the fringe amplitude in table column `Fringe_amplitude` for each input image.

4.7.2 Master fringe correction

This algorithm scales and subtracts a master fringe from a sequence of images.

4.7.2.1 Algorithm - short description

The function subtracts a passed master fringe image (`masterfringe`) from a set of input images (`ilist_fringe`). The amplitude of the fringes is computed for each input image and used to properly rescale the correction image before subtraction.

The algorithm computes fringe amplitudes for each individual image by a least squares fit of a linear combination of the estimated master-fringe and a constant background. Specifically, the ith fringe F_i is estimated as

$$F_i = a_i F + b_i, \tag{4.5}$$

where F is the estimated stacked master-fringe, and b_i is a constant representing the background. The unknown constants a_i and b_i are computed by a standard least squares fit preformed over the unmasked pixels.

Moreover, different masks controlling the algorithm in different stages can be passed to the hdrl function e.g. a static mask (`stat_mask`) to exclude regions where the fringe is weak - sometimes essential for an accurate scaling estimation of noisy images. The algorithm combines the bad pixel map (from `ilist_fringe`), the object mask (from `ilist_obj`), and static mask (`stat_mask`) for the scaling computation of the master fringe, but only uses the bad pixel map when subtracting the master-fringe. The object mask and static mask are ignored in this step. This ensures that the master fringe is properly subtracted (with error propagation) in all regions not affected by the bad pixel mask.

Please note, that the function directly works on the passed hdrl imagelist (`ilist_fringe`) in order to save memory thus modifying this imagelist i.e. removing the fringes directly from `ilist_fringe`. Moreover, the scaling factor derived and used in this function is considered to be noiseless, i.e. the associated error is supposed to be zero.

A detailed description of the algorithm can be found in appendix D

4.7.2.2 Functions

The fringe correction is done by the following function

```
cpl_error_code hdrl_fringe_correct(
        hdrl_imagelist           *  ilist_fringe,
        const cpl_imagelist      *  ilist_obj,
        const cpl_mask           *  stat_mask,
        const hdrl_image         *  masterfringe,
        cpl_table                ** qctable);
```

The function one-by-one scales and subtracts the passed master-fringe image `masterfringe` from the images stored in `ilist_fringe`. The output `qctable` product contains the background level (in table column `Background_level`) and the fringe amplitude (in table column `Fringe_amplitude`) computed by the algorithm for each input image. The corresponding object data has to be deleted by the user when not required any more.
Please note that if the background level and the fringe amplitude can not be computed a warning is issued. Moreover, in this case a background level of 0 and a fringe amplitude of 0 is assumed, i.e. no correction will be applied to this image.

4.7.2.3 Inputs

The list of images that should be fringe corrected by the master fringe is passed to the function with the hdrl imagelist `ilist_fringe` whereas the master-fringe is passed in the `masterfringe` input function-parameter.

If object masks are needed to distinguish between sky and object pixels (very important!) they can be passed with the cpl imagelist `ilist_obj` input function-parameter. Moreover, if a static mask is needed to distinguish between regions with prominent and weak fringes it can be passed with the cpl mask `stat_mask` input function-parameter. Please note that the static mask `stat_mask` and the object masks `ilist_obj` are optional input function-parameters, i.e. one can pass NULL if no mask should be used.

4.7.2.4 Outputs

The algorithm directly modifies the input hdrl images `ilist_fringe` when performing the fringe correction, thus the result is stored in the latter. Moreover, the output table `qctable` contains the derived background level in the table column `Background_level` and the fringe amplitude in table column `Fringe_amplitude` for each input image.

4.8 Object catalogue generation

In order to provide astrometric and photometric calibration information, the *High Level Data Reduction Library* implements a functionality to generate a catalogue of detected objects (i.e. stars, galaxies).

Objects are detected and parametrised using the processed input image (`img`) and confidence map (`cnf`). A confidence map is similar to a weight map. It gives an indication of the relative quantum efficiency of each pixel, rather than an indication of the Poisson uncertainty on the pixel flux and should be normalised to a mean of 100. "Bad" pixels (either dead or hot) are given a confidence value of zero. If stacked images are used, then the confidence map of each single image should be propagated in order to get a confidence map of the mosaic image. This confidence map should then be re-normalized to a mean of 100 and passed to the routine as confidence map.

A high-level summary of the implemented data reduction sequence is:

- estimate the local sky background over the image and track any variations at adequate resolution to eventually remove them,
- detect objects/blends of objects and keep a list of pixels belonging to each blend for further analysis (see [2] for details)
- parametrise the detected objects, i.e. perform astrometry, photometry and a shape analysis.

4.8.1 Algorithm - short description

If requested, (`bkg_estimate`) the possibly-varying sky background is estimated and removed automatically, prior to object detection, using a combination of robust iteratively-clipped estimators.

Any variation in sky level over the frame is dealt with by forming a coarsely sampled background map grid (specified by `bkg_mesh_size`). Within each background grid pixel, an iteratively k-sigma clipped median value of 'sky' is computed based on the flux values within the grid pixel zone. A robust estimate of sigma is computed by using the Median of the Absolute Deviation (MAD) from the median. This will then be further processed to form the frame background image.

After removing the (possibly) varying background component, a similar robust estimate of the average sky level and sky noise per pixel is made. This allows to robustly obtain the detection threshold for object analysis.

Individual objects are detected using a "standard match" filter approach. Since the only sources difficult to locate are those marginally above the sky noise, to assume (after factoring in the confidence map information) the noise as uniform, is a good approximation. The majority of objects detected in this way will have a shape dominated by the point spread function (PSF), which thereby defines the filter to use (controlled by `bkg_smooth_fwhm`.)

To be detected an object should consist of at least `obj_min_pixels` contiguous pixels above a threshold controlled by `obj_threshold` $\times$ sky-rms. Moreover, one can also activate a deblending algorithm (`obj_deblending`) in order to disentangle overlapping objects.

Pixels are weighted in the catalogue generation according to their value in the confidence map. Hence if a pixel is marked as bad, then it is not included in the flux computation over the aperture. The number of bad pixels

with an aperture is reported in the 'Error_bit_flag' column of the output table. The presence of bad pixels will also be reflected in the average confidence for the aperture (column 'Av_conf').

Please note that one also has to specify the effective gain (`det_eff_gain`) and the (`det_saturation`) limit to the routine. The gain is mainly used for the error estimation of the various object measurements whereas the saturation limit is used to mark and exclude saturated pixels.

A detailed description of the algorithm and the output table can be found in appendix E

4.8.2 Functions

The object catalogue, the background and the segmentation map are computed by the following function call:

```
hdrl_catalogue_result * hdrl_catalogue_compute(
      const cpl_image * image,
      const cpl_image * confidence_map,
        const cpl_wcs * wcs,
       hdrl_parameter * param)
```

where `image` is the input image, `confidence_map` is the (optional) confidence map, `wcs` contains the (optional) world coordinate system and `params` is a `hdrl_parameter` structure specifying the parameters controlling the catalogue generation.

4.8.3 Inputs

The input parameter to be passed to the function are created by the following function call:

```
hdrl_parameter * param = hdrl_catalogue_parameter_create(
      int                    obj_min_pixels,
      double                 obj_threshold,
      cpl_boolean            obj_deblending,
      double                 obj_core_radius,
      cpl_boolean            bkg_estimate,
      int                    bkg_mesh_size,
      double                 bkg_smooth_fwhm,
      double                 det_eff_gain,
      double                 det_saturation,
      hdrl_catalogue_options resulttype);
```

where `obj_min_pixels` is the minimum pixel area for each detected object, `obj_threshold` is the detection threshold in sigma above sky, `obj_deblending` is a parameter to switch on/off deblending, `obj_core_radius` is the value of the core radius in pixels, `bkg_estimate` is the parameter to switch on/off the background estimation from the input image, `bkg_mesh_size` is the background smoothing box size, `bkg_smooth_gauss_fwhm` is the FWHM of the Gaussian kernel used in convolution for object detection, `det_eff_gain` is the detector gain value to convert ADU counts to electron counts, `det_saturation`

is the detector saturation value, and `resulttype` is the `hdrl_catalogue_options` parameter which controls the different result types. The latter is a bit-map (the values can be combined with bitwise or) defining the computed and returned results:

- `HDRL_CATALOGUE_CAT_COMPLETE`: The full catalogue (see appendix XXX)
- `HDRL_CATALOGUE_BKG`: The background map
- `HDRL_CATALOGUE_SEGMAP`: The segmentation map [2]
- `HDRL_CATALOGUE_ALL`: `HDRL_CATALOGUE_CAT_COMPLETE|HDRL_CATALOGUE_BKG|HDRL_CATALOGUE_SEGMAP`

If a certain result type is not requested by setting the appropriate bit-pattern, the function returns a pointer to NULL.

4.8.4 Outputs

The function creates an output `hdrl_catalogue_result` structure containing the following quantities (or NULL, depending on the hdrl_catalogue_options `resulttype`):

```
typedef struct {
    /* object catalogue */
    cpl_table * catalogue;
    /* segmentation map */
    cpl_image * segmentation_map;
    /* background map */
    cpl_image * background;
    /* Additional information */
    cpl_propertylist * qclist;
} hdrl_catalogue_result;
```

The `hdrl_catalogue_result` object has to be deleted by the user when not required any more by calling the function `hdrl_catalogue_result_delete()`.

Currently the following additional informations are returned in the cpl_propertylist `qclist`:

```
APCOR1   / Stellar aperture correction - 1/2x core flux
APCOR2   / Stellar aperture correction - core/sqrt(2) flux
APCOR3   / Stellar aperture correction - 1x core flux
APCOR4   / Stellar aperture correction - sqrt(2)x core flu
APCOR5   / Stellar aperture correction - 2x core flux
APCOR6   / Stellar aperture correction - 2*sqrt(2)x core f
APCOR7   / Stellar aperture correction - 4x core flux
APCORPK  / Stellar aperture correction - peak height
```

[2] Contains the pixels attributed to each object, i.e. all pixels belonging to object XXX in the catalogue have a value of XXX in the segmentation map

and SYMBOL1 to SYMBOL9.

If the symbol keywords are present in the header of the catalogue fitsfile they are used by e.g. skycat or gaia to properly display the objects in the image.

The APCOR keywords can be used to convert (for stellar objects) fixed aperture fluxes to total magnitudes, as for this an aperture correction must be applied. This correction is calculated from the curve-of-growth and accounts for missing flux in a particular aperture. For example, the magnitude (m) of an object using the flux in the third bin ($i = 3$) may be calculated as

$$m = \texttt{MAGZPT} - 2.5\log_{10}(\texttt{APER\_FLUX\_3}/\texttt{EXPTIME}) - \texttt{APCOR3} \tag{4.6}$$

With APER_FLUX_3 being a column in the catalogue table, MAGZPT the zeropoint and EXPTIME the exposure time [3].

4.8.5 Examples

Detect objects on an image (img) with a given confidence map (cnf) and wcs informations (wcs):

```
cpl_image * img;
cpl_image * cnf;
cpl_wcs * wcs;

/* initialize the catalogue parameters */

hdrl_parameter * p =
     hdrl_catalogue_parameter_create(5,         /* obj_min_pixel */
                                     2.5,       /* obj_threshold */
                                     CPL_TRUE,  /* obj_deblending */
                                     3.0,       /* obj_core_radius */
                                     CPL_TRUE,  /* bkg_estimate */
                                     64,        /* bkg_mesh_size */
                                     2.0,       /* bkg_smooth_fwhm */
                                     2.8,       /* det_eff_gain */
                                     65200,     /* det_saturation */
                                     HDRL_CATALOGUE_ALL);

/* catalogue COMPUTATION */

hdrl_catalogue_result * res = hdrl_catalogue_compute(img, cnf, wcs, p);

/* save the catalogue, the background and the segmentation map */
```

[3] For more information, refer to http://casu.ast.cam.ac.uk/surveys-projects/vst/technical/catalogue-generation

```
  /* delete the result structure */
  hdrl_catalogue_result_delete(res);
```

4.8.6 Catalogue Parameters in a Recipe

In the context of a recipe, all input parameters needed by the catalogue computation may be provided as input parameters.

The creation of `cpl_parameter` objects for the catalogue recipe interface is facilitated by:

- `hdrl_catalogue_parameter_create_parlist`
- `hdrl_catalogue_parameter_parse_parlist`

The former provides a `cpl_parameterlist` which can be appended to the recipe parameter list, the latter parses the recipe parameter list for the previously added parameters and creates a ready to use catalogue `hdrl_parameter`.

```
 /* Create catalogue default parameters */

  hdrl_parameter * c_def =
      hdrl_catalogue_parameter_create(4, 2.5, CPL_TRUE, 5.0, CPL_TRUE, 64,
                                      2., 3.0, HDRL_SATURATION_INIT,
                                      HDRL_CATALOGUE_ALL);
  cpl_parameterlist * s_param =
      hdrl_catalogue_parameter_create_parlist(RECIPE_NAME, "", c_def);
  hdrl_parameter_delete(c_def) ;

  for (cpl_parameter * p = cpl_parameterlist_get_first(s_param);
          p != NULL; p = cpl_parameterlist_get_next(s_param))
      cpl_parameterlist_append(self, cpl_parameter_duplicate(p));
  cpl_parameterlist_delete(s_param);
```

This will create a esorex man-page as follows:

```
esorex --man-page hdrldemo_catalogue
...
  --obj.min-pixels          : Minimum pixel area for each detected object. [4]
  --obj.threshold           : Detection threshold in sigma above sky. [2.5]
  --obj.deblending          : Use deblending?. [TRUE]
  --obj.core-radius         : Value of Rcore in pixels. [5.0]
  --bkg.estimate            : Estimate background from input, if false it is assumed
                              into is already background corrected with median 0.
                              [TRUE]
  --bkg.mesh-size           : Background smoothing box size. [64]
  --bkg.smooth-gauss-fwhm   : The FWHM of the Gaussian kernel used in convolution
                              for object detection. [2.0]
```

```
  --det.effective-gain     : Detector gain value to rescale convert intensity to
                             electrons. [3.0]
  --det.saturation         : Detector saturation value. [inf.0]
...
```

If parlist is the recipe parameters list, we also offer a function to generate the Catalogue parameters directly from the recipe parameters:

```
/* Parse the Catalogue Parameters */

hdrl_parameter * p =
   hdrl_catalogue_parameter_parse_parlist(parlist, RECIPE_NAME);
```

4.9 Efficiency

This section deals with the HDRL module estimating the efficiency as a 1D function of wavelength.

The efficiency is used to monitor the system performance and health. It is calculated observing flux standard stars (in photometric conditions). Then, the observed 1D spectrum is compared with the reference spectrum, as it would be observed outside the Earth's atmosphere. The reference spectrum is provided by the user, usually via a catalog of standard stars.

4.9.1 Algorithm - short description

The efficiency is calculated according to

$$\epsilon(\lambda) = \frac{I_{std}(\lambda) \cdot 10^{[0.4(A_p - A_m)E_x^{(r)}(\lambda)]} \cdot G \cdot E_{ph}(\lambda)}{[T_{ex} A_{tel} I_{std-ref}^{(r)}(\lambda)]} \tag{4.7}$$

where:

- $I_{std}(\lambda)$ is the observed 1D spectrum [ADU] as function of wavelength [nm];
- $E_x^{(r)}(\lambda)$ is the resampled atmospheric extinction as function of wavelength[mag/airmass]. The wavelength is expressed in [nm]. The corresponding, user-provided spectrum will be here indicated as $E_x(\lambda)$;
- A_m is the airmass of the observed standard star spectrum;
- A_p is a parameter to indicate if the efficiency is computed at $A_m = 0$ or at a given value (usually the one at which the reference standard star spectrum may be tabulated);
- G is the gain of the detector in [e/ADU];
- $E_{ph}(\lambda)$ is the energy of one photon at wavelength λ. $E_{ph}(\lambda) = \frac{hc}{\lambda} = \frac{1.986 \cdot 10^{-12}}{\lambda} J/\mu m$. λ is expressed in μm;
- $I_{std-ref}^{(r)}(\lambda)$ resampled reference standard star spectrum. The wavelength is expressed in [nm]. The corresponding, user-provided spectrum is denoted as $I_{std-ref}(\lambda)$;
- $h = 6.62618 \cdot 10^{-34}$ Js (Planck constant);
- $c = 2.998 \cdot 10^{8}$ m/s (Speed of Light);

$E_x(\lambda)$ and $I_{std-ref}(\lambda)$ are resampled by the provided routine to match the wavelengths $I_{std}(\lambda)$ is defined on. If they already match the resampling is skipped. $\epsilon(\lambda)$ is defined for λ only if λ is contained in the interval $E_x(\lambda)$ and $I_{std-ref}(\lambda)$ are defined on.

4.9.2 Functions

The efficiency is computed by the following function call:

```
hdrl_spectrum1D * hdrl_efficiency_compute(const hdrl_spectrum1D * I_std,
const hdrl_spectrum1D * I_std_ref,const hdrl_spectrum1D * E_x,
const hdrl_parameter * pars)
```

where `I_std` is $I_{std}(\lambda)$, `I_std_ref` is $I_{std-ref}(\lambda)$ and `E_x` is $E_x(\lambda)$. `pars` is an `hdrl_parameter` structure containing all the other parameters required by 4.7, e.g. G.

4.9.3 Inputs

The input parameter `pars` to be passed to `hdrl_efficiency_compute` is created executing the following function call:

```
hdrl_parameter* hdrl_efficiency_parameter_create(
        const hdrl_value Ap, const hdrl_value Am, const hdrl_value G,
        const hdrl_value Tex, const hdrl_value Atel)
```

where the various input parameters are the ones in 4.7. Note that every parameter is an `hdrl_value`, hence an error can be also provided. In that case the routine performs error propagation.

4.9.4 Outputs

`hdrl_efficiency_compute` creates as output a `hdrl_spectrum1D` structure, containing the calculated efficiency.

4.9.5 Examples

The following listing shows how the implemented HDRL function calculates the efficiency using the user-provided spectra and parameters. The user has already provided the following spectra:

- `I_std`: observed spectrum, wavelength in [nm];
- `I_std_ref`: model of the spectrum of the observed star, wavelength in [nm]. If defined on different wavelengths than `I_std` the function will resample it using *Akima* interpolation;
- `E_x`: atmospheric extinction wavelength in [nm]. If defined on wavelengths different than `I_std` the function will resample it using *Akima* interpolation [4].

[4] See: https://www.gnu.org/software/gsl/manual/html_node/Interpolation-Types.html

```
hdrl_parameter * pars = hdrl_efficiency_parameter_create(
            (hdrl_value){1.2, 0.0},
            (hdrl_value){4.0, 0.0},
            (hdrl_value){1.0, 0.2},
            (hdrl_value){1.1, 0.0},
            (hdrl_value){2.2, 0.0});

hdrl_spectrum1D * sp_eff = hdrl_efficiency_compute(I_std, I_std_ref,
                                                   E_x, pars);

hdrl_parameter_delete(pars);
```

4.9.6 Notes

The fluxes of the observed and model spectra must be expressed in the same units. Usually the model is given in erg/s/cm2/Angstrom. Therefore the user should correct the observed spectrum accordingly, by, e.g. a factor equal to the sampling bin size expressed in Angstrom unit.

4.10 Response

This section deals with the HDRL module estimating the response as a 1D function of wavelength.

4.10.1 Algorithm - short description

The algorithm is divided in two parts: *Telluric correction* and *Response calculation*. In the provided implementation the *Telluric correction* is optional and can be disabled by the user.

4.10.1.1 Telluric Correction

The Telluric Correction functionality tries to find the best telluric model among the ones provided and uses it to correct the observed spectrum. The steps of the algorithm are as follows and they are repeated for each telluric model. The steps are grouped in two: *Spectrum correction* and *Quality evaluation*.

Spectrum correction: aims to correct the observed spectrum with the provided telluric model.

1. Convert wavelength scale (in nm) of the telluric model and the observation to natural logarithm if the ration lambda/FWHM is constant. A strong prerequisite is that the sampling step of both spectra is very small;
2. Extract the range of wavelengths needed for calculating the cross correlation (water vapour feature of medium strength);
3. Cross correlate over the extracted range of the telluric model with the observation spectrum. The cross correlation is done resampling both spectra on equally spaced samples, these two resampled spectrum are used only to determine the shift and subsequently discarded;
4. Fit a Gaussian around the cross correlation peak: finding the shift between the two spectra in sub-pixel precision and the FWHM;
5. Repeat the previous two steps refining the cross-correlation interval around the identified peak position;
6. Shift the telluric model by the shift obtained by the previous step;
7. Create a Gaussian kernel of size equal to the measured FWHM;
8. Convolve the shifted telluric model with the Gaussian kernel;
9. Convert the shifted and convolved telluric model to linear wavelength space (if needed). Resample it using integration to the wavelengths of the observed spectrum;
10. Divide the observed spectrum by the convolved and shifted telluric model: this is the *corrected observed spectrum*;

Quality evaluation: aims to evaluate the quality of the *corrected observed spectrum* calculated in the previous step.

1. Using the predefined continuum points, fit a spline to the *corrected observed spectrum* without smoothing;
2. Divide the *corrected observed spectrum* by the continuum fit;
3. Calculate the mean m_r of this ratio, skipping high absorption areas.

The quantity $q = |m_r - 1|$ is used as quality indicator for the fit of the telluric model. The model having the lowest q is deemed the best and its corresponding *corrected observed spectrum* is used in the *Response calculation* step. If telluric correction is disabled by the user, the *Response calculation* step uses the unmodified observed spectrum.

4.10.1.2 Response Calculation

Velocity compensation: this step can be disabled by the user and it is divided in two steps.

1. Determine the radial velocity of the observed (possibly telluric- corrected) spectrum by fitting a Gaussian to a known stellar line;
2. Apply the radial velocity to the stellar model spectrum as $\lambda = \lambda_0(1 + \frac{RV}{c})$.

The remaining steps to complete the response calculation are:

1. Calculate the raw response as described in equation 4.8;
2. Define fit points at which the median of the flux within a given window is determined, avoiding region of very high telluric absorption as well as the cores of the stellar lines;
3. Fit a spline through the points determined in the previous step.

The response is calculated according to

$$R(\lambda) = \frac{I^{(r)}_{std-ref}(\lambda) \cdot 10^{[0.4(A_p - A_m)E^{(r)}_x(\lambda)]} \cdot G \cdot T_{ex}}{I_{std}(\lambda)} \tag{4.8}$$

where:

- $I_{std}(\lambda)$ is the observed 1D spectrum [ADU] as function of wavelength [nm]. If telluric correction is enabled,
- $E^{(r)}_x(\lambda)$ is the resampled atmospheric extinction [mag/airmass] as function of wavelength [nm]. The corresponding, user-provided spectrum is denoted as $E_x(\lambda)$;
- A_m is the airmass of the observed standard star spectrum;
- A_p is a parameter to indicate if the response is computed at $A_m = 0$ or at a given value (usually the one at which the reference standard star spectrum may be tabulated);

- G is the gain of the detector in [e/ADU];
- $I^{(r)}_{std-ref}(\lambda)$ resampled reference standard star spectrum. The wavelength is expressed in [nm]. The corresponding, user-provided spectrum is denoted as $I_{std-ref}(\lambda)$;

$E_x(\lambda)$ and $I_{std-ref}(\lambda)$ are resampled by the provided routine to match the wavelengths $I_{std}(\lambda)$ is defined on. If they already match the resampling is skipped. $R(\lambda)$ is defined for λ only if λ is contained in the interval $E_x(\lambda)$ and $I_{std-ref}(\lambda)$ are defined on.

4.10.2 Functions

The response is computed by the following function call:

```
hdrl_response_result *
hdrl_response_compute(
        const hdrl_spectrum1D * obs_s,
        const hdrl_spectrum1D * ref_s,
        const hdrl_spectrum1D * E_x,
        const hdrl_parameter * telluric_par,
        const hdrl_parameter * velocity_par,
        const hdrl_parameter * calc_par,
        const hdrl_parameter * fit_par)
```

4.10.3 Inputs

The input parameters to be passed to `hdrl_response_compute` are created executing the following function calls:

```
hdrl_parameter * telluric_par =
hdrl_response_telluric_evaluation_parameter_create(telluric_models,
        x_corr_w_step, xcorr_half_win, xcorr_normalize,
        xcorr_is_shift_in_log, quality_areas, fit_areas,
        lmin, lmax);
```

`telluric_par` contains the parameters needed to perform the telluric correction, the parameters can be divided in parameters needed to perform che cross-correlation, and hence determine the shift, and parameters needed to calculate the quality of the fit.

```
hdrl_parameter * velocity_par =
hdrl_spectrum1D_shift_fit_parameter_create(velocity_wguess,
                velocity_range_wmin, velocity_range_wmax,
                velocity_fit_wmin, velocity_fit_wmax,
                velocity_fit_half_win);
```

`velocity_par` contains the parameters needed to perform the velocity correction.

```
hdrl_parameter * calc_par =
hdrl_response_parameter_create(Ap, Am, G, Tex);
```

`calc_par` contains the parameters needed to calculate the result of 4.8.

```
hdrl_parameter * fit_par =
    hdrl_response_fit_parameter_create(11, fit_points,
        fit_wrange, high_abs_regions);
```

`fit_par` contains the parameters needed to execute the final fit, after 4.8 is performed.

4.10.4 Outputs

The function returns a struct of type `hdrl_response_result` containing the response as an `hdrl_spectrum1D` and other parameters that can be used for QC purposes. For a complete description of such parameters, please, refer to the documentation of `hdrl_response_result`.

4.10.5 Examples

The following listing shows how to calculate the efficiency using the user-provided spectra and parameters. The user has already provided the following spectra:

- `I_std`: observed spectrum, wavelength in [nm];
- `I_std_ref`: model of the spectrum of the observed star, wavelength in [nm]. If defined on different wavelengths than `I_std` the function will resample it using *Akima* interpolation;
- `E_x`: atmospheric extinction wavelength in [nm]. If defined on different wavelengths than `I_std` the function will resample it using *Akima* interpolation [5].

```
hdrl_parameter * telluric_par =
hdrl_response_telluric_evaluation_parameter_create(telluric_models,
        x_corr_w_step, xcorr_half_win, xcorr_normalize,
        xcorr_is_shift_in_log, quality_areas, fit_areas,
        lmin, lmax);

hdrl_parameter * velocity_par =
hdrl_spectrum1D_shift_fit_parameter_create(velocity_wguess,
                velocity_range_wmin, velocity_range_wmax,
                velocity_fit_wmin, velocity_fit_wmax,
```

[5] See: https://www.gnu.org/software/gsl/manual/html_node/Interpolation-Types.html

```
                velocity_fit_half_win);

hdrl_parameter * calc_par =
hdrl_response_parameter_create(Ap, Am, G, Tex);

hdrl_parameter * fit_par =
    hdrl_response_fit_parameter_create(11, fit_points,
        fit_wrange, high_abs_regions);

hdrl_response_result *
response_res = hdrl_response_compute(
                        I_std,
                        I_std_ref,
                        E_x,
                        telluric_par,
                        velocity_par,
                        calc_par ,
                        fit_par);

hdrl_parameter_delete(telluric_par);
hdrl_parameter_delete(velocity_par);
hdrl_parameter_delete(calc_par);
hdrl_parameter_delete(fit_par);

/*... use the content of response_res  */

hdrl_response_result_delete(response_res);
```

4.10.6 Notes

- The fluxes of the observed and model spectra must be expressed in the same units. Usually the model is given in erg/s/cm2/Angstrom. Therefore the user should correct the observed spectrum accordingly, by, e.g. a factor equal to the sampling bin size expressed in Angstrom unit;
- The response should be determined in optimal (photo-metric) atmospheric conditions. Furthermore, in case of different modes, the results of one mode cannot be compared with the results of the other. For example, if the observation facility has support for a Laser Guide Star (LGS) observation can be obtained either using LGS or using a Natural Guide Star (NGS). Results obtained with NGS cannot be compared with the ones obtained with LGS;
- Observations should be obtained in the same optical setting mode.

4.11 Differential Atmospheric Refraction (DAR)

The effects of atmospheric refraction can be readily seen in any spectrograph or IFU. Atmospheric refraction will displace a source, or its spectrum, by an amount that is dependent on the source wavelength and the angular distance of the source from the zenith. This effect is due to the stratified density structure of our atmosphere, and the displacement will be toward the zenith and will be largest for shorter wavelengths. Because of this latter attribute, differential atmospheric refraction is not generally associated with infrared instruments. However, it can be readily seen in SINFONI data cubes observed with the largest wavelength coverage (H+K) and in the smallest pixel scales (25 mas). Here, the shift can be as large as 6 pixels from the beginning of the data cube to its end (see Figures H.3.1.1 and H.3.1.2).

This module uses an analytical approach to compute the expected differential refraction as a function of wavelength, zenith angle, and the refractive index of air which, in turn, depends on temperature, pressure, and water vapour pressure.

The `hdrl_dar` routines require the following inputs (all of which are, generally, available in the input data headers):

- the ambient atmospheric parameters: temperature, pressure, and humidity as contained in the environmental keyword headers: TEL.AMBI.TEMP, TEL.AMBI.PRES.START/END, and TEL.AMBI.RHUM, respectively.
- the instrument rotation angle on the sky
- the parallactic angle of instrument
- and, the world-coordinate system (WCS)

With this input, the differential atmospheric refraction is calculated (optionally, also with an error propagation), and provides an output of the X and Y-axis shifts as a function of the `cpl_vector` with the input wavelengths. The resulting shift corrections can be directly apply to the pixel image in order to correct this effect. The next section describes the algorithms used to make this calculation.

4.11.1 Algorithm - short description

This module contains routines to calculate the refractive index of air [6]. The main loop compute the differential refractive offset for the input reference wavelengths, and stores the shift in the coordinates, taking into account the instrument rotation angle on the sky and the parallactic angle at the time of observation.

The differential atmospheric refraction is calculated according to the algorithm from Filippenko (1982, PASP, 94, 715). This algorithm uses the Owens formula which converts relative humidity in water vapour pressure.

4.11.1.1 Owens saturation pressure

This function computes the saturation pressure using the J.C. Owens calibration (1967, Applied Optics, 6, 51-59). The saturation pressure is given by:

[6] See: `http://emtoolbox.nist.gov/Wavelength/Documentation.asp#AppendixA`, for the formulae used.

$$s_p = -10474 + 116.43\,T - 0.43284\,T^2 + 0.00053840\,T^3 \tag{4.9}$$

where verb+T+ is the temperature in Kelvins.

4.11.1.2 Filippenko refractive index

At sea level (P=760 `mm Hg` , T = 15 ^oC) the refractive index of dry air is given by (Edlén 1953; Coleman, Bozman, and Meggers 1960):

$$(n(\lambda)_{15,760} - 1)10^6 = 64.328 + \frac{29498.1}{146 - (1/\lambda)^2} + \frac{255.4}{41 - (1/\lambda)^2} \tag{4.10}$$

where λ is the wavelength of light in vacuum (microns). Since observatories are usually located at high altitudes, the index of refraction must be corrected for the lower ambient temperature and pressure (P) (Barrell 1951):

$$(n(\lambda)_{T,P} - 1) = (n(\lambda)_{15,760} - 1) \cdot \frac{P[1 + (1.049 - 0.0157\,T)10^{-6}\,P]}{720.883(1 + 0.003661\,T)} \tag{4.11}$$

In addition, the presence of water vapour in the atmosphere reduces $(n-1)10^6$ by:

$$\frac{0.0624 - 0.000680/\lambda^2}{1 + 0.003661\,T} f \tag{4.12}$$

where f is the water vapour pressure in mm of Hg, and T is the air temperature in ^oC (Barrell 1951) and is expressed as:

$$f = 0.75006158 \cdot s_p \cdot h \tag{4.13}$$

where s_p is the saturation pressure of equation 4.9 and h is the relative humidity in `[%]`.

4.11.2 Function

The differential atmospheric refraction is computed by the following function:

```
cpl_error_code hdrl_dar_compute(
        const hdrl_parameter * params,
        const hdrl_value       lambdaRef,
        const cpl_vector     * lambdaIn,
        cpl_vector           * xShift,
        cpl_vector           * yShift,
        cpl_vector           * xShiftErr,
        cpl_vector           * yShiftErr)
```

The output products contain the resulting differential atmospheric refraction, and include the shift in pixels (with their associated error). This results can be directly apply to the original image. Each output value is associated at each wavelength value included in the `lambdaIn` [Å] `cpl_vector`. The Filipenko method uses the value `lambdaRef` [Å] as the reference wavelength from which the differential atmospheric refraction is computed. In other words, the `xShift and yShift` values will be zero at `lambdaRef`. Generally, `lambdaRef` is chosen to be the mid-point of the wavelength range covered by the input data.

4.11.3 Inputs

The input parameters to be passed to `hdrl_dar_compute` are created by executing the following function

```
hdrl_parameter * hdrl_dar_parameter_create(
        hdrl_value              airmass,
        hdrl_value              parang,
        hdrl_value              posang,
        hdrl_value              temp,
        hdrl_value              rhum,
        hdrl_value              pres,
        cpl_wcs               * wcs)
```

The input parameters of the function are:

- `airmass`: Air mass.
- `parang`: Parallactic angle during exposure [deg].
- `posang`: Position angle on the sky [deg].
- `temp`: Temperature [^{o}C].
- `rhum`: Relative humidity [%].
- `pres`: Pressure [mbar].
- `wcs`: World coordinate system [deg].

Note that every parameter is an `hdrl_value`, hence an error can be also provided and the routine supports error propagation. See section 4.11.6 for a summary of the error input parameters in terms of the esorex call to this recipe.

Please note that during the testing of the algorithm it was discovered that the propagated errors in the source shifts, computed when the error of the temperature (temp.error) and pressure (pres.error) are specified, are over-estimated. We are currently investigating the source of this over-estimate and expect to be able to repair it soon. The core functionality of the algorithm, in computing the expected source shifts due to atmospheric refraction, are not affected by this issue.

4.11.4 Outputs

The function fills several input/output `cpl_vector` structures that contain the shift and error in each data axis.

This values are associated with each wavelength in the `lambdaIn cpl_vector` and they can be applied directly to the image. The user is free to apply these values either as a continuous corrective shift to the data (requiring, of course, interpolation), or to round the shifts to integer values and shift the data to a pixel grid.

4.11.5 Example

This section briefly describes the actual call to the function.

```
/* Declare and init variables */
hdrl_value airmass, parang, posang, temp, rhum, pres;
cpl_wcs *wcs = cpl_wcs_new_from_propertylist(plPri);
...
/* Create hdrl_parameter from call the main function */
hdrl_parameter *h_par = hdrl_dar_parameter_create(
      airmass, parang, posang, temp, rhum, pres, wcs);
...
/* Declare and init the input (lambdaIn) and outputs cpl_vector */
cpl_vector *lambdaIn, *xShift, *yShift, *xShiftErr, *yShiftErr;
...
/* Execute main funtion and clean memory*/
hdrl_dar_compute(h_par, lambdaRef, lambdaIn,
      xShift, yShift, xShiftErr, yShiftErr);
hdrl_parameter_delete(h_par);
```

4.11.6 DAR Parameters in a Recipe

In the context of a recipe, all input parameters needed by the differential atmospheric refraction computation may be provided as input parameters. This will create a esorex man-page as follows:

```
$ esorex --man-page hdrldemo_dar

--maxRatioNaNsInImage : Max ratio [%] between bad pixels (NaNs) and good pixels. [30.0]
--lambdaRef-err       : Error [%] of the reference lambda. [0.0]
--parang-err          : Error [%] of the parallactic angle. [0.0]
--posang-err          : Error [%] of the position angle. [0.0]
--temp-err            : Error [%] of the temperature. [0.0]
--rhum-err            : Error [%] of the relative humidity. [0.0]
--pres-err            : Error [%] of the pressure. [0.0]
--airm.method         : Method of approximation to calculate airmass.
                              <HARDIE_62 | YOUNG_IRVINE_67 | YOUNG_94> [HARDIE_62]
--airm.ra-err         : Error [%] of the right ascension (using for calculate airmass). [0.0]
--airm.dec-err        : Error [%] of the declination (using for calculate airmass). [0.0]
--airm.lst-err        : Error [%] of the local sideral time (using for calculate airmass). [0.0]
```

```
--airm.exptime-err    : Error [%] of the integration time (using for calculate airmass). [0.0]
--airm.geolat-err     : Error [%] of the latitude (using for calculate airmass). [0.0]
```

where the parameters listed include:

- `--maxRatioNaNsInImage` allows the selection of valid images. An image having a ratio of NaNs greater than that specified by `--maxRatioNaNsInImage` will be excluded from the analysis.
- `--lambdaRef-err` to `--pres-err` to set errors to the input parameters for DAR, that allow for the computation of error propagation.
- `--airm.method` for the selection of the method to used to approximate the airmass (see section 4.12 for a more detailed description).
- `--airm.ra-err` to `--airm.geolat-err` to set the errors in the airmass calculation, that allow for the computation of error propagation (see section 4.12 for a more detailed description).

4.12 Effective air mass

This section explains how the `hdrl_utils` module can be used to calculate the effective air mass of an observation. It can be calculated using one of three different methods: Hardie (1962), Young & Irvine (1967) and Young (1994).

4.12.1 Algorithm - short description

The algorithm calculates the average airmass for the line-of-sight given by the right ascension (aRA) and the declination (aDEC). The latitude (aLatitude) of the observatory site and the local sidereal time (aLST) at the beginning of the observation has to be given, as well as the duration of the observation, i.e. the exposure time (aExptime). If the exposure time is zero then only one value of airmass is computed, instead of weighting the beginning, middle, and end of the exposure according to Stetson (Stetson P., 1987, PASP 99, 191).

This function can calculate three different kinds of approximations to the airmass as specified in the `hdrl_airmass_approx` parameter[7]:

1. The formula given by Hardie (Hardie 1962, In: "Astronomical Techniques", ed. Hiltner, p. 184) to compute the airmass as a function of zenith distance.
2. The formula of Young and Irvine (Young A. T., Irvine W. M., 1967, Astron. J. 72, 945), where the range of trustworthy airmass outputs is limited to between 1 and 4.
3. The formula of Young (Young A. T., 1994 ApOpt, 33, 1108).

This algorithm can take into account the error propagation if the user enters the relative error of the input parameters in a `hdrl_value` structure {data,error}.

4.12.1.1 Zenith distance

Computes the zenith distance for an observation. It needs the hour angle, declination, and latitude (all in $[rad]$).

```
hdrl_value hdrl_get_zenith_distance(
          hdrl_value aHourAngle,
          hdrl_value aDelta,
          hdrl_value aLatitude)
```

The function computes the cosine of the zenith distance for an observation taken at an hour angle (`aHourAngle`) from the meridian, with valid values in the range extending from $-\pi$ to π, and the declination (`aDelta`) with possible values between $-0.5\ \pi$ and $0.5\ \pi$. The latitude (`aLatitude`) of the observing site takes values in the range of 0 to $2\ \pi$.

[7] You can find these and other collections of interpolative approximations formulas in http://en.wikipedia.org/w/index.php?title=Airmass&oldid=358226579#Interpolative_formulas.

4.12.1.2 Hardie approximation (1962)

Compute the airmass with the `Hardie` approximation. It need the secant of the zenith distance.

```
hdrl_value hdrl_get_airmass_hardie(hdrl_value hvaSecZ)
```

The function uses the approximation given by Hardie (Hardie 1962, In: "Astronomical Techniques", ed. Hiltner, p. 184) to compute the airmass as a function of zenith angle which is given in terms of its secant (`hvaSecZ`). It is supposedy more accurate than Young & Irvine (1967), and usable for zenith angles below 80 degrees.

4.12.1.3 Young & Irvine approximation (1967)

Compute the airmass with the `Young & Irvine` approximation. It need the secant of the zenith distance.

```
hdrl_value hdrl_get_airmass_youngirvine(hdrl_value hvaSecZ)
```

This function uses the approximation given by Young & Irvine (Young A. T., Irvine W. M., 1967, Astron. J. 72, 945) to compute the airmass for a given sec(z) (`hvaSecZ`). This approximation takes into account atmospheric refraction and curvature but is, in principle, only valid at sea level.

4.12.1.4 Young approximation (1994)

Computes the airmass using the `Young` approximation. It needs the cosine of the true zenith distance.

```
hdrl_value hdrl_get_airmass_young(hdrl_value hvaCosZt)
```

This function uses the approximation given by Young (Young A. T., 1994 ApOpt, 33, 1108) to compute the relative optical airmass as a function of true, rather than refracted, zenith angle which is given in terms of its cosine (`hvaCosZt`). It is supposedly more accurate than `Young & Irvine` (1967) but restrictions aren't known.

4.12.2 Functions

The differential atmospheric refraction is computed by the following function:

```
hdrl_value hdrl_utils_airmass(
        hdrl_value          aRA,
        hdrl_value          aDEC,
        hdrl_value          aLST,
        hdrl_value          aExptime,
        hdrl_value          aLatitude,
        hdrl_airmass_approx type)
```

where `type` is the method of airmass approximation to use.

The input parameters of the function are:

- `aRA`: right ascension [deg].
- `aDEC`: declination [deg].
- `aLST`: local sideral time elapsed since siderial midnight [s].
- `aExptime`: integration time [s].
- `aLatitude`: latitude of the observatory site [deg].
- `type`: method of airmass approximation.

The valid values for `type` are:

```
typedef enum {
        HDRL_AIRMASS_APPROX_HARDIE       = 1,
        HDRL_AIRMASS_APPROX_YOUNG_IRVINE = 2,
        HDRL_AIRMASS_APPROX_YOUNG        = 3
} hdrl_airmass_approx;
```

Note that every parameter is an `hdrl_value`, hence an error can be also provided and the routine supports error propagation.

4.12.3 Outputs

The algorithm returns the computed average airmass, or the `hdrl_parameter` value {-1, 0.} when an error is encountered.

4.12.4 Example

An example of an airmass calculation is given here:

```
/* Aproximation methods */
hdrl_airmass_approx type = HDRL_AIRMASS_APPROX_HARDIE;

/* Example of input parameters */
hdrl_value ra       = {122.994945,   0.};
hdrl_value dec      = {74.95304,     0.};
hdrl_value lst      = {25407.072748, 0.};
hdrl_value exptime  = {120.,         0.};
hdrl_value geolat   = {37.2236,      0.};

/* Set variation of error in variables */
```

```
double delta        = 1.;
double deltaRa      = delta/100.;
double deltaDec     = delta/100.;
double deltaLst     = delta/100.;
double deltaExptime = delta/100.;
double deltaGeolat  = delta/100.;

/* Set error propagation variables */
ra.error            = deltaRa      * fabs(ra.data);
dec.error           = deltaDec     * fabs(dec.data);
lst.error           = deltaLst     * fabs(lst.data);
exptime.error       = deltaExptime * fabs(exptime.data);
geolat.error        = deltaGeolat  * fabs(geolat.data);

/* Calcule airmass */
hdrl_value airm = hdrl_utils_airmass(ra, dec, lst, exptime, geolat, type);
```

Chapter 5

Cookbook

The cookbook contains additional help for the pipeline developer in order to use the *High Level Data Reduction Library* in a efficient way. It might also include recipe-level functionality as currently implemented and used in the hdrldemo pipeline.

5.1 Handling large datasets

In order to help handling datasets larger than the available memory and swap space HDRL provides an experimental object `hdrl_buffer` which can be used to allocate disk backed memory maps.

It provides following functions:

- `hdrl_buffer_new()`: create a new buffer object
- `hdrl_buffer_allocate(size_t n)`: allocate `n` bytes of memory
- `hdrl_buffer_free(void *p)`: free memory pointed to by `p`, depending on the position in the memory pool it might actually not free the memory.
- `hdrl_buffer_delete(hdrl_buffer *)`: delete the buffer object and all memory it allocated.

It is designed to be used like a memory pool (sometimes also called memory arena or object stack) which allows fast allocation and deletion of many objects but the ability to delete random objects in random order is limited. It is not threadsafe so if it is used in a multithreaded environment each thread must create and manage its own `hdrl_buffer`.

For good performance usage of this memory should be blocked to chunks of the available RAM to avoid unnecessary disk IO.

Note that on 32 bit machines one is still limited to 2GiB of allocations due to the address space limitations, on 64 bit machines one is only limited by the available disk space.

Memory maps can be slower than swap space backed memory (due to higher complexity and the kernel flushing dirty pages to disk periodically). In order to help test the impact the buffer object will use `cpl_malloc` for

memory allocation if the environment variable `HDRL_BUFFER_MALLOC` is set. On systems with sufficient amounts of swap space this can be the more efficient mode of operation.

5.1.1 Examples

```
hdrl_buffer * buf = hdrl_buffer_new();
/* allocate 2 GiB of memory */
void * mem = hdrl_buffer_allocate(1ull << 31ull);
void * mem2 = hdrl_buffer_allocate(100);
do_stuff(mem, mem2);
/* free all memory again */
hdrl_buffer_delete(buf);
```

HDRL provides a function `hdrl_image_new_from_buffer` to allocate a `hdrl_image` from a `hdrl_buffer` which can be treated like a normal image, including deletion with `hdrl_image_delete` and creation of views.

```
hdrl_buffer * buf = hdrl_buffer_new();
hdrl_imagelist * hl = hdrl_imagelist_new()
for (size_t i = 0; i < 1000; i++) {
    hdrl_image * img = hdrl_image_new_from_buffer(2000, 2000, buf);
    hdrl_image_add_scalar(img, 5., 3.);
    hdrl_imagelist_set(hl, img, i);
}
hdrl_image * out; cpl_image * contrib;
hdrl_imagelist_collapse(hl, HDRL_COLLAPSE_MEDIAN, &out, &contrib);

/* This is very inefficient as it will need to load the data from disk
   twice, instead one should loop over the images or use the iterator
   interface to work in RAM sized blocks */
hdrl_imagelist_add_scalar(hl, 5., 3.);
hdrl_imagelist_mul_scalar(hl, 5., 3.);

/* free all memory again, note the imagelist has to be explicitly deleted
   because structure has not been allocated from the buffer */
hdrl_imagelist_delete(hl);
hdrl_buffer_delete(buf);
```

Appendix A

Abbreviations and acronyms

CPL	Common Pipeline Library
HDRL	High-level Data Reduction Library
ESOREX	ESO Recipe Execution Tool
PPS	Pipeline Systems Group
QC	Quality Control
SGDP	Science Grade Data Product
SPD	Science Products Department
USD	User Support Department
VLT	Very Large Telescope

Appendix B

Statistical Estimators

B.1 Arithmetic mean - Arithmetic weighted mean

The arithmetic mean of a sample $x_1, x_2, \ldots, x_N$ is defined as

$$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \,. \tag{B.1}$$

For a random variable X that is normally distributed ($N(\mu, \sigma)$), the sample mean is the most efficient estimator of the population mean μ. By the central limit theorem this statement holds asymptotically for many noise distributions encountered in practice.

If the noise distribution of X is unknown, its variance can be estimated from the sample as

$$\hat{\sigma}^2 = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})^2 \,. \tag{B.2}$$

If X has variance σ then the average (B.1) has variance

$$\sigma^2_{\rm av} = \frac{\sigma^2}{N}. \tag{B.3}$$

This relation can be used in place of Eq. (B.2) if an *a priori* estimate of the noise in the images is available and is the same for all images (homescedastic case).

When each image has its own noise image $\sigma_i^2(x, y)$ (heteroscedastic case), then the weighted mean

$$\overline{x} = \left(\sum \frac{x_i}{\sigma_i^2} \right) \left(\sum \frac{1}{\sigma_i^2} \right)^{-1} \tag{B.4}$$

can be used, giving higher weight to values with lower noise. The variance of the average value at a given pixel is in this case given by

$$\frac{1}{\sigma^2_{\rm av}} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \,. \tag{B.5}$$

While the average is efficient, it is also sensitive to the presence of outliers, i.e. sample values which arise from a process which is not included in the $N(\mu, \sigma)$ model of the background noise. Cosmic ray hits and the like that affect single exposures in the set therefore show up clearly in the stacked image. The average fails in the presence of a single outlier in the sample.

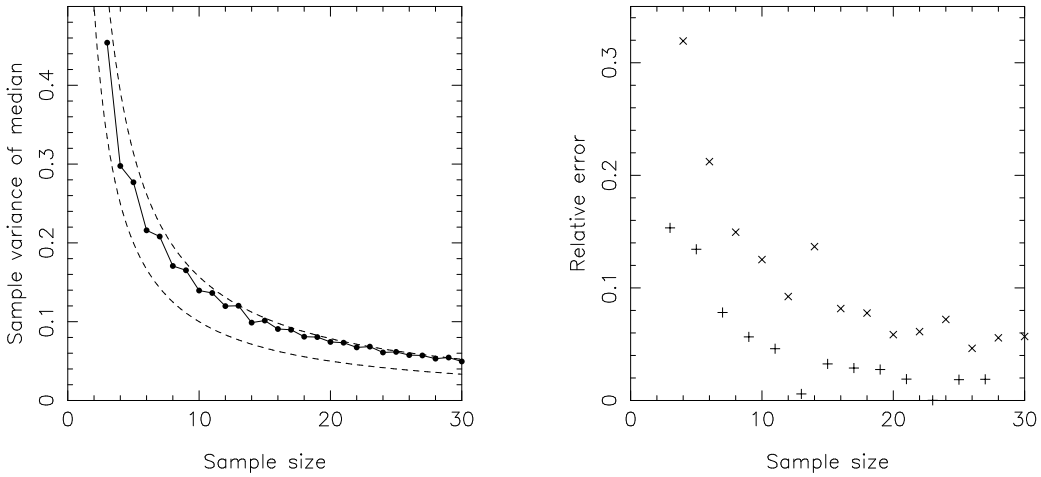

Figure B.2.0.1: Sample variance of the median as a function of sample size (left panel). The lower dashed curve is the expected variance of the arithmetic mean, the upper dashed curve the asymptotic variance of the median. The right panel shows the relative error compared to the asymptotic variance. Crosses are for even sample sizes, pluses for odd sample sizes.

B.2 Median

For the computation of the sample median, the sample $\{x_i,\, i = 1 \ldots N\}$ is first sorted such that $x_1 < x_2 < \cdots < x_N$. The median is then the 50th percentile of the ordered set, i.e.

$$x_{\rm med} = x_{\frac{N+1}{2}} \quad \text{if } N \text{ is odd} \tag{B.6}$$

and

$$x_{\rm med} = \frac{1}{2}(x_{\frac{N}{2}} + x_{\frac{N}{2}+1}) \quad \text{if } N \text{ is even.} \tag{B.7}$$

This definition for the median for even-sized samples ensures that the median is an unbiased estimator of the population mean μ for a Gaussian error distribution or indeed any distribution that is symmetric about its mean.

The sample median has a larger variance than the sample average, i.e. it is less efficient as an estimator of the population mean. It is however robust against outliers in the sample since only the central one or two sample values are actually used for the computation. The median only fails if half or more of the sample values are outliers, i.e. if a given pixel is affected by a cosmic ray hit in half or more of the images in the stacking list.

While the sample distribution of the median (which is not Gaussian) can be written down easily, the computation of its moments and hence its variance is difficult. A scheme to compute it analytically has been described in [3] and exact values for a few sample sizes and a number of parent distributions are given in [4]. However, these are hardly useful for practical purposes. The asymptotic value for the ratio of the variances of mean and median (the *asymptotic relative efficiency* of the median, [1]) in the Gaussian case is

$$\sigma^2_{\rm med} = \frac{\pi}{2}\sigma^2_{\rm av} = \frac{\pi\sigma^2}{2N}. \tag{B.8}$$

In Fig. B.2.0.1, we determine the variance of the median from simulations by drawing for any given sample size 10 000 samples from a standard normal distribution. The asymptotic value is approached from below, which

makes it a conservative choice for an estimate of the median's variance. The relative error, defined as

$$\frac{\pi/2N - \hat{\sigma}^2_{\rm med}}{\pi/2N}\,, \tag{B.9}$$

is plotted in the right hand panel of Fig. B.2.0.1. For even sample sizes, the relative error is significantly larger than for odd sizes because the variance is actually lower. For odd-sized samples, the error is below 10% for $N \geq 7$ ($N \geq 5$ according to [4]), whereas for even sized samples this threshold is not crossed until $N \gtrsim 12$. However, as mentioned above, the asymptotic value in Eq. (B.8) is conservative for any N and it is therefore used.

When each image has its own noise image $\sigma_i^2(x, y)$ (heteroscedastic case), then a weighted median can be defined as the point where cumulative weights is equal to $1/2$. Specifically, the weighted median of an ordered sample of N values x_i with weights w_i is

$$x_{\rm wmed} = \begin{cases} \dfrac{x_j + x_{j+1}}{2}, & \text{if} \quad \sum_{i=1}^{j} w_i = 0.5 \\ x_j, & \text{if} \quad \sum_{i=1}^{j} w_i > 0.5 \quad \text{and} \quad \sum_{i=1}^{j-1} w_i < 0.5 \end{cases} \tag{B.10}$$

B.3 Kappa-sigma clipping

In the min-max rejection algorithm, a fixed number of sample values are rejected, regardless whether they can be identified as outliers or not. If one wants to retain all "good" sample values and only reject true outliers, one has to employ an adaptive method which compares each sample value to the distribution of the entire sample.

In the $\kappa\sigma$ clipping algorithm, all values that deviate from the mean by more than κ standard deviations are rejected as outliers. Typically, $\kappa = 3$. The mean is usually estimated from the data, as is the standard deviation if no independent error estimate is available. One can introduce an iteration which stops once no more values are rejected.

Standard $\kappa\sigma$ clipping is not very efficient in removing low-significance outliers because the estimates of the mean and the standard deviation are themselves affected by the outliers. Using more robust estimators of location and scale, like the median and the inter-quartile range (IQR) or the median absolute deviation (MAD), improves the situation and makes $\kappa\sigma$ clipping a robust and easy-to-use adaptive outlier-rejection algorithm. The *High Level Data Reduction Library* implementation uses the median and the MAD as robust estimators for the location and scale. For a Gaussian distribution, the standard deviation σ is given in terms of the MAD through

$$\sigma = \mathrm{MAD} \times 1.4826. \tag{B.11}$$

$\kappa\sigma$ clipping has one parameter, the clipping threshold κ. The value of κ is not critical if outliers are expected to lie far from the sample distribution, as is the case for cosmic ray hits. Weaker effects, such as satellite trails, may require fine-tuning of κ.

The functions that make use of $\kappa\sigma$ clipping have a second parameter, e.g. `niter` which specifies the maximum number of iterations to run.

The method might fail if more than half of the input images are affected by outliers at a given pixel because then the estimate of the MAD might be affected and cause outlier rejection to fail. The method also requires a sufficient number of input images to be able to obtain reasonably accurate estimates of the mean and standard deviation.

The variance of the estimator now varies from pixel to pixel depending on the number of values that are rejected as outliers. Since outliers are not drawn from the same distribution as the "good" data values, the cleaned sample is equivalent to a smaller sample drawn from the noise distribution. In the general heteroscedastic case, the variance is therefore estimated as

$$\sigma^2_{\kappa\sigma} = \frac{1}{N^2_{\mathrm{good}}} \sum_{\mathrm{good}} \sigma_i^2 \,. \tag{B.12}$$

Appendix C

Bad Pixel Determination

C.1 Detailed Description of the various Algorithms

C.1.1 Bad-pixel detection on a single image

The `hdrl_bpm_2d` recipe identifies bad pixels on individual images by comparing the value in each pixel with those of the neighbouring pixels, via two methods.

In method 1 the algorithm first smooths the single image by applying a cpl-filter (default: median), then subtracts the smoothed images and derives the bad pixels by scaling the rms with specified thresholds on the residual image.

Method 2 is similar to method 1, but the smoothed image is obtained via a Legendre polynomial fit to the image. More details are given in Section 4.5.1.

We tested the 2 methods on flat fields exposures on sky (both simulated with artificial bad pixels and real exposures). We found the following:

- **Method 1.** The best performances (missed detections = 0%, false positives = 0%, recovered bad pixels = 100%), are obtained by setting the recipe parameters to use large smoothing windows (i.e. `-bpm.smooth_x/y = 49`), relatively high sigma clipping levels (`-bpm.kappa_low/high > 5`), and dilation post-filtering with `-pfx/y = 3`. On the contrary, the number of false detection increases dramatically. Figure C.1.1.1 shows the results on simulated flat fields.
- **Method 2.** The best performances ($> 95\%$ success rate) are obtained with the sigma clipping levels `-bpm.kappa_low/high = 3`, although this generates a false detection rate of 10%. The number of false detections can be lowered by increasing `-bpm.kappa_low/high`, but but the success rate decreases and the number of missed detection increases consequentially. The order of the fitting polynomial surface has no strong impact on the performances. Figure C.1.1.2 shows the results on simulated flat fields.

C.1.2 Bad-pixel detection on a stack of images

The `hdrl_bpm_3d` recipe identifies bad pixels on a stack of images. It compares the value in each pixel of the image i-th, with the same pixel in the other $N - 1$ images. It returns a N bad pixel map cube, with the 3rd axis of dimension N.

It has 3 methods, which are detailed in Section ? In a nutshell:

Method 1 identifies bad pixels using an absolute threshold on the residual images. The threshold depend strongly on the input data, and needs to be fine tuned case by case in order to select the best parameters.

Method 2 identifies bad pixels using a threshold on scaled rms. Method 3 identifies bad pixels by scaling the propagated error of each individual pixel and compares it with the measured residuals.

Method 2 and 3 have been tested on simulated bias and simulated flat fields. Both methods have a success rate $> 99\%$ in identifying the bad pixels. Method 3 tends to miss 50% of the bad pixels if `-bpm.kappa` $\geq$ `10`

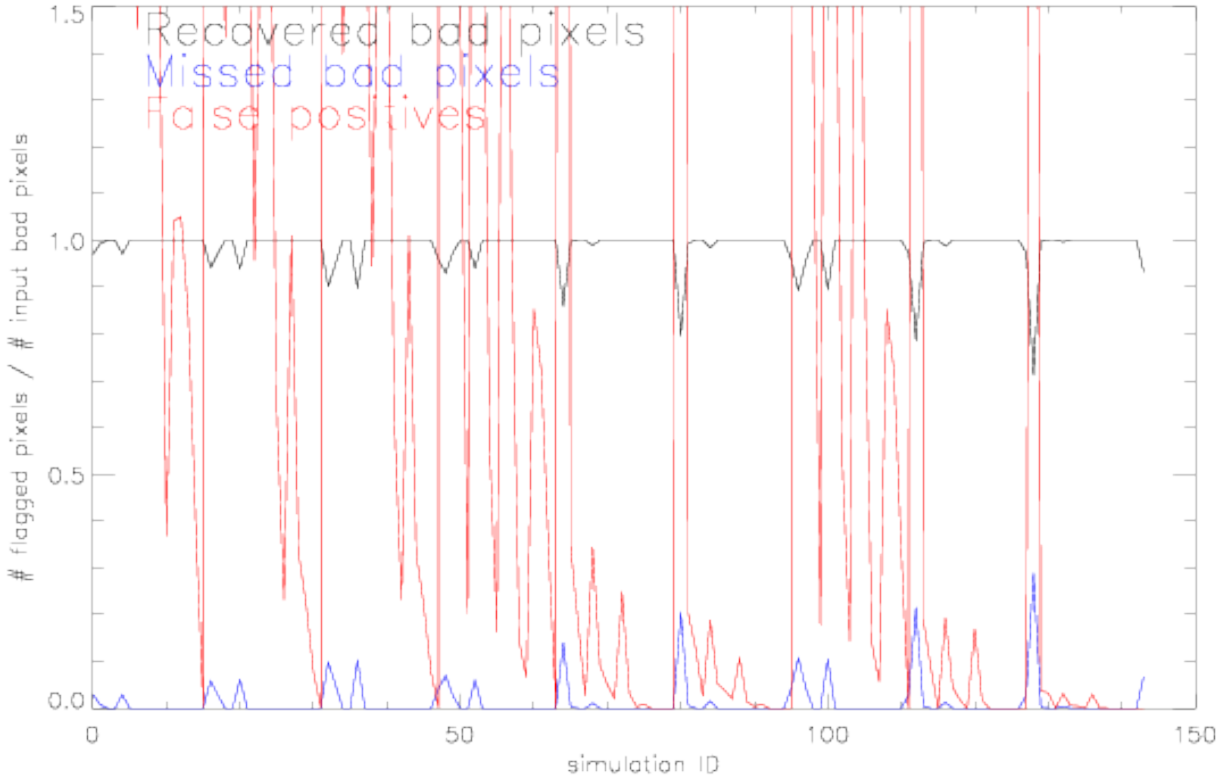

Figure C.1.1.1: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields using method 1 of the `hdrl_bpm_2d` recipe. The best performances are obtained setting `-bpm.smooth_x/y =49`, `-bpm.kappa_low/high > 5`, and `-pfx/y =3`. A large fraction of false detection is found for `-bpm.smooth_ x/y <5` (corresponding to the run ID that have peaks in the red curve in the above figure).

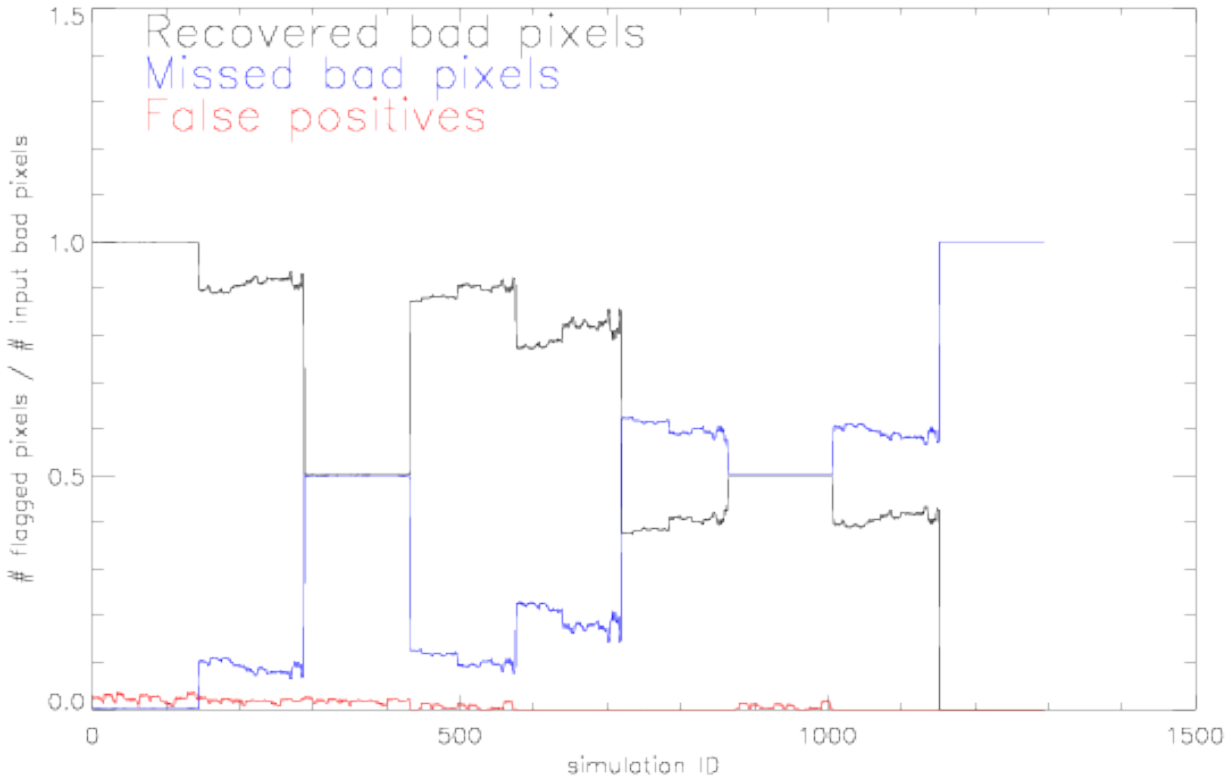

Figure C.1.1.2: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields using method 2 of the `hdrl_bpm_2d` recipe. The best performances are obtained setting `-bpm.kappa_low/high = 3` (run ID < 100). For larger values, the number of detected bad pixels decreases.

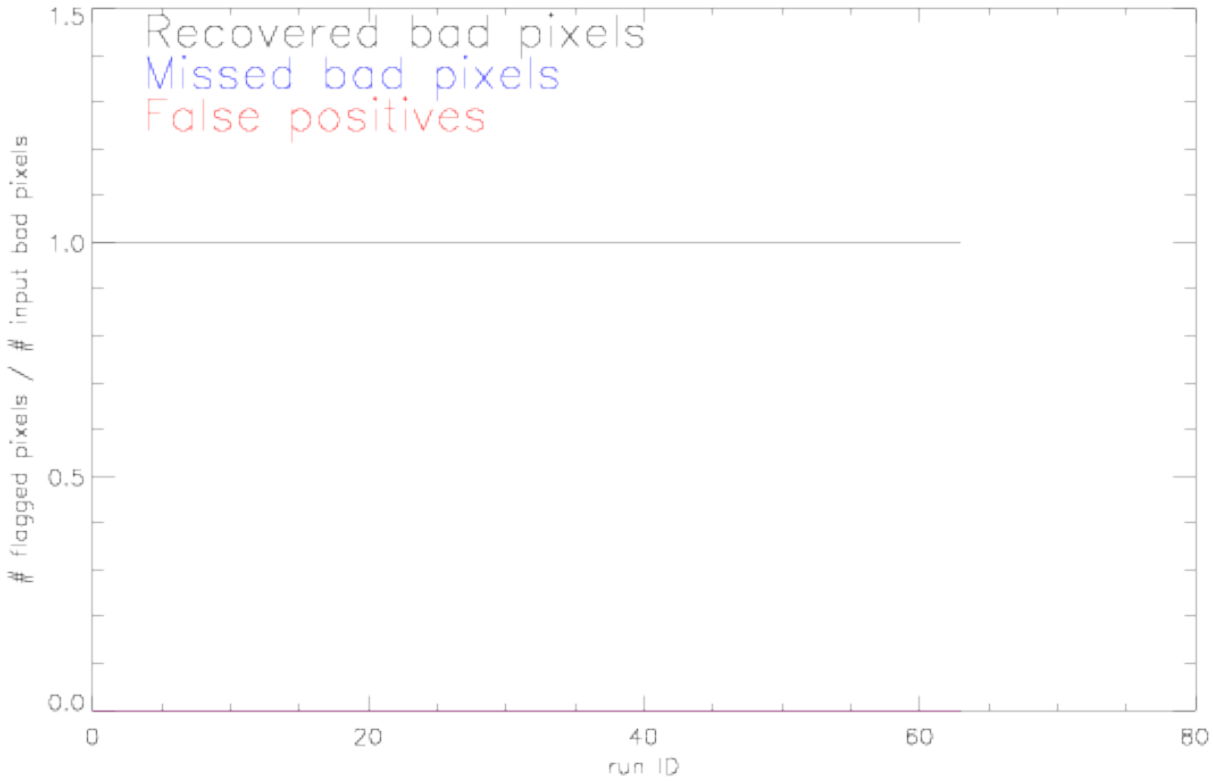

Figure C.1.2.1: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 2 of the `hdrl_bpm_3d` recipe.

.

(method 2 does not have this problem on the same images), whereas method 2 has a $\sim 6\%$ false detection rate on noise images (method 3 does not have this problem on the same images).

Figures C.1.2.1 and C.1.2.2 show the tests on methods 2 and 3 on noiseless simualted stack of flat fields.

C.1.3 Bad-pixel detection on a sequence of images

The `hdrl_bpm_fit` recipe is targeted to the analysis of a set of N frames with different exposure time (e.g. dome flats). At each position (x, y), bad pixels in each frame are identified via a polynomial fit of all the N pixels at position (x, y). The header keyword `EXPTIME` is used as to sampling position of the N pixels along the fit.

Warning: the routine does not disentangle between i) bad pixels or non-linear pixel (i.e. "persistent" bad values of the same pixel across the stack of images) and ii) "accidental" bad pixel values (i.e. the pixel itself is good, but in one frame it has a deviant value because, for example, of a cosmic ray event). If a pixel in the sequence of images is flagged in one frame (e.g. a cosmic ray hit it) but it is not on the other frames, it is flagged in the final mask.

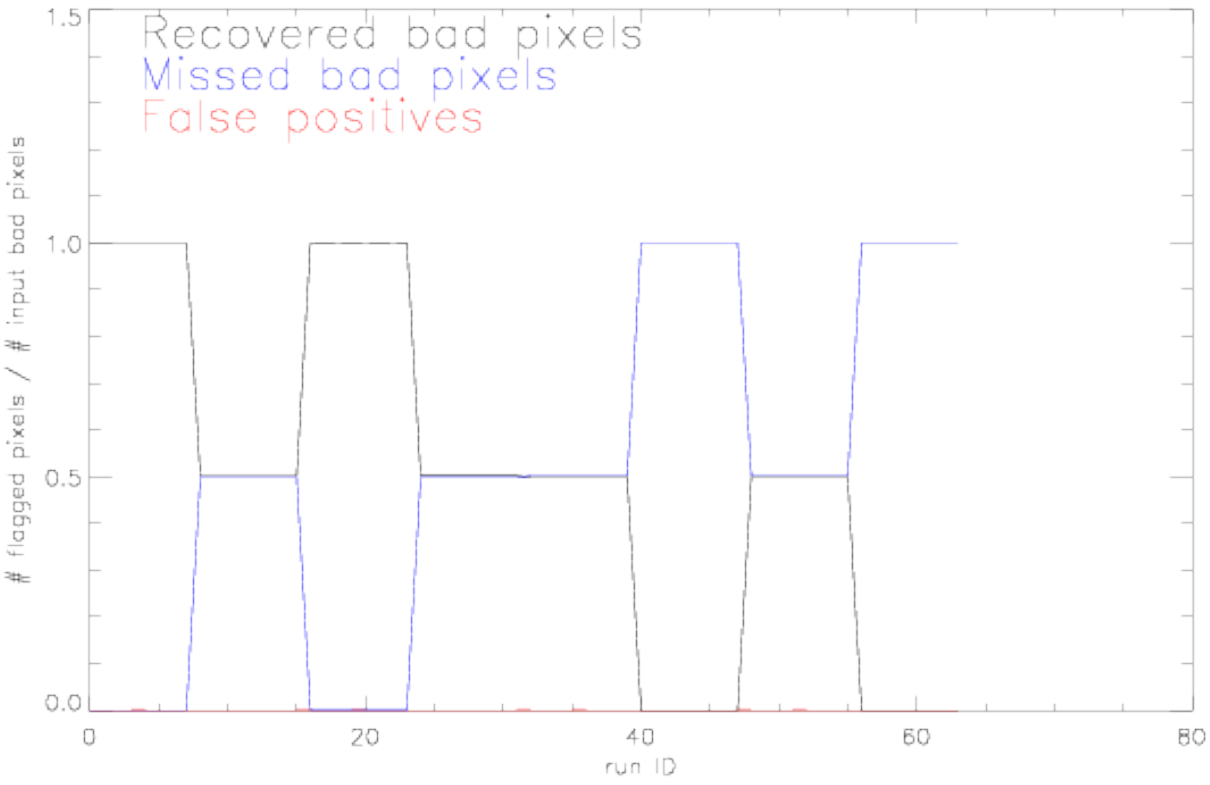

Figure C.1.2.2: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 3 of the `hdrl_bpm_3d` recipe. 50% of input bad pixels are missed if `-bpm.kappa_low/high > 10`.

It is therefore advisable to remove cosmic rays on individual frames, before running `hdrl_bpm_fit` on a sequence of images.

The `hdrl_bpm_fit` recipe has 4 methods:

- **Method 1.** (`-abs-rchi2-low/-high`): This method marks pixels as bad by using the absolute reduced chi squared (low/high) threshold: pixels with values below/above this threshold are marked as bad. This method needs to be fine-tuned for each dataset, because it refers to absolute chi2 values.
- **Method 2.** (`-rel-rchi2-low/-high`): This method marks pixels as bad by using the relative reduced-chi-squared (low/high) threshold: pixels with values below/above this threshold times measured-rms are marked as bad.
- **Method 3.** (`-rel-coef-low/-high`): This method marks pixels as bad by using the relative coefficient (low/high) threshold: pixels with values below/above this threshold times measured-rms are marked as bad.
- **Method 4.** (`-pval`): This method uses the p-value (between 0% and 100%) to discriminate between good and bad pixels. Fits with a p-value below the threshold are considered to be bad pixels.

Figures C.1.3.1, C.1.3.2, and C.1.3.3 show the results of tests performed on a sequence of simulated flat fields. Method 2 and 3 tend to miss a large fraction of bad pixels for `-rel-rchi2` and `-rel-coef` larger than 1 (which correspond to the first run ID in the mentioned figures). Method 4 gives the best performances, independently of the input `-pval` value.

Driven by robustness of method 4, we tested it on a series of real flat fields obtained with FORS2 and CRIRES. The test is aimed to confirm that the number of detected bad pixels is nearly independent of the adopted `-pval` in `hdrl_bpm_fit`. Figure C.1.3.4 indicates indeed that this is the case: the number of detected bad pixels does not vary significantly in the range $0.001 <$ `-pval` < 0.1. Therefore, the defaul value `-pval = 0.001` can be adopted for most applications.

C.1.4 Cosmic Ray Detection

The cosmic ray detection algorithm is derived from the LACosmic routine ([5]). LACosmic is based on a variation of Laplacian edge detection which uses the sharpness of the edges of cosmic rays to discriminate between legitimate sources and cosmic rays. A detailed description of the algorithm, the default parameters, and applicable data cases are summarized in [5].

In summary, the *hdrldemo_bpm_lacosmic* routine has the following input parameters:

Parameter	Description	Default Values
–ext-r	FITS extension of the RAW.	0
–ext-e	FITS extension of the ERROR.	0
–ext-b	FITS extension or the input BPM.	0
–gain	Gain in [e- / ADU].	2.5

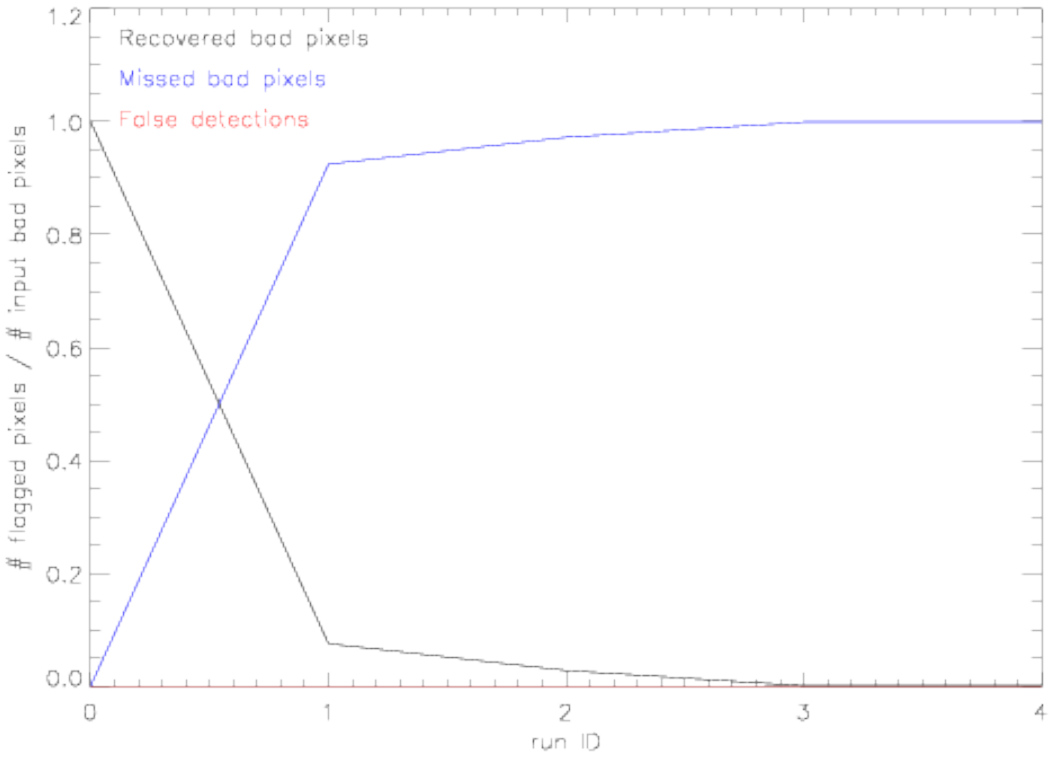

Figure C.1.3.1: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 2 (–rel-rchi2) of the `hdrl_bpm_fit` recipe. The best performance (run ID=0) corresponds to `-rel-rchi2 = 1`.

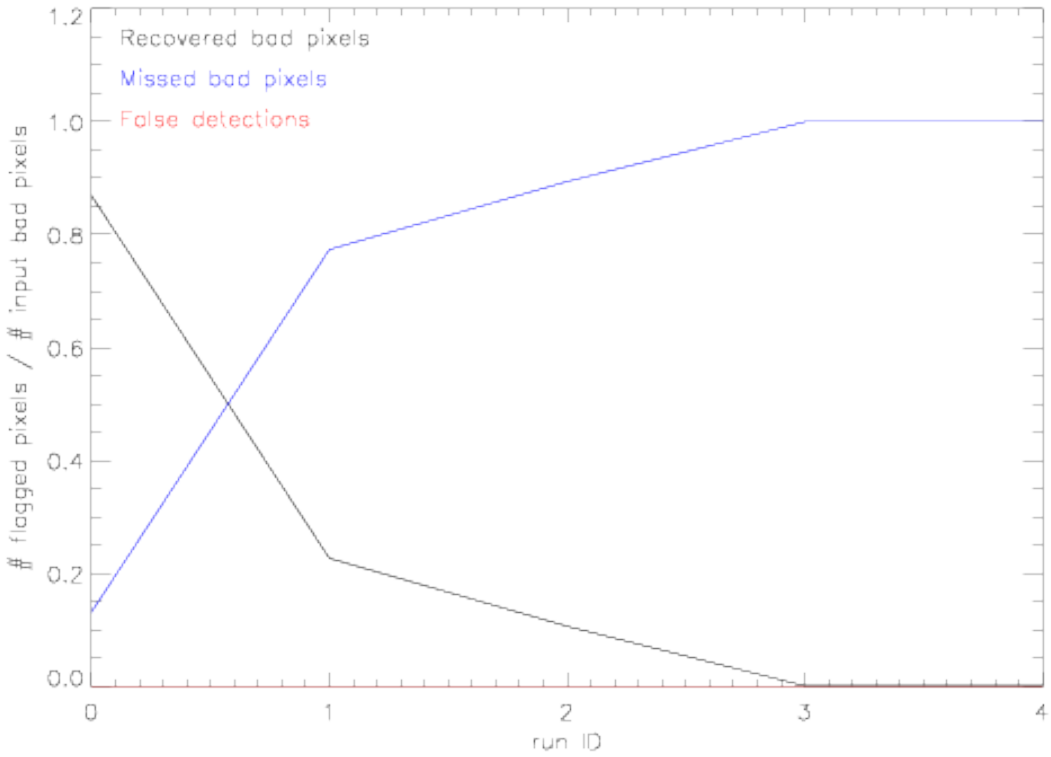

Figure C.1.3.2: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 3 (–rel-coef) of the `hdrl_bpm_fit` recipe. The best performance (run ID=0) corresponds to `-rel-coef = 1`.

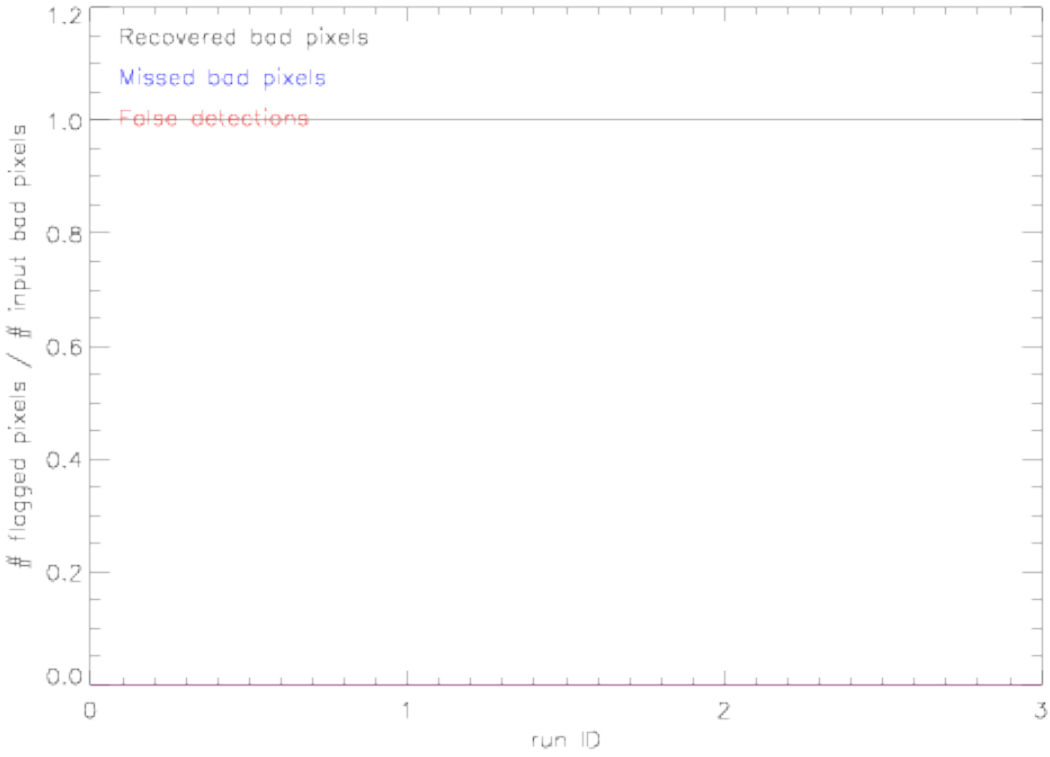

Figure C.1.3.3: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 4 (`-pval`) of the `hdrl_bpm_fit` recipe.

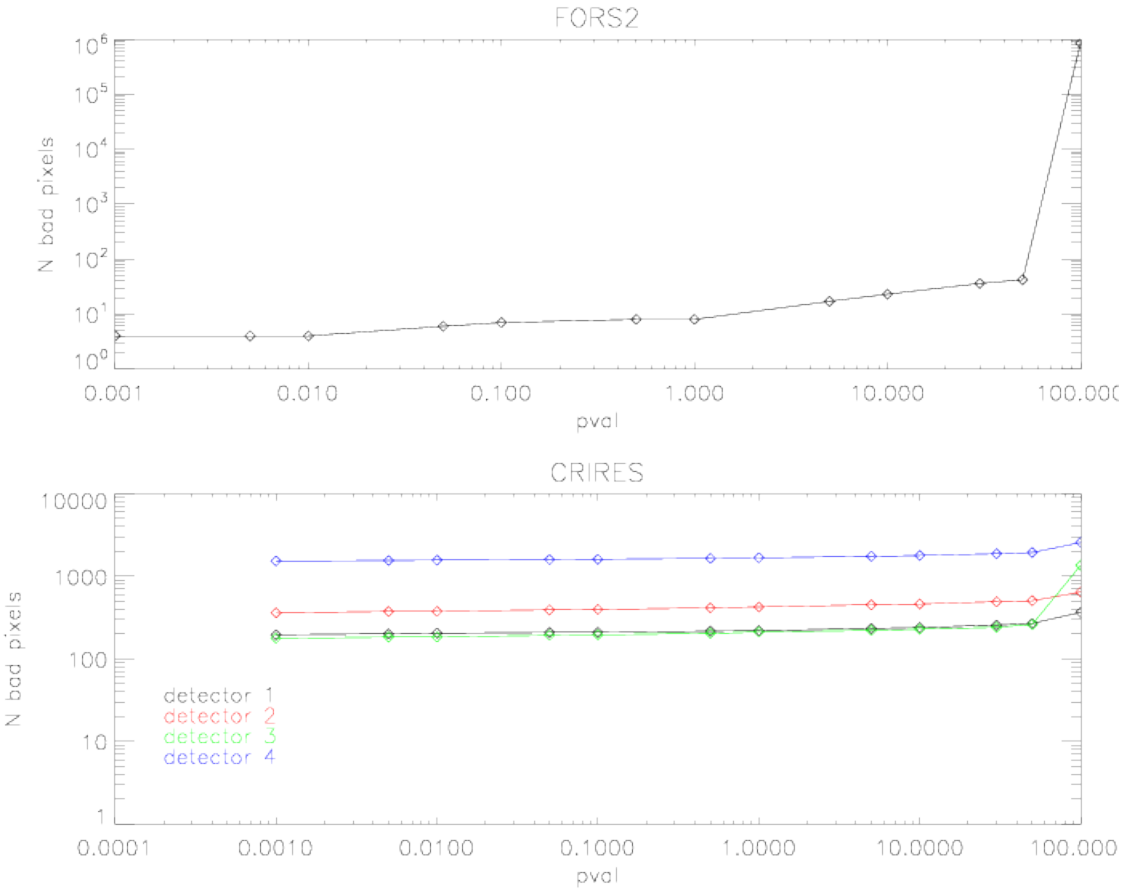

Figure C.1.3.4: Number of flagged bad pixels with `hdrl_bpm_fit`, method 4 as a function of the `-pval` on a series of internal flats for FORS2 (upper panel) and CRIRES (lowe panel). The number of detected bad pixels does not vary significantly within `0.001 < -pval < 0.1`.

–ron	Read-Out Noise.	1.0
–pfx	X Size of the post filtering kernel.	3
–pfy	Y Size of the post filtering kernel.	3
–pfm	Post filtering mode. [closing or dilation]	closing
–region-llx	Lower left x pos. (FITS) defining the region.	1
–region-lly	Lower left y pos. (FITS) defining the region.	1
–region-urx	Upper right x pos. (FITS) defining the region.	0
–region-ury	Upper right y pos. (FITS) defining the region.	0
–lacosmic.sigma_lim	Poisson fluctuation threshold to flag cosmic rays.	4.5
–lacosmic.f_lim	Minimum contrast between the Laplacian image and the fine structure image that a point must have to be flagged as a cosmic ray.	2.0
–lacosmic.max_iter	Max number of iterations.	5

The User can expect to achieve detection efficiencies of about 90% on well-sampled images. This is discussed in detail in section C.1.5.

A good strategy is to run the routine on a typical image using the default parameter values, and then to load the input image and the resultant cosmic ray detection mask into a display and blink the images. By slightly varying the parameters *lacosmic.sigma_lim* and *lacosmic.f_lim* the User can balance the optimal detection efficiency versus spurious cosmic ray detections.

The detection efficiency and the number of spurious cosmic ray detections can be expected to worsen in data that is under-sampled. This may also be true for long slit spectra in which the source spectrum is very sharp.

C.1.5 Testing the Cosmic Ray Detection

The defining parameters of any cosmic ray detection algorithm is its detection efficiency (the ratio of the number of cosmic rays found by the routine divided by the total number of cosmic rays), the number of spurious detections, and how the total number of cosmic rays and the background environment affects the detection. Therefore, all testing was done by creating a known number of cosmic rays of various intensity, size, and shape, and adding these to three different synthetic images: an empty field with a Poisson noise distribution, a sparse star and galaxy field, and a dense globular cluster field. The number of cosmic rays was allowed to vary from 50, affecting 300 pixels and covering 0.007% of the field, up to 10,000 cosmic rays, affecting 61,000 pixels and covering 1.5% of the image. The recipe was then run on $3 \times 37 = 111$ images, stepping between these two extremes of cosmic ray density, and the input cosmic ray frames were compared to the detections achieved with the *hdrldemo bpm lacosmic* routine. The cosmic ray detection rate was a good 89.4% ś 0.4% for the empty and sparse fields, all the way out to unrealistic cosmic ray covering factors of 1.5%. In general, the detection efficiency in the very dense images was, on average, only about 0.3% worse at a level of 89.1% $\pm$ 0.4%. The fraction of spurious cosmic ray detections are at a level of about 6.8% ś 0.3% over the full range of cosmic ray densities.

As is evident in figure C.1.5.1, the cosmic ray detection rate is at a level of 89.4% $\pm$ 0.4% for the empty and sparse fields, all the way out to the extremely high cosmic ray covering factors of 1.5%. There is no significant difference in the detection efficiency for the empty images and the sparse fields. For the very dense images the

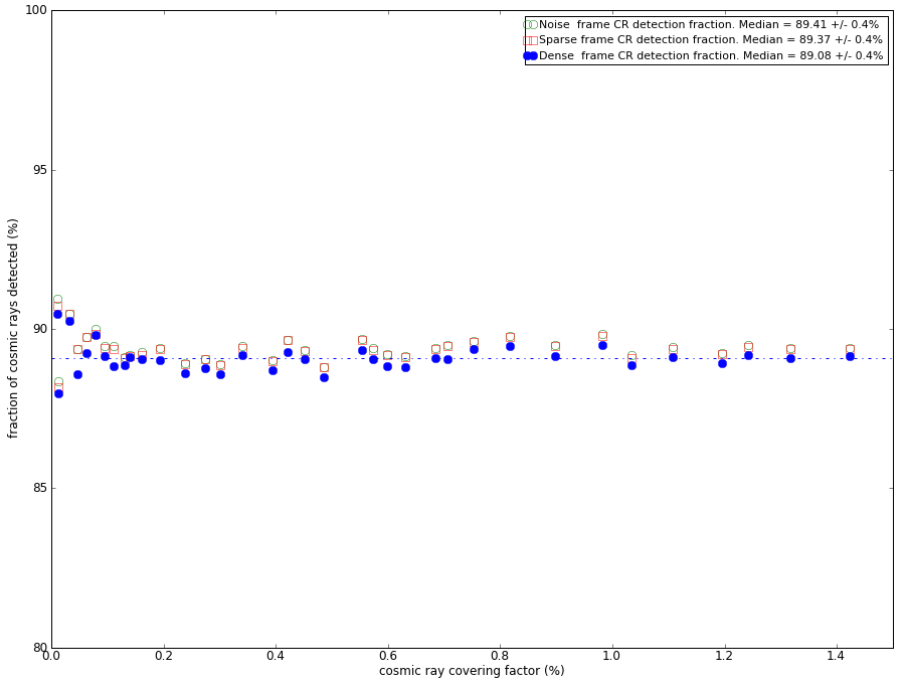

Figure C.1.5.1: The detection efficiency of *hdrldemo bpm lacosmic* versus the cosmic ray covering factor (in percent). The efficiency (given in percent) is measured from the number of cosmic rays detected by *hdrldemo bpm lacosmic* when applied to the pure noise frame (open green circles), the sparsely populated frame (red squares), and the dense frame (solid blue circles). The average detection rates for the pure background and sparse frames are very similar at 89.4% $\pm$ 0.4%, while the detection rates in the dense globular cluster frame is always slightly lower at an average of 89.1% $\pm$ 0.4%. The detection efficiency remains remarkably constant from cosmic ray covering factors of about 0.1% to 1.4% (i.e. from 4,000 to 61,000 affected pixels).

detection efficiency is not notably worse and is, on average, at a level of 89.1% $\pm$ 0.4%. The difference between the dense fields and the sparse and empty fields is roughly constant at about 0.3%.

Beyond a cosmic ray covering factor of 0.1%, however, the detection efficiency remains remarkably constant all the way to the most densely populated cosmic ray fields. In this range of covering factors, the number of affected cosmic ray pixels goes from 4,000 to 61,000.

The inverse of figure C.1.5.1 is given by the fraction of input cosmic rays not detected (see figure C.1.5.2). The total count of undetected cosmic rays (figure C.1.5.2, bottom plot) is made independently of the number of detections and is, therefore, included. Again, for the full range of cosmic ray covering factors, the non-detection rate is about 10.6% $\pm$ 0.4 for the empty and sparse fields, and 11.0% $\pm$ 0.4% for the very dense field.

Finally, a count of cosmic rays was made that were detected by *hdrldemo bpm lacosmic*, but that do not exist in the original input image (see figure C.1.5.3). The fraction of spurious cosmic ray detections are large for very low covering factors, but this is simply due to the small total numbers of cosmic rays. Beyond a covering factor of about 0.2%, the fraction of spurious cosmic ray detections settles down to a value of about 6.8% $\pm$ 0.3%. Within this regime, the numbers of false-positive detections are virtually indistinguishable between the empty, the sparse, and the dense fields.

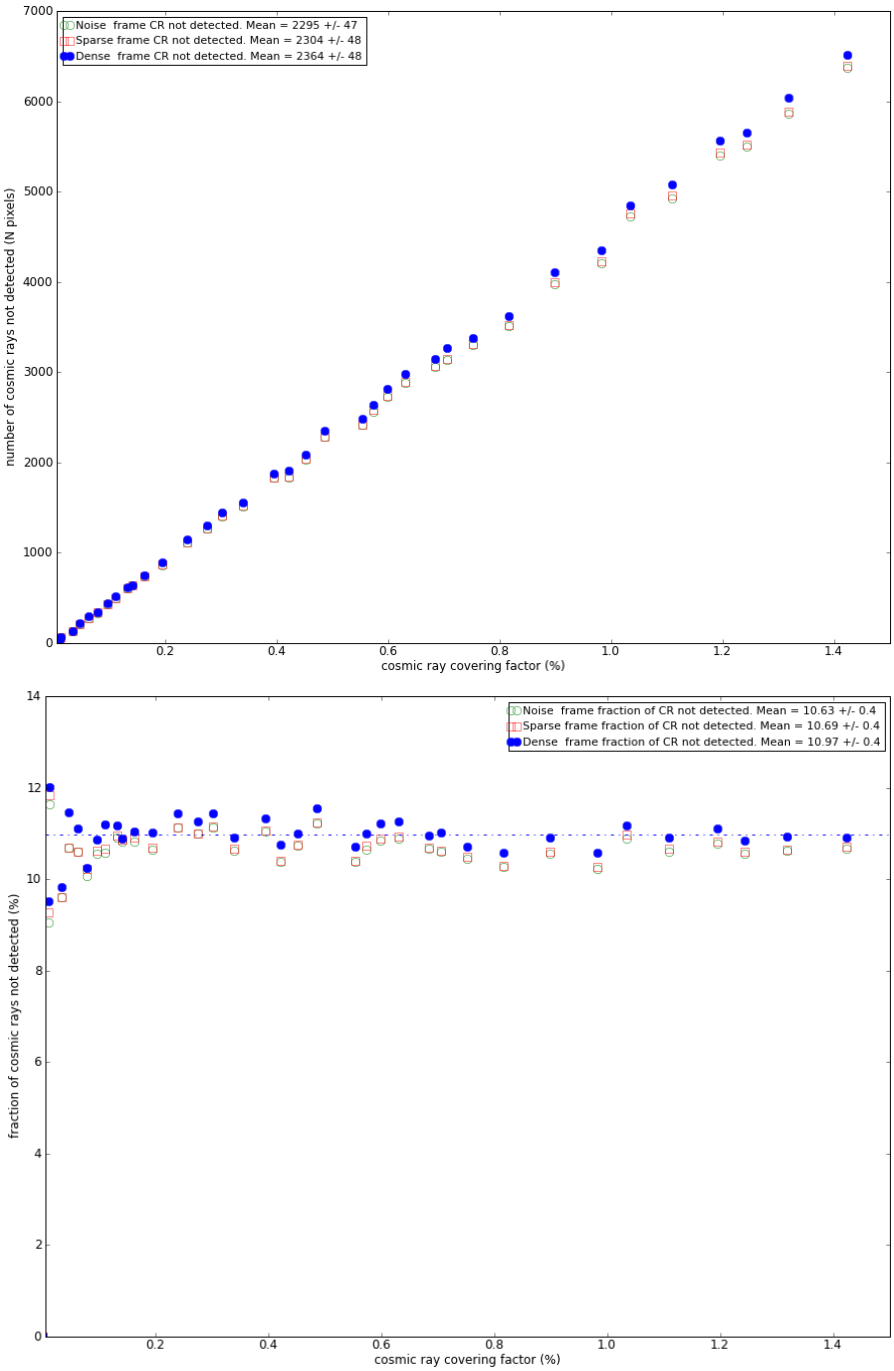

Figure C.1.5.2: The number of cosmic rays **not** detected (top panel) and the non-detection fraction (bottom panel) versus the cosmic ray covering factor (given in percent).

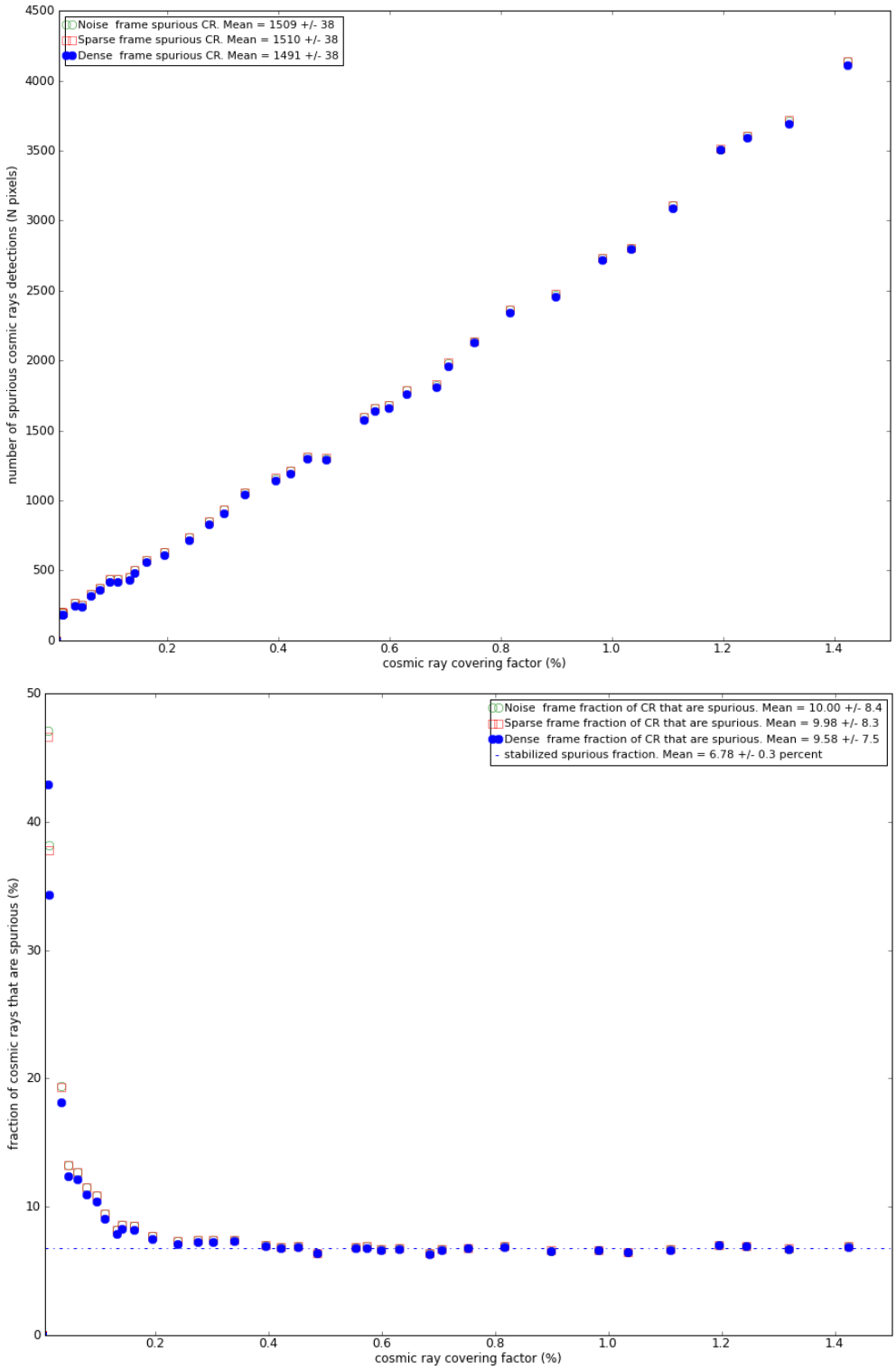

Figure C.1.5.3: The number of spurious cosmic ray detections (top panel) and the spurious fraction (bottom panel) versus the cosmic ray covering factor (given in percent).

Appendix D

Masterfringe Computation and Removal

D.1 Detailed Description of the de-fringing Algorithm

TO BE WRITTEN (SDPG)

Appendix E

Object Catalogue Generation

E.1 Detailed Description of the Object Catalogue Generation Algorithm

TO BE WRITTEN (SDPG)

E.2 Detailed Description of the Object Catalogue Table

TO BE WRITTEN (SDPG)

Appendix F

Efficiency Calculation

F.1 Detailed Description of the Efficiency Computation

TO BE WRITTEN (SDPG)

Appendix G

Response Calculation

G.1 Detailed Description of the Response Computation

TO BE WRITTEN (SDPG)

Appendix H

Differential Atmospheric Refraction

H.1 Introduction

The effects of atmospheric refraction can be readily seen in any spectrograph or IFU. Atmospheric refraction will displace a source, or its spectrum, by an amount that is dependent on the source wavelength and the angular distance of the source from the zenith. This effect is due to the stratified density structure of our atmosphere, and the displacement will be toward the zenith and will be largest for shorter wavelengths. Because of this latter attribute, differential atmospheric refraction is not generally associated with infrared instruments. However, it can be readily seen in SINFONI data cubes observed with the largest wavelength coverage (H+K) and in the smallest pixel scales (25 mas). Here, the shift can be as large as 6 pixels from the beginning of the data cube to its end (see Figures H.3.1.1 and H.3.1.2).

This module uses an analytical approach to compute the expected differential refraction as a function of wavelength, zenith angle, and the refractive index of air which, in turn, depends on temperature, pressure, and water vapour pressure.

The *hdrl_dar* routines require the following inputs (all of which are, generally, available in the input data headers):

- the ambient atmospheric parameters: temperature, pressure, and humidity as contained in the environmental keyword headers: TEL.AMBI.TEMP, TEL.AMBI.PRES.START/END, and TEL.AMBI.RHUM, respectively.
- the instrument rotation angle on the sky
- the parallactic angle of instrument
- and, the world-coordinate system (WCS)

H.2 Testing HDRL DAR

The goal of this test series is to compare the actual source displacements, as measured in processed SINFONI data cubes, with the displacements predicted by *hdrl_dar*.

SINFONI standard stars were selected for this comparison as they tend to be bright, display some flux for the full wavelength range covered, and are point sources. The standard star data was selected from the ESO archive for the years 2014, 2015, and 2016. Furthermore, this data was selected to cover the J and $H+K$-bands as they are most susceptible to atmospheric refraction; the former because it is the bluest filter, and the latter because it covers the largest span in wavelength. Data from all three SINFONI image scales was used. In order to test the full range of parameters relevant to atmospheric refraction, an effort was made to ensure that the data covered a significant range of instrument rotation angles, airmasses, air pressures, temperatures, and relative humidities (see Table H.2.0.0).

Table H.2.0.0: Parameter Ranges Covered by SINFONI Data

SINFONI Cube Parameter	**Range of Parameters**	
	Minimum	**Maximum**
instrumental rotation angle [degrees]	-230	234.2
airmass [degrees]	1.001	2.350
air pressure [HPa]	739.5	747.8
temperature [°C]	3.4	20.0
relative humidity [%]	3.0	63.0

The standard star data was processed using the SINFONI pipeline version 3.0.0 running as a Reflex workflow. All processing parameters were left in their default values. In total, 754 SINFONI data cube products were created in this processing.

H.3 Testing the *hdrl_dar* Differential Atmospheric Refraction Routines with SINFONI Data

H.3.1 I. Method and Qualitative Results

For each SINFONI data cube, a source centroid for each cube plane (omitting the first and last 100 planes due to higher noise levels) was computed, in order to measure the actual source shifts due to atmospheric refraction. This is done using the source detection catalogue routines (*hdrl_catalogue*) adapted from CASU (Madsen, G., HAWK-I Pipeline User Manual, February 19, 2016. Issue 1.0.). For each of the 754 data cubes analysed, plots were created mapping the measured source centroids (X_c and Y_c) as a function of wavelength through the data cube. For each of these source traces, the shift values computed by the *hdrl_dar* output tables were applied to the centroids to correct the differential atmospheric refraction. A typical example of such a trace is shown in Figure H.3.1.1. A comparison of the median-collapsed cube images before and after being corrected for atmospheric refraction (at an integer shift level) is shown in Figure H.3.1.2. In both plots, a significant qualitative improvement is apparent.

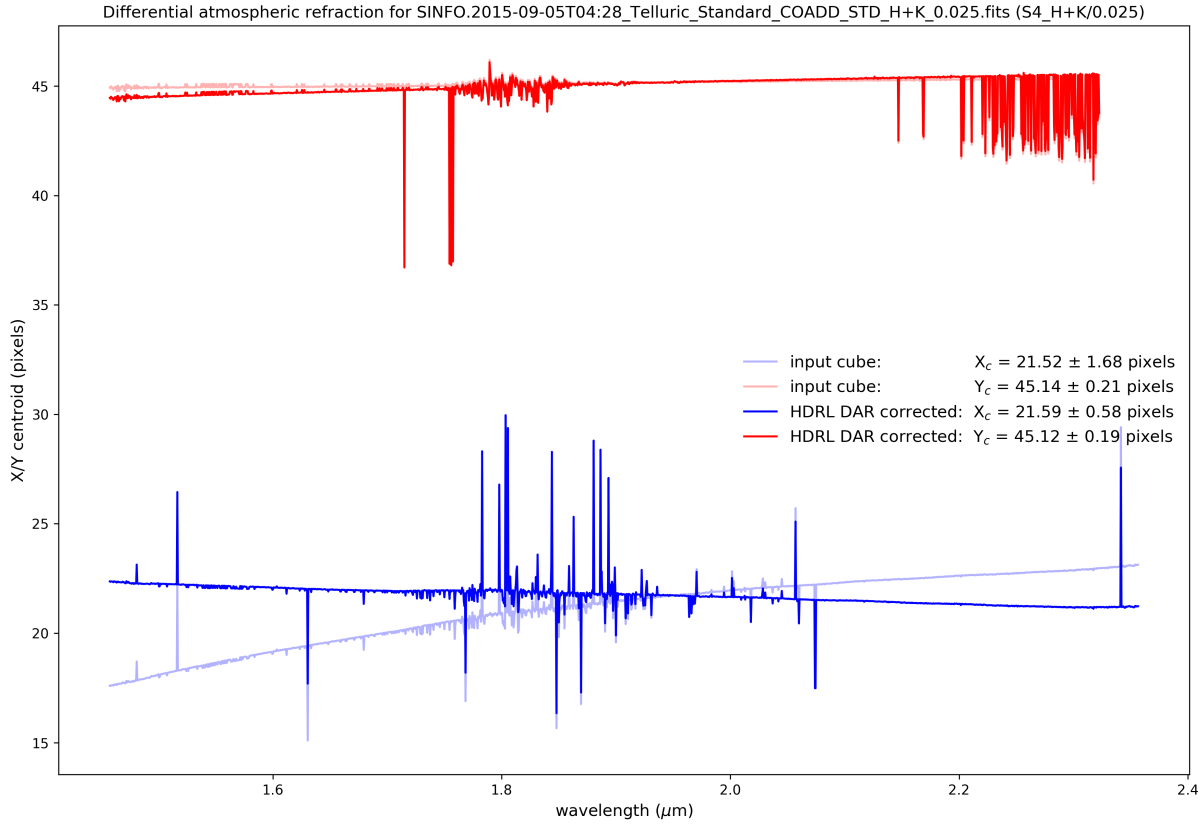

Figure H.3.1.1: A line trace of the source centroid position as a function of wavelength for the SINFONI data cube `SINFO.2015-09-05T04:28_Telluric_Standard_COADD_STD_H+K_0.025.fits`. The source centroid is computed through the data cube using the catalogue source-detection module available in *hdrl_catalogue*. The blue and red lines are the x-axis (X_c) and y-axis centroids (Y_c), respectively. The faded lines are the raw input data centroids, while the dark lines are the centroids after being corrected for differential atmospheric refraction by *hdrl_dar*. The raw data cube has a shift in the source centroid, over its full wavelength range, of $\Delta X_c = 6.9$ and $\Delta Y_c = 0.2$ pixels.

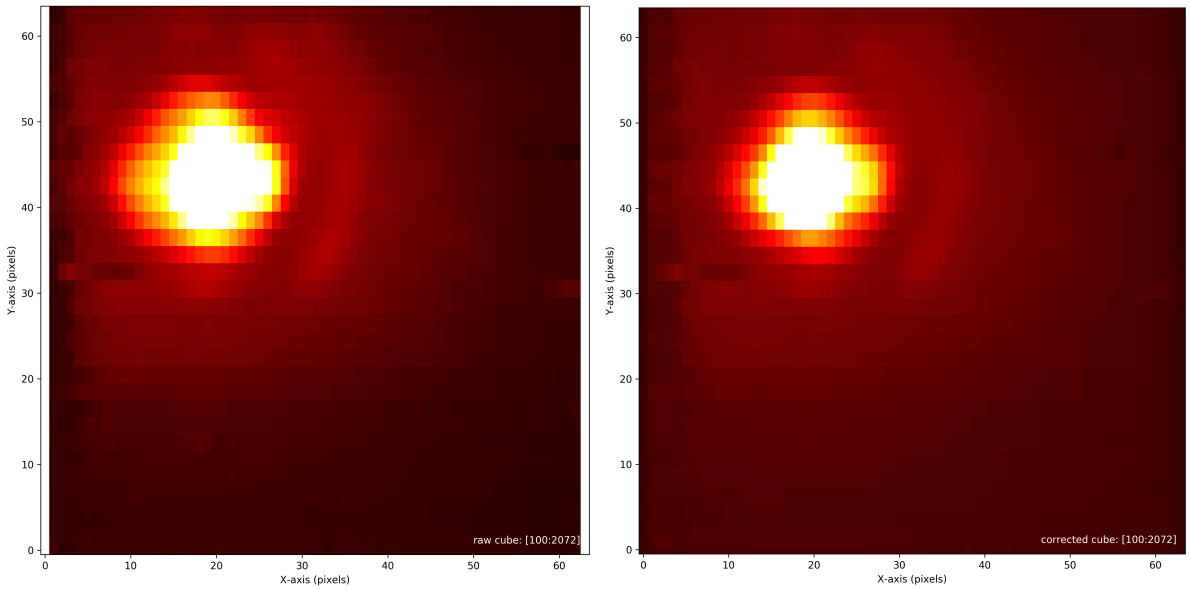

Figure H.3.1.2: **Left Panel:** A median collapse of the same raw data cube shown as a trace in Figure H.3.1.1.
Right Panel: The same median collapsed cube, integer-pixel-corrected for the differential atmospheric refraction. The improvement between the raw and corrected data cubes is visually apparent. A comparison between the raw and corrected frames reveals an x-axis FWHM of 6.5 and 5.8 pixels, respectively.

H.3.2 II. Quantitative Results

In order to quantify the improvements made by applying the theoretical shifts computed by *hdrl_dar*, we define a global atmospheric refraction based the source position at the end points of each SINFONI data cube (see Figure H.3.2.1). Using the median value of the source centroid over the first 100 cube planes and the last 100 cube planes, we define an overall shift as:

$$\Delta X_c = < X_c >_{100} - < X_c >_{N_{max}-100} \quad \text{(H.1)}$$

$$\Delta Y_c = < Y_c >_{100} - < Y_c >_{N_{max}-100} \quad \text{(H.2)}$$

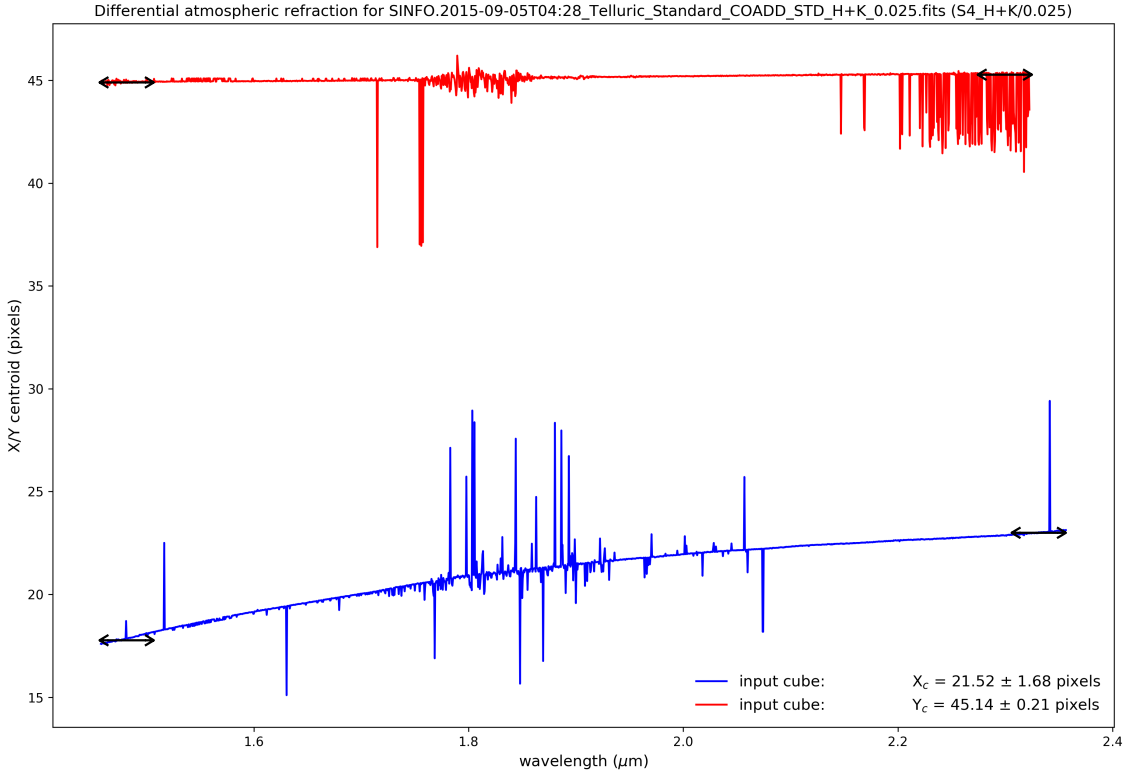

Figure H.3.2.1: In order to measure the global atmospheric refraction offsets in all SINFONI data cubes, the median centroid position is determined over the first 100 and the final 100 cube planes. This is used to define the ΔX_c and ΔY_c.

This global shift is defined for both the measured source centroids and for the theoretically predicted shifts. For all 754 SINFONI data cubes, we find a very good match between the measured data offsets and those predicted by *hdrl_dar*:

$$\Delta X_c(\text{data}) - \Delta X_c(\text{theory}) = 0.29 \pm 0.63 \text{ pixels} \quad \text{(H.3)}$$

$$\Delta Y_c(\text{data}) - \Delta Y_c(\text{theory}) = 0.02 \pm 0.19 \text{ pixels} \quad \text{(H.4)}$$

It should be noted that the reconstructed SINFONI data cubes have rectangular pixels, with the Y-axis pixels being twice as large as those in the X-axis. This explains the difference in standard deviation measured in the offsets of equations (3) and (4). There appears to be a slight under-correction in the ΔX_c of about $\frac{1}{3}$ of a pixel. However, due to the difficulty measuring the source centroids due to object faintness and/or large background noise, this fraction of a pixel offset is not significant.

The differences between the global offsets measured in the SINFONI data cubes and the offsets predicted by *hdrl_dar*, can be summarised in a surface density plot displaying ΔX_c(data) - ΔX_c(theory) vs. ΔY_c(data) - ΔY_c(theory). This plot is shown in Figure H.3.2.2.

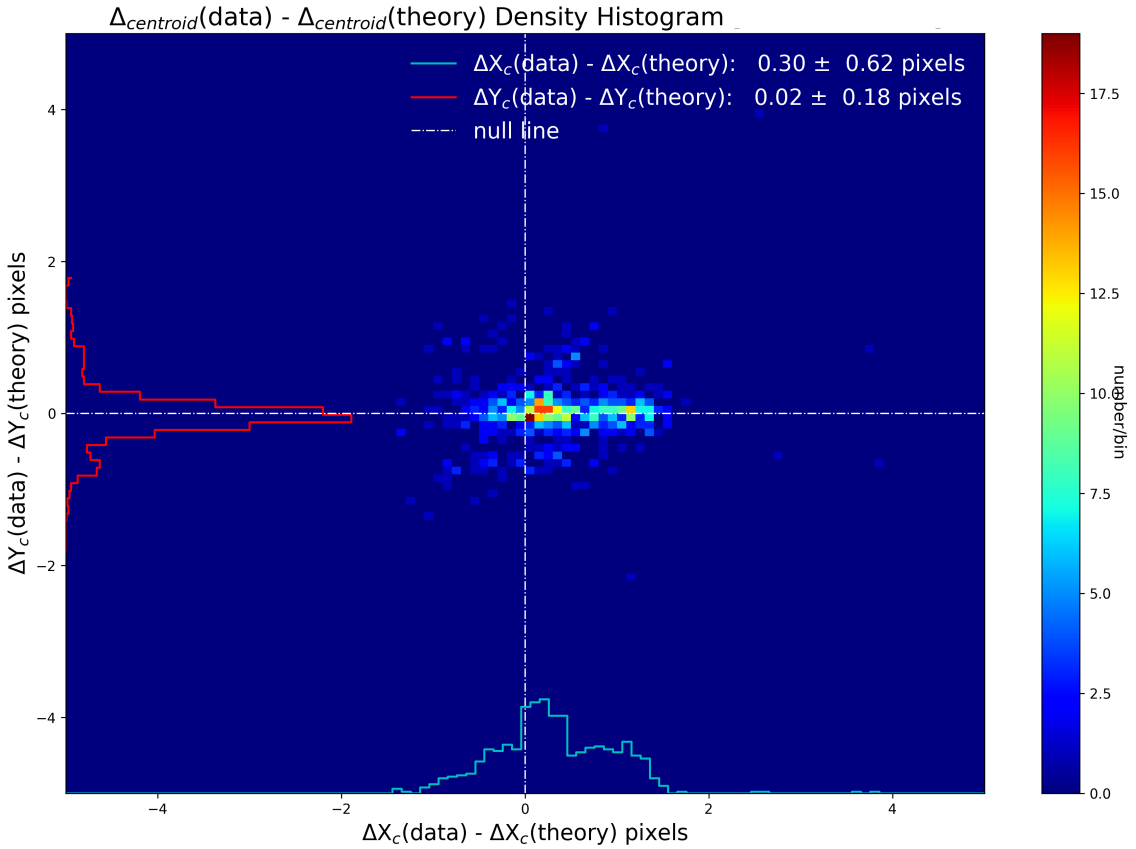

Figure H.3.2.2: Surface density plot of ΔX_c(data) - ΔX_c(theory) vs. ΔY_c(data) - ΔY_c(theory) for the 754 SINFONI data cubes analysed. Note that the differing widths of the ΔX_c and ΔY_c histograms is due to the fact that SINFONI has rectangular pixels in a 1:2 ratio for X:Y.

An alternative measure of the difference between the measured and predicted offsets can be done using the residuals. For each SINFONI data cube, the median centroid position of the standard star was measured ($< X_c >$, $< Y_c >$). Then, for each cube plane the shift predicted by *hdrl_dar* was added to the measured value of the source centroid and subtracted from the median centroid position. Thus, the residuals were computed as:

$$X_c\,\text{residual} = |< X_c(\text{data}) > -[X_c(\text{data}) + X_{\text{shift}}(\text{theory})]| \quad \text{pixels} \tag{H.5}$$

$$Y_c\,\text{residual} = |< Y_c(\text{data}) > -[X_c(\text{data}) + Y_{\text{shift}}(\text{theory})]| \quad \text{pixels} \tag{H.6}$$

A surface density plot of these residuals are shown in Figure H.3.2.3.

$$X_c\,\text{residual} = 0.15 \pm 0.13 \quad \text{pixels} \tag{H.7}$$

$$Y_c\,\text{residual} = 0.04 \pm 0.03 \quad \text{pixels} \tag{H.8}$$

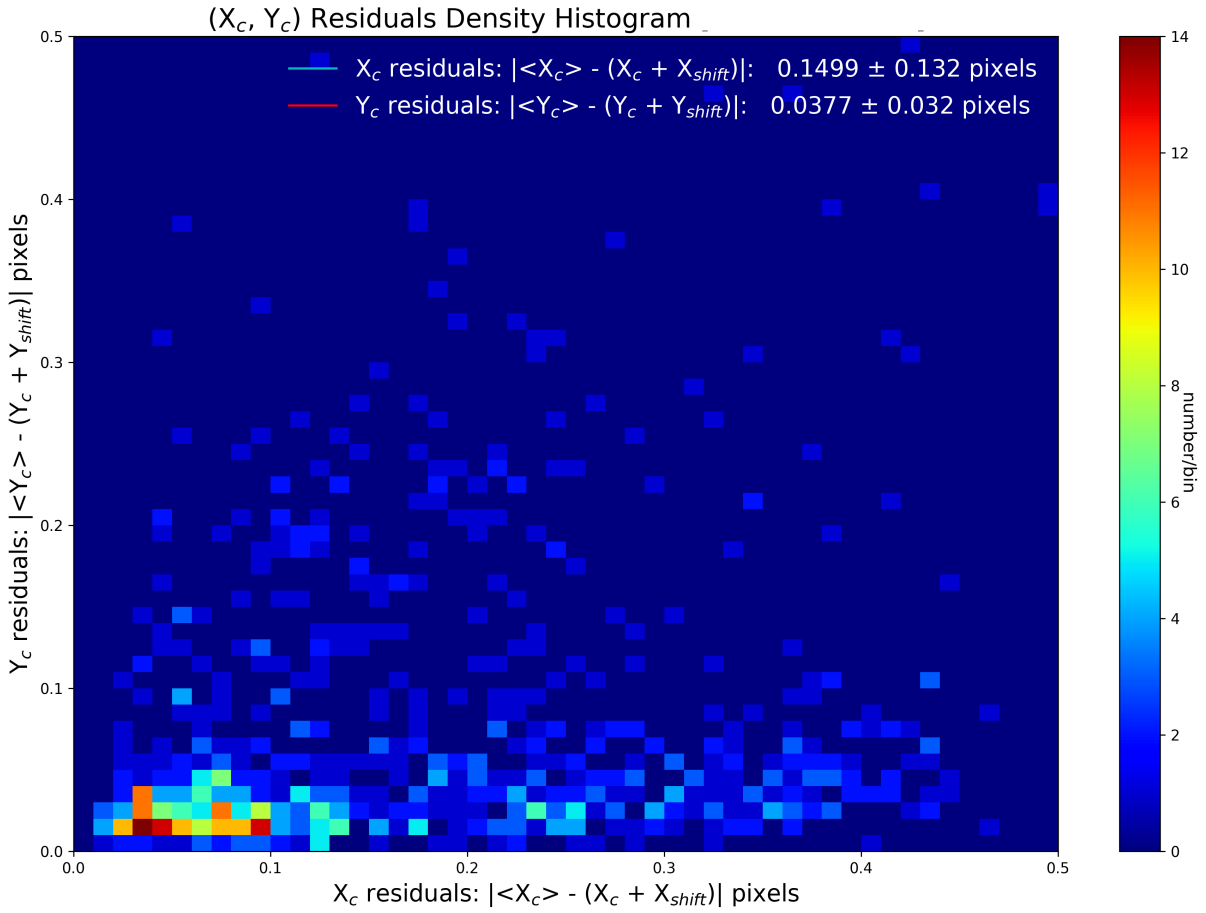

Figure H.3.2.3: Surface density residual plot of $| < X_c(\text{data}) > -[X_c(\text{data}) + X_{\text{shift}}(\text{theory})] |$ vs. $| < Y_c(\text{data}) > -[X_c(\text{data}) + Y_{\text{shift}}(\text{theory})] |$ for the 753 SINFONI data cubes analysed. Note that the differing widths of the ΔX_c and ΔY_c histograms is due to the fact that SINFONI has rectangular pixels in a 1:2 ratio for X:Y.

H.3.3 The Strehl Ratios in *hdrl_dar*-Corrected Images

A further test was made of the improvements possible in the image quality when the *hdrl_dar* correction is applied to the SINFONI standard star data cubes. *hdrl_dar* uses its calculated shifts as a function of wavelength, summarised in the results table (`HDRLDEMO_DAR_RESULT`), to apply an integer correction to the source position. This is done as the simplest correction and to avoid having to interpolate.

For each corrected data cube a median collapsed image was create and compared to the median collapsed image of the input data cube (an example of this can be seen in Figure H.3.1.2). For each image, `SExtractor` was run using a detection and analysis threshold of 3σ. The *hdrl_strehl* routine was then run on the same image using a `flux-radius` set to 1.5×FWHM computed by `SExtractor`. The `bkg-radius-low` was set to 2.0×FWHM and the `bkg-radius-high` was set to 3.0×FWHM. This analysis was restricted to the SINFONI standard star data cubes that show the largest atmospheric refraction; namely, the J and $H+K$ filters with the 25 mas pixel scale. The strehl ratio distributions for the 126 uncorrected (raw) and corrected data cubes are shown as histograms in Figure H.3.3.1. The corrected data (green histogram) shows a marked improvement

over the uncorrected data, with both the strehl median and standard deviation being reduced by 26% and 32%, respectively. Since only an integer shift correction was made, this can be considered a lower-limit to the strehl improvement that is possible when applying *hdrl_dar* corrections.

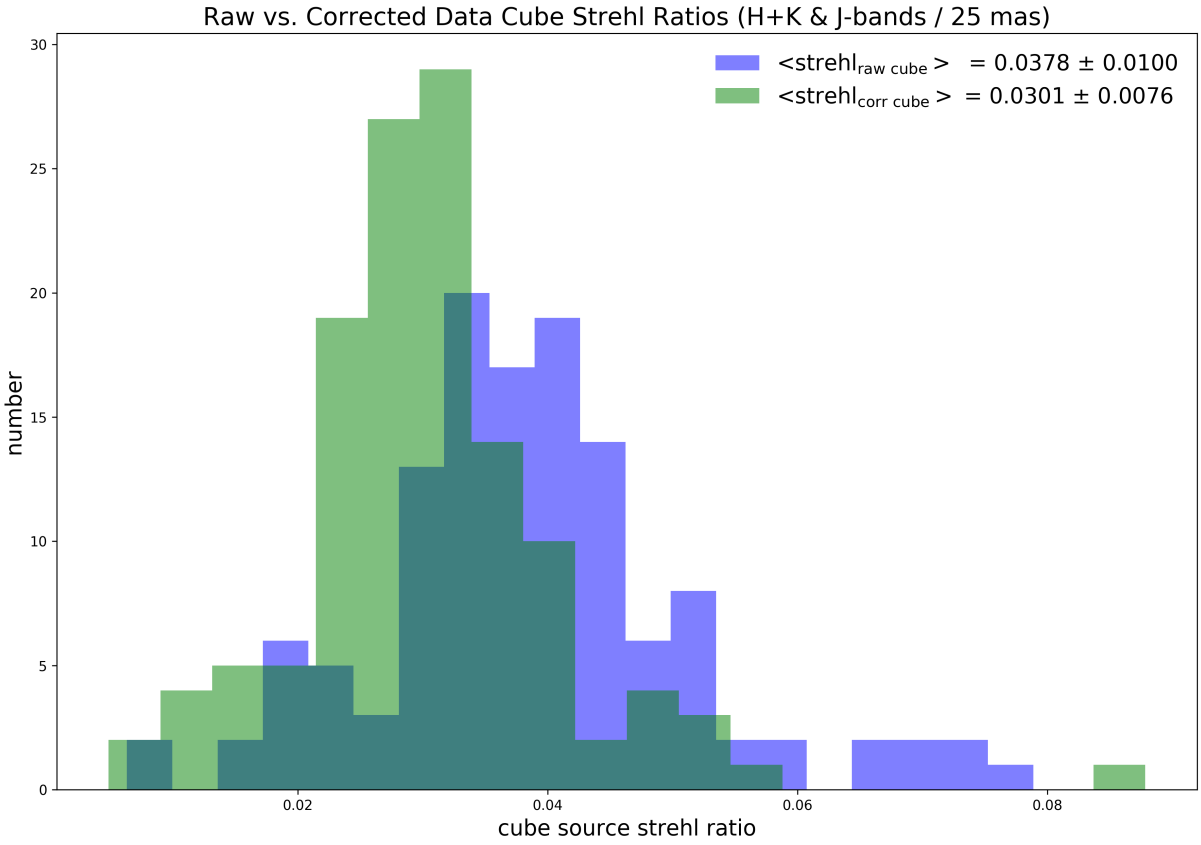

Figure H.3.3.1: A comparison between strehl ratios measured on the median collapsed SINFONI data cubes. The blue histogram shows the strehl ratios of the original data while the green histogram shows the strehl ratios of data corrected for differential atmospheric refraction using *hdrl_dar*. The data is restricted to the 126 SINFONI cubes with 25 mas pixels scale as these have the largest displacements. The *hdrl_dar* correction of the differential atmospheric refraction employed integer pixel shifts only. No interpolation was done. Despite this, the improvement is significant both in terms of the histogram distribution and in the 26% improvement in the median strehl ratio.