



European Organisation for Astronomical Research in the Southern Hemisphere

**Programme:** GEN

**Project/WP:** Science Operation Software Department

# HDRL Pipeline Developer Manual

**Document Number:** ESO-299492

**Document Version:** 7

**Software Version:** 1.6.0

**Document Type:** Manual (MAN)

**Released on:** 2026-06-02

**Document Classification:** ESO internal

Prepared by:

Validated by:

Approved by:

Name



## Authors

Name	Affiliation
The HDRL Team	ESO

## 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 → 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
1.2.0	01/10/2018	4.13, 4.6.5, 4.9	Fixed Pattern Noise, Strehl Ratio, Spectral Efficiency
1.3.0	12/02/2021	4.14 and J	Resampling and Assessment of 2D images and 3D cubes
1.4.0	30/09/2021	4.15, K, and B.4	Limiting magnitude and distribution mode
1.5.0	01/06/2022	4.16, 2.2, and 2.3	Barycentric correction and library dependencies
1.6.0	02/06/2026	2.2.1, 4.8	Release 1.6.0 (catalogue updates, standalone build support)



## Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Motivation and Purpose . . . . .	11
1.2	Stakeholders . . . . .	11
1.3	Scope and references . . . . .	11
<b>2</b>	<b>Software Infrastructure</b>	<b>13</b>
2.1	SVN usage and library integration . . . . .	13
2.2	Releases . . . . .	13
2.2.1	Release version 1.6.0 . . . . .	14
2.2.2	Release version 1.5.0 . . . . .	14
2.2.3	Release version 1.4.0 . . . . .	15
2.2.4	Release version 1.3.0 . . . . .	15
2.2.5	Release version 1.2.0 . . . . .	16
2.2.6	Release version 1.1.0 . . . . .	16
2.2.7	Release version 1.0.0 . . . . .	17
2.2.8	Release version 0.3.0b1 . . . . .	17
2.2.9	Release version 0.2.0 . . . . .	18
2.2.10	Release version 0.1.5 . . . . .	18
2.2.11	Release version 0.1.0 . . . . .	18
2.2.12	Release version 0.0.3 . . . . .	18
2.3	Dependencies . . . . .	18
<b>3</b>	<b>Core Objects</b>	<b>20</b>
3.1	Images . . . . .	20
3.1.1	Example usage . . . . .	20
3.2	Image Lists . . . . .	21
3.2.1	Views . . . . .	21
3.2.1.1	Example . . . . .	21
3.2.2	Collapse Operations . . . . .	22
3.3	Spectrum . . . . .	24
3.3.1	Resampling . . . . .	25
3.3.2	Example usage . . . . .	25



<b>4</b>	<b>Functionalities</b>	<b>27</b>
4.1	Overscan	27
4.1.1	Overscan Computation	27
4.1.1.1	Inputs	28
4.1.1.2	Outputs	28
4.1.1.3	Algorithm	29
4.1.2	Overscan Correction	31
4.1.2.1	Inputs	31
4.1.2.2	Outputs	32
4.1.2.3	Algorithm	32
4.1.3	Examples	33
4.1.3.1	Overscan Computation	33
4.1.3.2	Overscan Correction	34
4.1.4	Overscan Parameters in a Recipe	35
4.2	Bias	36
4.2.1	Algorithm - short description	36
4.2.2	Functions	36
4.2.3	Inputs	36
4.2.4	Outputs	37
4.2.5	Examples	37
4.2.6	Master Bias Parameters in a Recipe	37
4.3	Dark	39
4.3.1	Algorithm - short description	39
4.3.2	Functions	39
4.3.3	Inputs	39
4.3.4	Outputs	39
4.3.5	Examples	39
4.3.6	Master Dark Parameters in a Recipe	40
4.4	Flat	40
4.4.1	High frequency master flatfield	41
4.4.1.1	Algorithm - short description	41
4.4.1.2	Functions	41
4.4.1.3	Inputs	41



4.4.1.4	Outputs	42
4.4.2	Low frequency master flatfield	42
4.4.2.1	Algorithm - short description	43
4.4.2.2	Functions	43
4.4.2.3	Inputs	43
4.4.2.4	Outputs	43
4.4.3	Examples	43
4.4.4	Master Flat Parameters in a Recipe	44
4.5	Bad-pixel detection	46
4.5.1	Bad-pixel detection on a single image	46
4.5.1.1	Algorithm - short description	46
4.5.1.2	Functions	46
4.5.1.3	Inputs	47
4.5.1.4	Outputs	47
4.5.2	Bad-pixel detection on a stack of identical images	47
4.5.2.1	Algorithm - short description	47
4.5.2.2	Functions	48
4.5.2.3	Inputs	48
4.5.2.4	Outputs	48
4.5.3	Bad-pixel detection on a sequence of images	48
4.5.3.1	Algorithm - short description	49
4.5.3.2	Functions	49
4.5.3.3	Inputs	49
4.5.3.4	Outputs	50
4.5.4	Cosmic Ray Hits detection	50
4.5.4.1	Algorithm - short description	50
4.5.4.2	Functions	50
4.5.4.3	Inputs	50
4.5.4.4	Outputs	51
4.5.5	Examples	51
4.5.6	Bad-Pixel Parameters in a Recipe	51
4.5.6.1	Bad-pixel detection on a single image	51
4.5.6.2	Bad-pixel detection on a set of images	51



4.5.6.3	Cosmic Ray Hits detection	52
4.6	Strehl	52
4.6.1	Algorithm - short description	52
4.6.2	Functions	52
4.6.3	Inputs	53
4.6.4	Outputs	53
4.6.5	Examples	54
4.6.6	Strehl Parameters in a Recipe	54
4.7	Fringing	54
4.7.1	Master fringe computation	54
4.7.1.1	Algorithm - short description	55
4.7.1.2	Functions	56
4.7.1.3	Inputs	57
4.7.1.4	Outputs	57
4.7.2	Master fringe correction	58
4.7.2.1	Algorithm - short description	58
4.7.2.2	Functions	58
4.7.2.3	Inputs	59
4.7.2.4	Outputs	59
4.8	Object catalogue generation	59
4.8.1	Algorithm - short description	60
4.8.2	Functions	60
4.8.3	Inputs	61
4.8.4	Outputs	61
4.8.5	Examples	62
4.8.6	Catalogue Parameters in a Recipe	63
4.9	Efficiency	64
4.9.1	Algorithm - short description	65
4.9.2	Functions	65
4.9.3	Inputs	66
4.9.4	Outputs	66
4.9.5	Examples	66
4.9.6	Notes	67



- 4.10 Response . . . . . 67
  - 4.10.1 Algorithm - short description . . . . . 67
    - 4.10.1.1 Telluric Correction . . . . . 67
    - 4.10.1.2 Response Calculation . . . . . 68
  - 4.10.2 Functions . . . . . 69
  - 4.10.3 Inputs . . . . . 69
  - 4.10.4 Outputs . . . . . 70
  - 4.10.5 Examples . . . . . 70
  - 4.10.6 Notes . . . . . 71
- 4.11 Differential Atmospheric Refraction (DAR) . . . . . 72
  - 4.11.1 Algorithm - short description . . . . . 72
    - 4.11.1.1 Owens saturation pressure . . . . . 73
    - 4.11.1.2 Filippenko refractive index . . . . . 73
  - 4.11.2 Function . . . . . 73
  - 4.11.3 Inputs . . . . . 74
  - 4.11.4 Outputs . . . . . 75
  - 4.11.5 Example . . . . . 75
  - 4.11.6 DAR Parameters in a Recipe . . . . . 75
- 4.12 Effective air mass . . . . . 76
  - 4.12.1 Algorithm - short description . . . . . 76
    - 4.12.1.1 Zenith distance . . . . . 77
    - 4.12.1.2 Hardie approximation (1962) . . . . . 77
    - 4.12.1.3 Young & Irvine approximation (1967) . . . . . 77
    - 4.12.1.4 Young approximation (1994) . . . . . 78
  - 4.12.2 Functions . . . . . 78
  - 4.12.3 Outputs . . . . . 79
  - 4.12.4 Example . . . . . 79
- 4.13 Fixed Pattern Noise (fpn) . . . . . 79
  - 4.13.1 Algorithm - short description . . . . . 80
  - 4.13.2 Function . . . . . 81
  - 4.13.3 Inputs . . . . . 81
  - 4.13.4 Outputs . . . . . 81
  - 4.13.5 Example . . . . . 81



4.14	Image and Cube Resampling . . . . .	83
4.14.1	Algorithm - short description . . . . .	83
4.14.1.1	Nearest Neighbour . . . . .	83
4.14.1.2	Linear . . . . .	84
4.14.1.3	Quadratic . . . . .	84
4.14.1.4	Renka . . . . .	84
4.14.1.5	Drizzle . . . . .	84
4.14.1.6	Lanczos . . . . .	84
4.14.2	Functions . . . . .	85
4.14.3	Inputs . . . . .	85
4.14.4	Outputs . . . . .	89
4.14.5	Example . . . . .	91
4.14.6	Additional Information . . . . .	92
4.15	Limiting Magnitude . . . . .	93
4.15.1	Algorithm - short description . . . . .	93
4.15.2	Functions . . . . .	94
4.15.3	Inputs . . . . .	94
4.15.4	Outputs . . . . .	95
4.15.5	Example . . . . .	95
4.16	Barycentric correction . . . . .	96
4.16.1	Algorithm - short description . . . . .	96
4.16.2	Functions . . . . .	96
4.16.3	Inputs . . . . .	97
4.16.4	Outputs . . . . .	98
4.16.5	Example . . . . .	98
<b>5</b>	<b>Cookbook</b>	<b>99</b>
5.1	Handling large datasets . . . . .	99
5.1.1	Examples . . . . .	99
<b>A</b>	<b>Abbreviations and acronyms</b>	<b>101</b>
<b>B</b>	<b>Statistical Estimators</b>	<b>102</b>
B.1	Arithmetic mean - Arithmetic weighted mean . . . . .	102



---

B.2	Median	102
B.3	Kappa-sigma clipping	104
B.4	Mode	105
B.4.1	Mode algorithm: HDRL_MODE_FIT	107
B.4.2	Mode algorithm: HDRL_MODE_WEIGHTED	108
B.4.3	Mode algorithm: HDRL_MODE_MEDIAN	109
<b>C</b>	<b>Bad Pixel Determination</b>	<b>111</b>
C.1	Detailed Description of the various Algorithms	111
C.1.1	Bad-pixel detection on a single image	111
C.1.2	Bad-pixel detection on a stack of images	111
C.1.3	Bad-pixel detection on a sequence of images	114
C.1.4	Cosmic Ray Detection	116
C.1.5	Testing the Cosmic Ray Detection	121
<b>D</b>	<b>Masterfringe Computation and Removal</b>	<b>125</b>
D.1	Detailed Description of the de-fringing Algorithm	125
<b>E</b>	<b>Object Catalogue Generation</b>	<b>126</b>
E.1	Detailed Description of the Object Catalogue Generation Algorithm	126
E.2	Detailed Description of the Object Catalogue Table	126
<b>F</b>	<b>Efficiency Calculation</b>	<b>127</b>
F.1	Introduction	127
F.2	Testing	127
<b>G</b>	<b>Response Calculation</b>	<b>128</b>
G.1	Introduction	128
G.2	Testing	128
<b>H</b>	<b>Differential Atmospheric Refraction</b>	<b>132</b>
H.1	Introduction	132
H.2	Testing HDRL DAR	132
H.3	Testing the <i>hdrl_dar</i> Differential Atmospheric Refraction Routines with SINFONI Data	133
H.3.1	I. Method and Qualitative Results	133
H.3.2	II. Quantitative Results	135



---

H.3.3	The Strehl Ratios in <i>hdrl_dar</i> -Corrected Images . . . . .	138
<b>I</b>	<b>Fixed Pattern Noise</b>	<b>140</b>
<b>J</b>	<b>Image and Cube Resampling</b>	<b>143</b>
J.1	Introduction . . . . .	143
J.2	Testing the <i>hdrldemo_resample</i> Routines Using Synthetic Data . . . . .	143
J.3	Results from HAWK-I Globular Cluster Images . . . . .	147
J.3.1	NGC288 . . . . .	147
J.3.2	M30 . . . . .	151
J.4	Results from Strongly Rotated HAWK-I Images . . . . .	156
J.5	The Propagation of Edge Effects with Resampled Images . . . . .	159
J.6	Results from Synthetic Images With Right Ascension values At the Meridian and Declination near the Poles	159
J.7	Testing the <i>hdrldemo_resample</i> Routines Using MUSE Data Cubes . . . . .	161
J.7.1	Spatial resolution . . . . .	162
J.7.2	Difference in spectra . . . . .	164
J.7.3	Difference in derived properties . . . . .	167
J.8	Error Handling in <i>hdrldemo_resample</i> . . . . .	168
J.8.1	Gaussian Errors . . . . .	168
J.8.2	Constant Errors . . . . .	170
<b>K</b>	<b>Definition of Limiting Magnitude</b>	<b>171</b>



## 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* project 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 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] E. Bertin and S. Arnouts. SExtractor: Software for source extraction. *Astronomy and Astrophysics, Supplement*, 117:393–404, June 1996. [145](#), [147](#)
- [2] F. Clarke, N. Thatte, M. Tecza, I. Bryson, H. Schetler, T. Fusco, J.-F. Sauvage, B. Neichel, S. Ramsay, P. Hammersley, J. Vernet, and S. Egner. HARMONI Technical Specification. *ESO-257872*, 2019. [143](#)
- [3] Anirban DasGupta. *Asymptotic Theory of Statistics and Probability*. Springer, New York, 2008. [103](#)
- [4] M. Hamuy, N. B. Suntzeff, S. R. Heathcote, A. R. Walker, P. Gigoux, and M. M. Phillips. Southern spectrophotometric standards, 2. *Publications of the Astronomical Society of the Pacific*, 106:566–589, June 1994. [127](#), [128](#)



- [5] M. Hamuy, A. R. Walker, N. B. Suntzeff, P. Gigoux, S. R. Heathcote, and M. M. Phillips. Southern spectrophotometric standards. *Publications of the Astronomical Society of the Pacific*, 104:533–552, July 1992. [127](#), [128](#)
- [6] M. J. Irwin. Automatic analysis of crowded fields. *Monthly Notices of the Royal Astronomical Society*, 214:575–604, June 1985. [17](#), [59](#)
- [7] 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. [103](#)
- [8] M. Neeser, L. Piqueras, and A Gabasch. Interpolation Algorithms and Sub-Pixel Regridding for the High-Level Data Reduction Library (HDRL). *Science Data Products Board Proposal*, May 2019. [143](#)
- [9] Paul R. Rider. Variance of the median of small samples from several special populations. *Journ. Am. Stat. Assoc.*, 55:148–150, mar 1960. [103](#), [104](#)
- [10] P. G. van Dokkum. Cosmic-ray rejection by laplacian edge detection. *PASP*, 113:1420–1427, nov 2001. [116](#)



## 2 Software Infrastructure

### 2.1 SVN usage and library integration

The *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 sub-directory of `http://svn.hq.eso.org/p2/tags/Pipelines/common/hdr1` (see also sect. 2.2). In order to build the library one has to add the *hdr1.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 hdr1 folder as containing a configurable package
AC_CONFIG_SUBDIRS([hdr1])

# check and define all required variables to build and
# link hdr1 external located in the hdr1 folder
HDRL_CHECK([hdr1])
```

- *Makefile.am*:

```
SUBDIRS = hdr1 ...
```

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 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 <hdr1.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 1.6.0

The *High Level Data Reduction Library* release version 1.6.0 can be obtained from the HDRL Git repository <https://gitlab.eso.org/pipelines/hdrl.git> (tag `hdrl-1.6.0`)

In this release we updated and/or added the following algorithms:

- The *High Level Data Reduction Library* catalogue has been updated to follow [ESO Phase 3 conventions](#). This ensures compatibility with ESO Phase 3 data products and standards.
- The build system has been enhanced to support *High Level Data Reduction Library* as a standalone third-party library. While *High Level Data Reduction Library* remains usable as a convenience library as before, it can now be installed as a standalone library using the `-enable-standalone` configure option, providing greater flexibility for integration into different build environments.
- Various quality control and bug-fix improvements have been included (see the `ChangeLog` file for detailed information).
- Licensing and documentation have been clarified for GPLv2-or-later use by homogenising headers and FSF address, replacing the legacy `LICENSE` file with a canonical `COPYING`, and adding GNU-style distribution files (`README`, `INSTALL`, `AUTHORS`, `ChangeLog`, `NEWS`) that are shipped in the tarball and installed under `docdir` for standalone builds.

## 2.2.2 Release version 1.5.0

The *High Level Data Reduction Library* release version 1.5.0 can be included from <http://svn.hq2.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-1.5.0>

In this release we updated and/or added the following algorithms:

- The computation of the barycentric correction, i.e. the wavelength shift to apply to a spectrum to compensate for the motion of the observer with respect to the barycenter of the solar system. The implemented algorithm derives the barycentric correction of an observation by using the [ERFA](#) (Essential Routines for Fundamental Astronomy) library. ERFA is a C library containing key algorithms for astronomy, and is based on the [SOFA library](#) published by the International Astronomical Union (IAU). See section 4.16 for detailed information.
- The build system has been modified to include GSL as direct dependency to the `hdrl` unit-test as direct GSL calls are performed when unit-testing the limiting magnitude module.
- For the barycentric correction algorithm two additional dependencies were added to the *High Level Data Reduction Library*.
  - A dependency to the [ERFA](#) (Essential Routines for Fundamental Astronomy) library
  - A dependency to the [libcurl](#) (The multiprotocol file transfer library) library



## 2.2.3 Release version 1.4.0

The *High Level Data Reduction Library* release version 1.4.0 can be included from <http://svnHQ2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-1.4.0>

In this release we updated and/or added the following algorithms:

- The computation of the limiting magnitude of an image as defined in the *ESO Phase 3 Standard*. The limiting magnitude characterizes the depth of an observation and is defined as the magnitude of a unresolved source whose flux is 5 times the noise background, i.e. the magnitude of a point like source detected with  $\frac{S}{N} = 5$ . See section 4.15 and appendix K for detailed information.
- To the statistical estimators we added the **mode** of a distribution, i.e. the following algorithms are now supported for collapsing imagelists or deriving statistics on images:
  - Mean
  - Weighted mean
  - min-max rejected mean
  - $\kappa\sigma$  clipped mean
  - Median
  - Mode

Please note that all methods but the mode are doing **error propagation**. The mode method is special in this case as it **calculates the error from the data**. The error estimation can either be done analytically or based on a bootstrap Montecarlo simulation. In this case the input data are perturbed with the bootstrap technique and the mode is calculated N times (controlled with a parameter). From this N modes the standard deviation is calculated and returned as error. See section B.4 for detailed information on the mode algorithm.

- Due to the addition of the mode, the functions `hdrl_overscan_parameter_create_parlist()` and `hdrl_collapse_parameter_create_parlist()` have changed. The two functions now require an additional default mode hdrl parameter. See the doxygen information and e.g. section 4.1.4 and section 4.2.6 for more details.

## 2.2.4 Release version 1.3.0

The *High Level Data Reduction Library* release version 1.3.0 can be included from <http://svnHQ2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-1.3.0>

In this release we updated and/or added the following algorithms:

- Resampling of 2-dimensional images and 3-dimensional cubes. A common problem in astronomy is the resampling of images (or cubes) onto a common grid. Ideally, this is done only once in the data reduction workflow as each sub pixel resampling redistributes the flux and leads to correlations. The algorithm provided by the *High Level Data Reduction Library* is doing the 2D and 3D interpolation



in 2-dimensional and 3-dimensional spaces, respectively. Currently there are six different interpolation methods implemented:

- **Nearest:** Nearest neighbour resampling
- **Linear:** Weighted resampling using an inverse distance weighting function
- **Quadratic:** Weighted resampling using a quadratic inverse distance weighting function
- **Renka:** Weighted resampling using a Renka weighting function
- **Drizzle:** Weighted resampling using a drizzle-like weighting scheme
- **Lanczos:** Weighted resampling using a lanczos-like restricted sinc as weighting function

See section 4.14 and appendix J for detailed information.

- The object catalogue generation code (see section 4.8) has been updated. In previous releases, pixels with a value of exactly 0 were automatically added to the confidence map as zero<sup>1</sup> and excluded in all further computations. This has been removed.

## 2.2.5 Release version 1.2.0

The *High Level Data Reduction Library* release version 1.2.0 can be included from <http://svn.hq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/hdrl-1.2.0>

In this release we updated and/or added the following algorithms:

- Detection of fixed pattern noise. A classical example is pick noise, i.e. low-amplitude, quasi-periodical patterns super-imposed on the normal read-noise. It is due to electronic interference and might show up or disappear on short timescales (days or hours). The algorithm tries to identify it by the usage of the power spectrum. See section 4.13 for detailed information.
- An error in the documentation of the Strehl ratio variable `m1_radius` and `m2_radius` was corrected (section 4.6.5). The code was correct.
- In the spectral efficiency computation (section 4.9) a sign error in the atmospheric correction (equation 7) was corrected in the documentation as well as in the code.

## 2.2.6 Release version 1.1.0

The *High Level Data Reduction Library* release version 1.1.0 can be included from <http://svn.hq2.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.

<sup>1</sup>“Bad” pixels are given a confidence value of zero.



- 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.

## 2.2.7 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 [6] for details)
- parametrise the detected objects, i.e. perform astrometry, photometry and a shape analysis.

For details see section 4.8.

## 2.2.8 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).



## 2.2.9 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)

## 2.2.10 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.

## 2.2.11 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)

## 2.2.12 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.



## HDRL Pipeline Developer Manual

Doc. Number: ESO-299492  
Doc. Version: 7  
Sof. Version: 1.6.0  
Released on: 2026-06-02  
Page: 19 of 173

---

- The [ERFA](#) (Essential Routines for Fundamental Astronomy) library version 1.3 or higher. ERFA is a C library containing key algorithms for astronomy, and is based on the [SOFA library](#) published by the International Astronomical Union (IAU).
- The [libcurl](#) (The multiprotocol file transfer library) library - any recent version.



## 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 where 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 an 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` whose 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)`
- `hdrl_collapse_mode_parameter_create(histo_min, histo_max, bin_size, mode_method, error_niter)`

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 high 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);
```

```
cpl_error_code hdrl_imagelist_collapse_mode(  
    const hdrl_imagelist * himlist,  
    double histo_min,  
    double histo_max,
```



```
double          bin_size,  
hdrl_mode_type  mode_method,  
cpl_size        error_niter,  
hdrl_image      ** out,  
cpl_image       ** contrib) ;
```

Please note that all methods but the mode are doing **error propagation**. The mode method is special in this case as it **calculates the error from the data**. The calculation depends on the parameter `error_niter`. If the parameter is set to 0, it is doing an analytically error estimation. If the parameter is larger than 0, the error is calculated by a bootstrap Montecarlo simulation from the input data with the value of the parameter specifying the number of simulations. In this case the input data are perturbed with the bootstrap technique and the mode is calculated `error_niter` times. From this modes the standard deviation is calculated and returned as error.

### 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 detect 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 `hdlr_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

```
hdlr_spectrum1D * s1 = /* get first spectrum from the outside */
hdlr_spectrum1D * s2 = /* get second spectrum from the outside */

const hdlr_spectrum1D_wavelength spec_wav =
    hdlr_spectrum1D_get_wavelength(s1);

/* Resample s2 on s1 wavelengths */
hdlr_parameter * params =
    hdlr_spectrum1D_resample_interpolate_parameter_create
    (hdlr_spectrum1D_interp_akima);

hdlr_spectrum1D * s3 =
    hdlr_spectrum1D_resample(s2, &spec_wav, params);

hdlr_parameter_delete(params);

/* Multiply every flux value by 0.4, and write the result in s3 */
hdlr_spectrum1D_mul_scalar(s3, (hdlr_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");
```



## 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, mode or sigma/minmax-clipped<sup>2</sup> 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 not to 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) ;
```

<sup>2</sup>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



## 4.1.1.1 Inputs

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

```
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) ;  
  
    hdrl_collapse_mode_parameter_create(double histo_min,  
                                       double histo_max,  
                                       double bin_size,  
                                       hdrl_mode_type mode_method,  
                                       cpl_size error_niter) ;
```

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 gives a general description of the overscan computation method.

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 4.1.2). `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  $\chi^2$  (`chi2`) and the reduced  $\chi^2$  (`red_chi2`), additionally `sigclip_reject_low`, `_high`, and `niter` if the collapsing method is sigma/minmax-clipping rejection as well as `histo_min`, `histo_max`, `bin_size`, `mode_method`, and `error_niter` if the collapsing method is the mode).

The possible collapsing methods can be mean, median, weighted mean, mode, sigma clipping, or minmax rejection. The appropriate function needs 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.

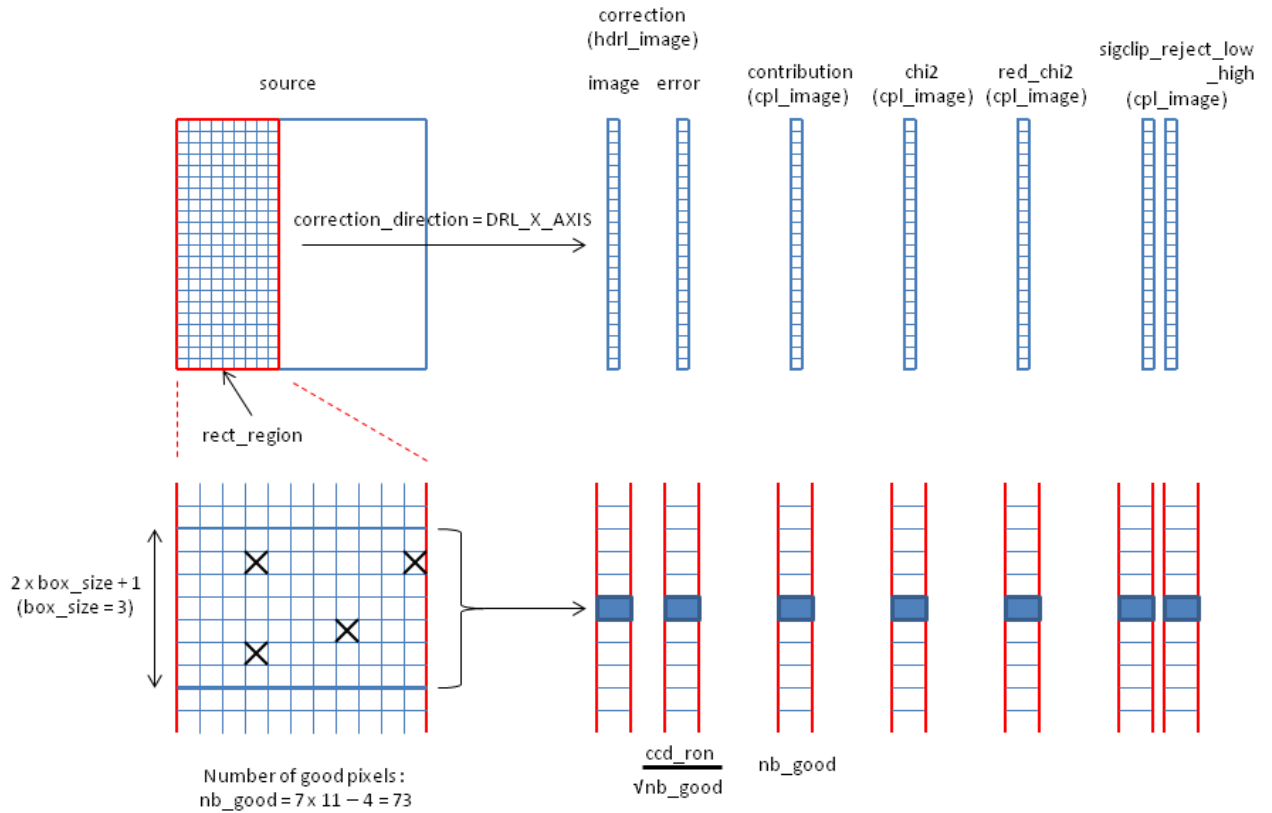


Figure 4.1.1: *Overscan Computation Algorithm description.*

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



and  $\frac{ccd\_ron}{\sqrt{contribution}}$  when the contribution is 1 or 2 pixels. The mode method is special in this case as it calculates the error from the data. The calculation depends on the parameter `error_niter`. If the parameter is set to 0, it is doing an analytically error estimation. If the parameter is larger than 0, the error is calculated by a bootstrap Montecarlo simulation from the input data with the value of the parameter specifying the number of simulations. In this case the input data are perturbed with the bootstrap technique and the mode is calculated `error_niter` times. From this modes the standard deviation is calculated and returned as error.

`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, mode 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  $\sum_i \left(\frac{source_i - correction}{ccd\_ron}\right)^2$  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  $\chi^2$ , i.e the  $\chi^2$  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 sigclip/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  $\sqrt{\text{overscan\_computation.error}^2 + \text{source\_error}^2}$ , 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.

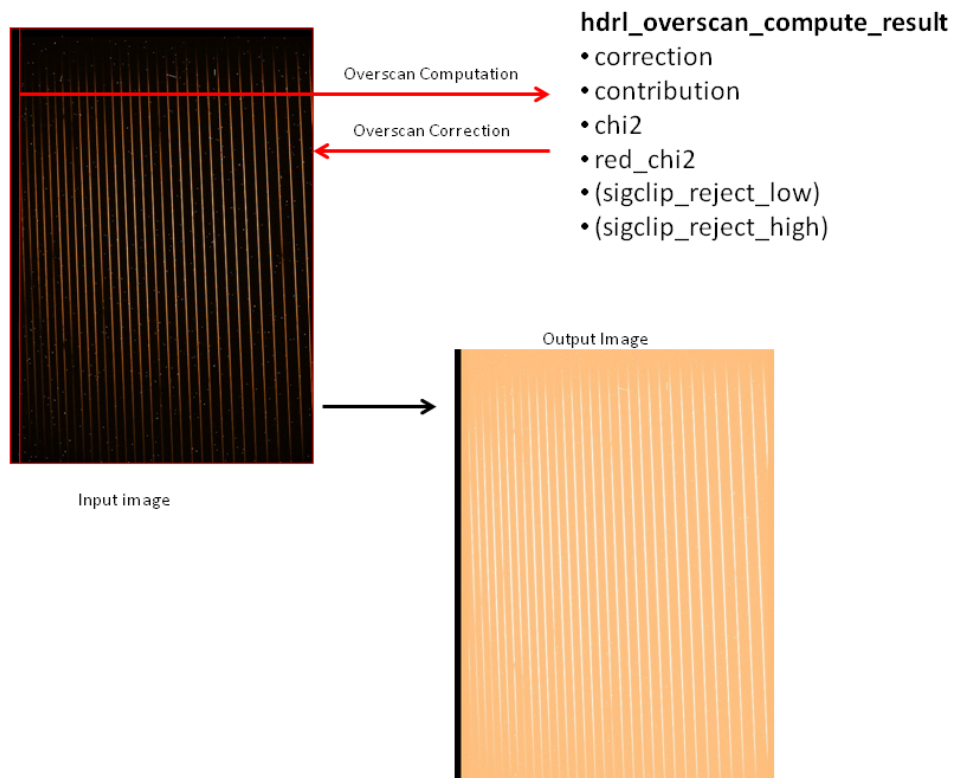


Figure 4.1.2: *Overscan Computation Example.*

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.

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 Overscan Computation parameters */
hdrl_parameter * rect_region_def =
    hdrl_rect_region_parameter_create(1, 1, 20, 1000);
hdrl_parameter * sigclip_def =
    hdrl_collapse_sigclip_parameter_create(3., 3., 5);
hdrl_parameter * minmax_def =
    hdrl_collapse_minmax_parameter_create(1., 1.);
hdrl_parameter * mode_def =
    hdrl_collapse_mode_parameter_create(10., 1., 0., HDRL_MODE_MEDIAN, 0);
cpl_parameterlist * os_comp = hdrl_overscan_parameter_create_parlist(
    RECIPE_NAME, "", "alongX", 0, 0., rect_region_def, "MEDIAN",
    sigclip_def, minmax_def, mode_def);
hdrl_parameter_delete(rect_region_def);
hdrl_parameter_delete(sigclip_def);
hdrl_parameter_delete(minmax_def);
hdrl_parameter_delete(mode_def);
for (cpl_parameter * p = cpl_parameterlist_get_first(os_comp) ;
     p != NULL; p = cpl_parameterlist_get_next(os_comp))
    cpl_parameterlist_append(self, cpl_parameter_duplicate(p));
cpl_parameterlist_delete(os_comp);
```

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 needs 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(kappa_low, kappa_high, niter)`
- `hdrl_collapse_minmax_parameter_create(nlow, nhigh)`
- `hdrl_collapse_mode_parameter_create(histo_min, histo_max, bin_size, mode_method, error_niter)`

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 high 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);  
hdrl_parameter * minmax_def =  
    hdrl_collapse_minmax_parameter_create(1., 1.);  
hdrl_parameter * mode_def =  
    hdrl_collapse_mode_parameter_create(10., 1., 0., HDRL_MODE_MEDIAN, 0);  
  
/*create parameters with context RECIPE_NAME under the collapse hierarchy*/  
  
cpl_parameterlist * pbiascollapse =hdrl_collapse_parameter_create_parlist(  
RECIPE_NAME, "collapse", "MEDIAN", sigclip_def, minmax_def, mode_def) ;  
  
hdrl_parameter_delete(sigclip_def);  
hdrl_parameter_delete(minmax_def);  
hdrl_parameter_delete(mode_def);  
  
/* add to recipe parameter list, change aliases,  
disable not required ones etc. */  
  
for (cpl_parameter * p = cpl_parameterlist_get_first(pbiascollapse) ;  
    p != NULL; p = cpl_parameterlist_get_next(pbiascollapse))  
    cpl_parameterlist_append(self, cpl_parameter_duplicate(p));  
cpl_parameterlist_delete(pbiascollapse);
```

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 |  
                            MODE> [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]  
--collapse.minmax.nlow      : Low number of pixels to reject for the minmax  
                            clipping algorithm. [1.0]  
--collapse.minmax.nhigh     : High number of pixels to reject for the  
                            minmax clipping algorithm. [1.0]
```



```
--collapse.mode.histo-min      : Minimum pixel value to accept for mode
                               : computation. [10.0]
--collapse.mode.histo-max      : Maximum pixel value to accept for mode
                               : computation. [1.0]
--collapse.mode.bin-size       : Binsize of the histogram. [0.0]
--collapse.mode.method         : Mode method (algorithm) to use. <MEDIAN |
                               : WEIGHTED | FIT> [MEDIAN]
--collapse.mode.error-niter    : Iterations to compute the mode
                               : error. [0]
...
```

## 4.3 Dark

The dark current of a CCD detector can be measured by using frames with an exposure time larger than 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 an 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 than 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(kappa_low, kappa_high, niter)`
- `hdrl_collapse_minmax_parameter_create(nlow, nhigh)`
- `hdrl_collapse_mode_parameter_create(histo_min, histo_max, bin_size, mode_method, error_niter)`

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);  
hdrl_parameter * minmax_def =  
    hdrl_collapse_minmax_parameter_create(1., 1.);  
hdrl_parameter * mode_def =  
    hdrl_collapse_mode_parameter_create(10., 1., 0., HDRL_MODE_MEDIAN, 0);  
  
/* create collapse parameters with context RECIPE_NAME under the  
collapse hierarchy */
```



```
cpl_parameterlist * pflatcollapse =hdrl_collapse_parameter_create_parlist(  
RECIPE_NAME, "collapse", "MEDIAN", sigclip_def, minmax_def, mode_def) ;  
  
hdrl_parameter_delete(sigclip_def);  
hdrl_parameter_delete(minmax_def);  
hdrl_parameter_delete(mode_def);  
  
/* add to recipe parameter list, change aliases, disable not required ones etc. */  
  
for (cpl_parameter * p = cpl_parameterlist_get_first(pflatcollapse) ;  
     p != NULL; p = cpl_parameterlist_get_next(pflatcollapse))  
    cpl_parameterlist_append(self, cpl_parameter_duplicate(p));  
cpl_parameterlist_delete(pflatcollapse);  
  
/* 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 |  
MODE> [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]  
--collapse.minmax.nlow : Low number of pixels to reject for the minmax  
clipping algorithm. [1.0]  
--collapse.minmax.nhigh : High number of pixels to reject for the  
minmax clipping algorithm. [1.0]  
--collapse.mode.histo-min : Minimum pixel value to accept for mode  
computation. [10.0]  
--collapse.mode.histo-max : Maximum pixel value to accept for mode  
computation. [1.0]  
--collapse.mode.bin-size : Binsize of the histogram. [0.0]
```



```
--collapse.mode.method      : Mode method (algorithm) to use. <MEDIAN |  
                           WEIGHTED | FIT> [MEDIAN]  
--collapse.mode.error-niter : Iterations to compute the mode  
...  
  
--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 * imglst_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  $\text{mean} \pm \text{cutoff} \times \text{standard deviation}$  are considered bad.
- fit-coefficient-method: Relative cutoff on the distribution of the fit coefficients. Pixels with fit coefficients which exceed of  $\text{mean} \pm \text{cutoff} \times \text{standard deviation}$  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 * imglst_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 were 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 recipe 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 an 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` refer to  $\sigma_{lim}$  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.

```
hdr1_parameter * bpm_2d_param =  
    hdr1_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 `hdr1` image, `himage`, and the previously defined bad pixels:

```
cpl_mask * mask = hdr1_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.

```
hdr1_parameter * bpm_2d_param =  
    hdr1_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 `hdr1` image, `himage`, and the previously defined bad pixels:

```
imlist = hdr1_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 `himg` 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 than 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  $1.635 \mu\text{m}$  with a telescope<sup>3</sup> of 5.08 m diameter with a central obstruction of 1.8288 m, and the detector pixel size on Sky is  $33.1932 \text{ mas} \times 33.1932 \text{ mas}$  (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,  
5.08/2, 1.8288/2, 0.0331932, 0.0331932, 1.5, 1.5, 2.5);  
  
strehl = hdrl_strehl_compute(hima, params);
```

Please note that for ESO-VLT instruments, the UT telescope diameter is 8.115 m, while the central obstruction depends on the instrument used, some instrument deploy a baffle, some not. The user is recommended to adopt the following obstruction values:

- 1.116m for UT1-3 with no baffle (KMOS, NACO, VISIR, SPHERE)
- 1.550m for UT1-3 with baffle (FORS, X-shooter, UVES, FLAMES)
- 1.550m for UT4 (all instruments)

## 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 `hdrl_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.

<sup>3</sup>The telescope diameter and central obstruction values indicated in this example refer to the Palomar telescope.



## 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 `hdlr` 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`: An 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 `hdlr` 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

$$f(x) = c_1 e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + c_2 e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}}. \quad (1)$$

The means  $\mu_1$  and  $\mu_2$  ( $\mu_1 < \mu_2 = \mu_1 + a$ ) 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:

$$f(x) \approx \sum_{n=0}^p c_n h_n \left( \frac{x - \mu}{\sigma} \right), \quad (2)$$

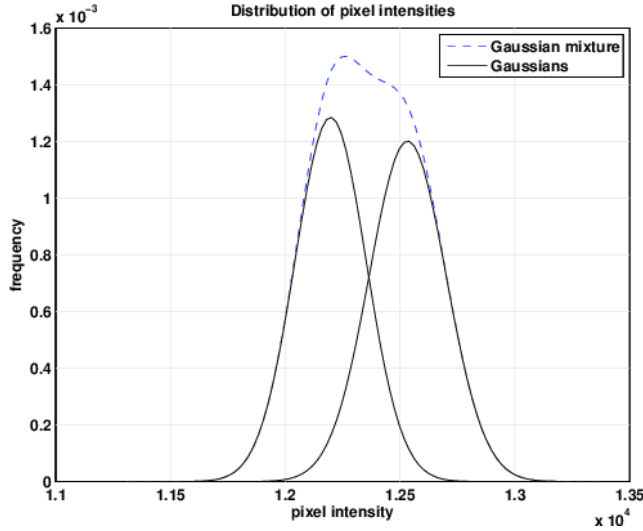


Figure 4.7.1: Density of a mixture of two normal distributions, whose means are the background and the fringe amplitudes, respectively.

where  $h_n$  is the normalized Hermite function

$$h_n(x) = \pi^{-\frac{1}{4}} 2^{-\frac{n}{2}} (n!)^{-\frac{1}{2}} (1)^n e^{\frac{x^2}{2}} \frac{d^n}{dx^n} (e^{-x^2}), \quad (3)$$

$\mu$  and  $\sigma$  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

$$c_n = \frac{1}{\sigma N} \sum_{i=1}^N h_n \left( \frac{I_i - \mu}{\sigma} \right), \quad (4)$$

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

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(kappa_low, kappa_high, niter)`
- `hdrl_collapse_minmax_parameter_create(nlow, nhigh)`
- `hdrl_collapse_mode_parameter_create(histo_min, histo_max, bin_size, mode_method, error_niter)`

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  $i$ th fringe  $F_i$  is estimated as

$$F_i = a_i F + b_i, \quad (5)$$

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 performed 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 [6] 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 <sup>4</sup>
- `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`):

<sup>4</sup>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



```
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
```

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

$$m = \text{MAGZPT} - 2.5 \log_{10}(\text{APER\_FLUX\_3}/\text{EXPTIME}) - \text{APCOR3} \quad (6)$$

With `APER_FLUX_3` being a column in the catalogue table, `MAGZPT` the zeropoint and `EXPTIME` the exposure time<sup>5</sup>.

## 4.8.5 Examples

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

<sup>5</sup>For more information, refer to <http://casu.ast.cam.ac.uk/surveys-projects/vst/technical/catalogue-generation>



```
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 */

/* 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

$$\epsilon(\lambda) = \frac{I_{std}(\lambda) \cdot 10^{[-0.4(A_p - A_m)E_x^{(r)}(\lambda)]} \cdot G \cdot E_{ph}(\lambda)}{[T_{ex} A_{tel} I_{std-ref}^{(r)}(\lambda)]} \quad (7)$$

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  $\lambda$ .  $E_{ph}(\lambda) = \frac{hc}{\lambda} = \frac{1.986 \cdot 10^{-12}}{\lambda} J/\mu m$ .  $\lambda$  is expressed in  $\mu m$ ;
- $I_{std-ref}^{(r)}(\lambda)$  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  $\lambda$  only if  $\lambda$  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 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 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 <sup>6</sup>.

```
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);
```

<sup>6</sup>See: [https://www.gnu.org/software/gsl/manual/html\\_node/Interpolation-Types.html](https://www.gnu.org/software/gsl/manual/html_node/Interpolation-Types.html)



## 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  $\text{erg/s/cm}^2/\text{Angstrom}$ . 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 ratio  $\lambda/\text{FWHM}$  is constant. A strong prerequisite is that the sampling step of the logarithmic wavelength scale is very fine for both spectra to provide proper sampling. These resampled spectra are used only for the cross correlation described below;
2. Extract the range of wavelengths needed to calculate the cross correlation (water vapour feature of medium strength);
3. Cross correlate the telluric model with the observation spectrum over the extracted range. The cross correlation is needed to determine the shift and difference in resolution between the telluric model spectrum and the observation;
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 (this assumes that the resolution of the telluric model is much higher than that of the observation);



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 as well as regions without telluric absorption.

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 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

$$R(\lambda) = \frac{I_{std-ref}^{(r)}(\lambda) \cdot 10^{[0.4(A_p - A_m)E_x^{(r)}(\lambda)]} \cdot G \cdot T_{ex}}{I_{std}(\lambda)} \quad (8)$$

where:



- $I_{std}(\lambda)$  is the observed 1D spectrum [ADU] as function of wavelength [nm]. If telluric correction is enabled,
- $E_x^{(r)}(\lambda)$  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 [ADU/e];
- $I_{std-ref}^{(r)}(\lambda)$  resampled reference standard star spectrum. The wavelength is expressed in [nm]. The corresponding, user-provided spectrum is denoted as  $I_{std-ref}(\lambda)$ ;

$E_x(\lambda)$  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  $\lambda$  only if  $\lambda$  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 the 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 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 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 <sup>7</sup>.

<sup>7</sup>See: [https://www.gnu.org/software/gsl/manual/html\\_node/Interpolation-Types.html](https://www.gnu.org/software/gsl/manual/html_node/Interpolation-Types.html)



```
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,
    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/cm<sup>2</sup>/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 and H.3.2).

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 `hdr1_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<sup>8</sup>. The main loop compute the differential refractive offset for the input reference wavelengths, and stores the shift in the coordinates, taking into account

<sup>8</sup>See: <https://emtoolbox.nist.gov/Wavelength/Documentation.asp#AppendixA>, for the formulae used.



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:

$$s_p = -10474 + 116.43 T - 0.43284 T^2 + 0.00053840 T^3 \quad (9)$$

where  $T$  is the temperature in Kelvins.

#### 4.11.1.2 Filippenko refractive index

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

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

where  $\lambda$  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):

$$(n(\lambda)_{T,P} - 1) = (n(\lambda)_{15,760} - 1) \cdot \frac{P[1 + (1.049 - 0.0157 T)10^{-6} P]}{720.883(1 + 0.003661 T)} \quad (11)$$

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

$$\frac{0.0624 - 0.000680/\lambda^2}{1 + 0.003661 T} f \quad (12)$$

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

$$f = 0.75006158 \cdot s_p \cdot h \quad (13)$$

where  $s_p$  is the saturation pressure of equation 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`: Parallax angle during exposure [deg].
- `posang`: Position angle on the sky [deg].
- `temp`: Temperature [°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 function 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 sidereal 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<sup>9</sup>:

<sup>9</sup>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](http://en.wikipedia.org/w/index.php?title=Airmass&oldid=358226579#Interpolative_formulas).



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  $\pi$ , 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  $2\pi$ .

#### 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 supposedly 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 sidereal time elapsed since sidereal 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);
```

## 4.13 Fixed Pattern Noise (fpn)

This section describes how HDRL identifies regular noise on detector data. A classical example is provided by the pick noise, i.e. low-amplitude, quasi-periodical patterns super-imposed on the normal read-noise. It is due to electronic interference and might show up or disappear on short timescales (days or hours).

Those artifacts are visible only on the detector data but of course exist also on other calibration data and on science data where they may compromise the detector sensitivity.



## 4.13.1 Algorithm - short description

The algorithm implements three steps that can be described as follows:

1. Derive the power spectrum of the image using the Fast Fourier Transform (FFT):

$$\begin{aligned}fft &= FFT\_2D(img) \\power\_spec &= abs(fft)^2\end{aligned}$$

2. Mask the peak of the power spectrum that corresponds to the pixel-to-pixel variations. By default only one pixel placed in the corner on the bottom left is masked (the DC component), but more pixels can be masked providing an optional bad pixel mask, e.g. if there is a pixel-to-pixel cross talk on longer spatial scale).

$$power\_spec\_filter = filter\_peak(power\_spec, 1, 1)$$

3. Calculate the standard deviation  $std$  and the standard deviation based on the Median Absolute Deviation (MAD)  $std_{mad}$  of the power spectrum by taking the masked regions into account:

$$\begin{aligned}std &= stdev(power\_spec\_filter) \\std_{mad} &= mad(power\_spec\_filter) * 1.4826\end{aligned}$$

As mentioned above, when computing the standard deviation and the MAD-based standard deviation on the power spectrum the algorithm excludes the masked region. For this the user can provide an optional mask `mask_in` or use the `dc_mask_x` and `dc_mask_y` function parameter to create one on the fly. The mask created on the fly will start at pixel (1,1) and extend in both direction up to  $(dc\_mask\_x, dc\_mask\_y)^{10}$ . Moreover, the mask created on the fly by setting `dc_mask_x` and `dc_mask_y` and the optional mask (`mask_in`) are combined and are both taken into account when calculating the statistics (`std` and `std_mad`). Furthermore, the final (combined) mask is attached to the returned `power_spectrum` image as normal `cpl_mask` and can be retrieved with the CPL function `cpl_image_get_bpm(power_spectrum)`

---

<sup>10</sup>Please note that in the current implementation the power spectrum contains the DC component (the DC term is the 0 Hz term and is equivalent to the average of all the samples in the window) in pixel (1,1).



## 4.13.2 Function

The power spectrum and its statistics are computed by the following function:

```
cpl_error_code hdrl_fpn_compute(  
    cpl_image      * img_in,  
    const cpl_mask * mask_in,  
    const cpl_size  dc_mask_x,  
    const cpl_size  dc_mask_y,  
    cpl_image      ** power_spectrum,  
    double         * std,  
    double         * std_mad)
```

The `power_spectrum` pointer is filled by the function with an allocated image which must be deleted by the user when not required anymore.

## 4.13.3 Inputs

The following function parameters have to be passed to the `hdrl_fpn_compute` function:

- `img_in`: the input image. Please note that bad pixels are not allowed
- `mask_in` (or `NULL`): an optional mask which is used when deriving the standard deviations on the power spectrum
- `dc_mask_x`: x-pixel window ( $\geq 1$ ) to discard DC component starting from pixel (1, 1)
- `dc_mask_y`: y-pixel window ( $\geq 1$ ) to discard DC component starting from pixel (1, 1)

## 4.13.4 Outputs

The result is the computed power spectrum (`power_spectrum`), the standard deviation (`std`) of the power spectrum, the MAD based standard deviation of the power spectrum (`std_mad`) and the final mask used to compute `std` and `std_mad` which can be retrieved with `cpl_image_get_bpm(power_spectrum)`

Please note that the `power_spectrum` pointer is filled by the function with an allocated image which must be deleted by the user when not required anymore.

## 4.13.5 Example

This section briefly describes the call to the function. For a more detailed implementation please see also the `hdrl_demo_fpn` example recipe.

```
/* Load the image and the (optional) mask */  
cpl_mask *mask_in = cpl_mask_load(...);  
cpl_image *img_in = cpl_image_load(...);  
  
/* Compute the power spectrum and the statistics */  
cpl_image *power_spectrum = NULL;  
double std = -1.;
```



```
double    std_mad    = -1.;  
  
hdrl_fpn_compute(img_in, mask_in, dc_mask_x, dc_mask_y,  
                 &power_spectrum, &std, &std_mad);  
  
/* save the power spectrum and the used mask */  
...  
  
/* Cleanup the memory */  
cpl_image_delete(power_spectrum);  
cpl_image_delete(img_in);  
cpl_mask_delete(mask_in);
```



## 4.14 Image and Cube Resampling

A common problem in astronomy is the resampling of 2D images and 3D cubes onto a common grid. Ideally, this is done only once in the data reduction workflow as each sub-pixel resampling redistributes the flux and creates correlation. In the resampling *High Level Data Reduction Library* project we would like to address two different scenarios. Assuming that one would like to re-grid and stack 10 dithered images/cubes with pre-determined World Coordinate System (WCS). Then one should be able to:

- Combine all images/cubes in a single cloud of points (we use a table for this step - see below) and when creating the resampled output image/cube all information is used in a single interpolation step, i.e. no image-by-image interpolation is done.
- Resample each individual image/cube on the same output grid (image by image interpolation) and combine the single resampled images in a second step with hdrl stacking methods.

Both scenarios interpolate only once, but the second scenario is important in case not all bad pixels could be properly determined and inserted in the bad pixel mask upfront. Then the implemented HDRL pixel resampling function offers the possibility to use a robust pixel combination method (median or  $\kappa\sigma$  clipping) to combine the images/cubes.

### 4.14.1 Algorithm - short description

The implemented interpolation algorithms are based on the MUSE pipeline and work for 2D images and 3D cubes. The 2D and 3D interpolation is done in 2-dimensional and 3-dimensional spaces, respectively. Currently there are six different interpolation methods implemented:

- **Nearest:** Nearest resampling
- **Linear:** Weighted resampling using an inverse distance weighting function
- **Quadratic:** Weighted resampling using a quadratic inverse distance weighting function
- **Renka:** Weighted resampling using a Renka weighting function
- **Drizzle:** Weighted resampling using a drizzle-like weighting scheme
- **Lanczos:** Weighted resampling using a lanczos-like restricted sinc as weighting function

#### 4.14.1.1 Nearest Neighbour

The algorithm does not use any weighting function but simply uses the value of the nearest neighbour inside an output voxel<sup>11</sup> centre as the final output value. If there is no nearest neighbour inside the voxel (but e.g. only outside), the voxel is marked as bad. This speeds up the algorithm considerably. There are no control parameter for this method.

---

<sup>11</sup>In 3D computer graphics, a voxel represents a value on a regular grid in three-dimensional space. See <http://en.wikipedia.org/wiki/Voxel> for more information.



## 4.14.1.2 Linear

The algorithm uses a linear inverse distance weighting function ( $\frac{1}{r}$ ) for the interpolation. The parameter `loop_distance` controls the number of surrounding pixels that are taken into account on the final grid, e.g. a `loop_distance` of 1 uses 3 pixels ( $x - 1, x, x + 1$ ) in each dimension, i.e. 9 in total for a 2D image and 27 in total for a 3D cube. Moreover, if the parameter `use_errorweights` is set to `TRUE`, an additional weight, defined as  $1/\text{variance}$ , is taken into account<sup>12</sup>.

## 4.14.1.3 Quadratic

The algorithm uses a quadratic inverse distance weighting function ( $\frac{1}{r^2}$ ) for the interpolation. The parameter `loop_distance` controls the number of surrounding pixels that are taken into account on the final grid, e.g. a `loop_distance` of 1 uses 3 pixels ( $x - 1, x, x + 1$ ) in each dimension, i.e. 9 in total for a 2D image and 27 in total for a 3D cube. Moreover, if the parameter `use_errorweights` is set to `TRUE`, an additional weight, defined as  $1/\text{variance}$ , is taken into account<sup>12</sup>.

## 4.14.1.4 Renka

The algorithm uses a modified Shepard-like distance weighting function following Renka for the interpolation. The parameter `critical_radius` defines the distance beyond which the weights are set to 0 and the pixels are therefore not taken into account. The parameter `loop_distance` controls the number of surrounding pixels that are taken into account on the final grid, e.g. a `loop_distance` of 1 uses 3 pixels ( $x - 1, x, x + 1$ ) in each dimension, i.e. 9 in total for a 2D image and 27 in total for a 3D cube. Moreover, if the parameter `use_errorweights` is set to `TRUE`, an additional weight, defined as  $1/\text{variance}$ , is taken into account<sup>12</sup>.

## 4.14.1.5 Drizzle

The algorithm uses a drizzle-like distance weighting function for the interpolation. The down-scaling factors `pix_frac_x`, `pix_frac_y`, and `pix_frac_lambda`, for  $x$ ,  $y$ , and wavelength direction control the percentage of flux of the original pixel/voxel that drizzles into the target pixel/voxel. The parameter `loop_distance` controls the number of surrounding pixels that are taken into account on the final grid, e.g. a `loop_distance` of 1 uses 3 pixels ( $x - 1, x, x + 1$ ) in each dimension, i.e. 9 in total for a 2D image and 27 in total for a 3D cube. Moreover, if the parameter `use_errorweights` is set to `TRUE`, an additional weight defined as  $1/\text{variance}$  is taken into account<sup>12</sup>.

## 4.14.1.6 Lanczos

The algorithm uses a restricted *sinc* distance weighting function ( $\frac{\text{sinc}(r)}{\text{sinc}(r/\text{kernel\_size})}$ ) with the kernel size given by the parameter `kernel_size` for the interpolation. The parameter `loop_distance` controls the number of surrounding pixels that are taken into account on the final grid, e.g. a `loop_distance` of 1 uses 3 pixels ( $x - 1, x, x + 1$ ) in each dimension, i.e. 9 in total for a 2D image and 27 in total for a 3D cube. Moreover, if



the parameter `use_errorweights` is set to `TRUE`, an additional weight defined as  $1/\text{variance}$  is taken into account<sup>12</sup>.

## 4.14.2 Functions

The input data is resampled using the following function call

```
hdrl_resample_result * hdrl_resample_compute(  
    const cpl_table      * restable,  
    hdrl_parameter      * method,  
    hdrl_parameter      * outputgrid,  
    const cpl_wcs       * wcs);
```

The routine is not directly working on an image or a cube but on a table (`restable`). For 2D images, the table is created by the function `hdrl_resample_image_to_table()`, whereas for a 3D data cube the function `hdrl_resample_imagelist_to_table()` generates the appropriate table. In the case that many images/cubes have to be combined into a single mosaic, the two functions can be called multiple times and the returned tables should be merged into a single table by using `cpl_table_insert()`.

## 4.14.3 Inputs

The `hdrl_resample_compute` function has four arguments:

### **restable**

The first function argument `restable` is a `cpl_table` with information on the data to be resampled. It can be created in case of a 2D image or a 3D data cube with the following function call:

```
cpl_table * hdrl_resample_image_to_table(  
    const hdrl_image      * hima,  
    const cpl_wcs         * wcs);
```

```
cpl_table * hdrl_resample_imagelist_to_table(  
    const hdrl_imagelist * himlist,  
    const cpl_wcs        * wcs);
```

where `hima` is a `hdrl` image containing the 2D image and `himlist` a `hdrl` imagelist containing the 3D cube that should be resampled. The `wcs` is a `cpl wcs` object that encodes the world coordinate system of the given image/cube.

In case the above mentioned functions can not be used to create the table (e.g. the pixel to sky mapping is very complex and can not be encoded by the `cpl wcs` object) the pipeline developer has to create and fill the table. The table should be created and initialised as follows:

<sup>12</sup>Please note that only if the variance of a pixel is  $> 0$  this additional weight is applied



```
tab = cpl_table_new(size);

/* create the table columns */
cpl_table_new_column(tab, HDRL_RESAMPLE_TABLE_RA, HDRL_RESAMPLE_TABLE_RA_TYPE);
cpl_table_new_column(tab, HDRL_RESAMPLE_TABLE_DEC, HDRL_RESAMPLE_TABLE_DEC_TYPE);
cpl_table_new_column(tab, HDRL_RESAMPLE_TABLE_LAMBDA, HDRL_RESAMPLE_TABLE_LAMBDA_TYPE);
cpl_table_new_column(tab, HDRL_RESAMPLE_TABLE_DATA, HDRL_RESAMPLE_TABLE_DATA_TYPE);
cpl_table_new_column(tab, HDRL_RESAMPLE_TABLE_BPM, HDRL_RESAMPLE_TABLE_BPM_TYPE);
cpl_table_new_column(tab, HDRL_RESAMPLE_TABLE_ERRORS, HDRL_RESAMPLE_TABLE_ERRORS_TYPE);

/* init column values */
cpl_table_fill_column_window_double(tab, HDRL_RESAMPLE_TABLE_RA, 0, size, 0.);
cpl_table_fill_column_window_double(tab, HDRL_RESAMPLE_TABLE_DEC, 0, size, 0.);
cpl_table_fill_column_window_double(tab, HDRL_RESAMPLE_TABLE_LAMBDA, 0, size, 0.);
cpl_table_fill_column_window_double(tab, HDRL_RESAMPLE_TABLE_DATA, 0, size, 0.);
cpl_table_fill_column_window_int(tab, HDRL_RESAMPLE_TABLE_BPM, 0, size, 0);
cpl_table_fill_column_window_double(tab, HDRL_RESAMPLE_TABLE_ERRORS, 0, size, 0.);
```

Please note that the interpolation function `hdrl_resample_compute()` was only tested with tables created by `hdrl_resample_image_to_table()` or `hdrl_resample_imagelist_to_table()`, i.e. mappings encoded by a `cpl wcs` object. Creating and using a handcrafted table could lead to side effects that the interpolation function cannot properly handle.

## method

Currently six different interpolation methods can be used to resample the data. The corresponding `hdrl` parameter method is created executing the appropriate function call:

- **Nearest:** Nearest neighbour resampling (see section 4.14.1.1)

```
hdrl_parameter * hdrl_resample_parameter_create_nearest(void);
```

- **Linear:** Weighted resampling using inverse distance weighting function (see section 4.14.1.2)

```
hdrl_parameter * hdrl_resample_parameter_create_linear(
    const int    loop_distance,
    cpl_boolean  use_errorweights);
```

`loop_distance`: in pixel units ( $\geq 0$ )

`use_errorweights`: `cpl_boolean` (`CPL_TRUE` or `CPL_FALSE`)

- **Quadratic:** Weighted resampling using quadratic inverse distance weighting function (see section 4.14.1.3)

```
hdrl_parameter * hdrl_resample_parameter_create_quadratic(
    const int    loop_distance,
    cpl_boolean  use_errorweights);
```

`loop_distance`: in pixel units ( $\geq 0$ )

`use_errorweights`: `cpl_boolean` (`CPL_TRUE` or `CPL_FALSE`)



- **Renka:** Weighted resampling using Renka weighting function (see section 4.14.1.4)

```
hdrl_parameter * hdrl_resample_parameter_create_renka (  
    const int    loop_distance,  
    cpl_boolean  use_errorweights,  
    const double critical_radius);
```

loop\_distance: in pixel units ( $\geq 0$ )  
use\_errorweights: cpl\_boolean (CPL\_TRUE or CPL\_FALSE)  
critical\_radius: in pixel units ( $> 0$ )

- **Drizzle:** Weighted resampling using a drizzle-like weighting scheme (see section 4.14.1.5)

```
hdrl_parameter * hdrl_resample_parameter_create_drizzle (  
    const int    loop_distance,  
    cpl_boolean  use_errorweights,  
    const double pix_frac_x,  
    const double pix_frac_y,  
    const double pix_frac_lambda);
```

loop\_distance: in pixel units ( $\geq 0$ )  
use\_errorweights: cpl\_boolean (CPL\_TRUE or CPL\_FALSE)  
pix\_frac\_x: fraction of pixel/voxel ( $> 0$ )  
pix\_frac\_y: fraction of pixel/voxel ( $> 0$ )  
pix\_frac\_lambda: fraction of pixel/voxel ( $> 0$ )

- **Lanczos:** Weighted resampling using a lanczos-like restricted sinc for weighting (see section 4.14.1.6)

```
hdrl_parameter * hdrl_resample_parameter_create_lanczos (  
    const int    loop_distance,  
    cpl_boolean  use_errorweights,  
    const int    kernel_size);
```

loop\_distance: in pixel units ( $\geq 0$ )  
use\_errorweights: cpl\_boolean (CPL\_TRUE or CPL\_FALSE)  
kernel\_size: in pixel units ( $> 0$ )

## outputgrid

The `outputgrid` parameter defines basic properties of the resampled image/cube. Depending on the input data (image or cube) the `outputgrid` parameter should be created with the appropriate function call, i.e. `hdrl_resample_parameter_create_outgrid2D_userdef()` for an image and `hdrl_resample_parameter_create_outgrid3D_userdef()` for a cube. For convenience, the `outputgrid` parameter can also be created with `hdrl_resample_parameter_create_outgrid2D()` or `hdrl_resample_parameter_create_outgrid3D()`. In this case, only the step sizes in right ascension (`delta_ra`), declination (`delta_dec`) and wavelength (`delta_lambda`) of the output image/cube have to be given. All the rest is automatically derived from the data by the `hdrl_resample_compute()` function. The four functions read as follows:



```
hdrl_parameter * hdrl_resample_parameter_create_outgrid2D_userdef (  
    const double delta_ra,  
    const double delta_dec,  
    const double ra_min,  
    const double ra_max,  
    const double dec_min,  
    const double dec_max,  
    const double fieldmargin);
```

```
hdrl_parameter * hdrl_resample_parameter_create_outgrid3D_userdef (  
    const double delta_ra,  
    const double delta_dec,  
    const double delta_lambda,  
    const double ra_min,  
    const double ra_max,  
    const double dec_min,  
    const double dec_max,  
    const double lambda_min,  
    const double lambda_max,  
    const double fieldmargin);
```

where `delta_ra`, `delta_dec`, and `delta_lambda` are the output grid steps in right ascension, declination and wavelength. Moreover, these two functions allow the user to set the full output grid by defining `ra_min`, `ra_max`, `dec_min`, `dec_max`, and `lambda_min`, `lambda_max`. These parameters define the minimum and maximum boundaries of the output image in terms of right ascension, declination and wavelength. Furthermore, with the parameter `fieldmargin` one can add a margin to the output image in all spacial directions.

The parameters `delta_ra`, `delta_dec`, `ra_min`, `ra_max`, `dec_min`, and `dec_max` have to be given in degrees [deg], whereas the parameters `delta_lambda`, `lambda_min`, and `lambda_max` have to be given in meter [m]. The `fieldmargin` has to be given in percent with 0 adding no margin to the output image/cube

For convenience, the *High Level Data Reduction Library* also offers the following convenience functions that calculate the boundaries of the output image/cube from the data and automatically set the `fieldmargin`:

```
hdrl_parameter * hdrl_resample_parameter_create_outgrid2D (  
    const double delta_ra,  
    const double delta_dec);
```

```
hdrl_parameter * hdrl_resample_parameter_create_outgrid3D (  
    const double delta_ra,  
    const double delta_dec,  
    const double delta_lambda);
```



## wcs

This input parameter specifies the World Coordinate System which should be representative of the images that are going to be resampled. The resampling functions uses the wcs input (CD matrix) mostly do determine the scales between the input and output grid. Please note, that in case the user would like to combine images/cubes with substantially different pixel sizes into a single output image/cube, the single tables have to be properly scaled to the same scales before merging them into the final table.

### Pixel scale computation (PIPE-12792 fix)

When extracting pixel scales (`delta_ra`, `delta_dec`) from the WCS CD matrix, the library now automatically ensures correct computation accounting for rotation. For rotated WCS systems (with significant off-diagonal CD matrix terms), pixel scales must be computed from the full CD matrix using the magnitude of transformation vectors:  $\delta_{ra} = \sqrt{CD_{1,1}^2 + CD_{1,2}^2}$  and  $\delta_{dec} = \sqrt{CD_{2,1}^2 + CD_{2,2}^2}$ , rather than using only diagonal elements. The library proactively computes correct pixel scales from the full CD matrix and automatically replaces incorrectly computed pixel scales (e.g., when only diagonal elements were used) before computing output grid dimensions. This prevents unreasonably large output grid dimensions and memory allocation failures. The fix ensures robust handling of rotated WCS systems while preserving user intent for intentional oversampling or undersampling.

### 4.14.4 Outputs

The `hdrl_resample_compute()` function creates an output `hdrl_resample_result` structure containing the following quantities:

```
typedef struct {
    /* cpl propertylist containing the output wcs information */
    cpl_propertylist * header;
    /* hdrl imagelist containing the data, the errors, and the bpm */
    hdrl_imagelist * himlist;
} hdrl_resample_result;
```

This structure contains for convenience a cpl propertylist `header` with the output wcs that can directly be used by the cpl saving functions to save the image/cube. In case the wcs object itself is needed, the cpl function `cpl_wcs_new_from_propertylist(header)` should be used. Moreover, the structure contains the resampled image/cube as `hdrl_imagelist` `himlist`. If a 2D image was resampled, the returned imagelist is of size 1 and the resampled hdrl image can be accessed with `hdrl_imagelist_get(himlist, 0)`.

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

Please note that the output is using the Gnomonic projection<sup>13</sup>. The gnomonic projection is from the centre of a sphere to a plane tangential to the sphere. The sphere and the plane touch at the tangent point. This is encoded in the header file by:

<sup>13</sup>see [https://en.wikipedia.org/wiki/Gnomonic\\_projection](https://en.wikipedia.org/wiki/Gnomonic_projection)



# HDRL Pipeline Developer Manual

Doc. Number: ESO-299492  
Doc. Version: 7  
Sof. Version: 1.6.0  
Released on: 2026-06-02  
Page: 90 of 173

---

CTYPE1	=	'RA---TAN'	/ Gnomonic projection
CTYPE2	=	'DEC--TAN'	/ Gnomonic projection



#### 4.14.5 Example

This section briefly describes a typical cube regridding example .

```
/* Load input data, errors and bpm frames */
cpl_imagelist* imlist_data = cpl_imagelist_load(name, CPL_TYPE_DOUBLE, extension_data);
cpl_imagelist* imlist_error = cpl_imagelist_load(name, CPL_TYPE_DOUBLE, extension_error);
cpl_imagelist* imlist_bpm = cpl_imagelist_load(name, CPL_TYPE_INT, extension_bpm);

/* Load input wcs */
cpl_propertylist *xheader_data = cpl_propertylist_load(name, extension_data);
cpl_wcs *wcs = cpl_wcs_new_from_propertylist(xheader_data);
cpl_propertylist_delete(xheader_data);

/* Add the bad pixel mask to the corresponding cpl image */
cpl_size size = cpl_imagelist_get_size(imlist_data);
for(cpl_size k = 0; k < size; k++) {
    cpl_image* data = cpl_imagelist_get(imlist_data, k);
    cpl_image* bpm = cpl_imagelist_get(imlist_bpm, k);
    cpl_mask* mask = cpl_mask_threshold_image_create(bpm, 0, INT_MAX);
    cpl_image_reject_from_mask(data, mask);
    cpl_mask_delete(mask);
    cpl_imagelist_set(imlist_data, data, k);
}

/* Create the hdrl imagelist and free memory */
hdrl_imagelist* hlist = NULL;
hlist = hdrl_imagelist_create(imlist_data, imlist_error);
hdrl_imagelist_delete(imlist_data);
hdrl_imagelist_delete(imlist_error);
hdrl_imagelist_delete(imlist_bpm);

/* store the imagelist information into the cpl_table and free memory */
cpl_table* restable = hdrl_resample_imagelist_to_table(hlist, wcs);
hdrl_imagelist_delete(hlist);

/* Define the output grid */
hdrl_parameter *outgrid = NULL;
outgrid = hdrl_resample_parameter_create_outgrid2D(5.5e-05, 5.5e-05, 2.8e-10);

/* define the method - here LANCZOS */
hdrl_parameter *method = NULL;
method = hdrl_resample_parameter_create_lanczos(1, CPL_FALSE, 2);

/* Do the resampling */
hdrl_resample_result *result = NULL;
result = hdrl_resample_compute(restable, method, outgrid, wcs);

/* save the relevant results */
...
/* Cleanup the memory */
hdrl_parameter_delete(method);
hdrl_parameter_delete(outgrid);
hdrl_resample_result_delete(result);
cpl_wcs_delete(wcs);
cpl_table_delete(restable);
```



## 4.14.6 Additional Information

As mentioned above, the `hdrl_resample_compute` function requires as an input a cpl table and not `hdrl` images or an `hdrl` imagelist. This choice was made for flexibility but comes with a cost on the memory consumption. Moreover, the current implementation does not distinguish between a 2D image and a 3D cube in the table structure, i.e. the wavelength column is always filled. Furthermore, cpl functions are usually used to read the input images and they are then inserted into a `hdrl` image, which implies a small overhead.

We can understand the requirements in term of memory consumption as follows. In the typical use case the developer has input three cubes: one containing the data, stored in a `float`<sup>14</sup> data type, one the error, stored in a `float` data type, and the bad pixels, stored in an `integer`<sup>15</sup> data type. Then the `cpl_imagelist` objects are inserted into a `hdrl` imagelist which has the data and error in double precision and each has a `bpm` in `unsigned char`<sup>16</sup> precision [see step 2. below]. Then the information of the `hdrl` imagelist is stored into a cpl table [see step 3. below]. Finally one has to define and fill the `hdrl` imagelist containing the final resampled image/cube [see step 4. below]. The size of the resampled image [step 4. below] strongly depends on the use-case: i.e. if the input images/cubes are mostly stacked, or if a mosaic is created and, if a mosaic is created, its size strongly depends on possible gaps between the CCDs.

1. `cpl_imagelist` (read the raw data  $x, y, \lambda$ )
  - data (`float`) + error (`float`) + bpm (`int`) [**requires**  $\approx 3 \times float$ ]
2.  $\Rightarrow$  `hdrl_imagelist` (join the raw data/error/bpm)
  - $\Rightarrow$  data ( $2 \times float$ ) + error ( $2 \times float$ ) + bpm ( $2 \times unsigned\ char \approx 0.5 \times float$ ) [**requires**  $\approx 4.5 \times float$ ]
3.  $\Rightarrow$  `cpl_table` (transform raw data  $x, y, \lambda \rightarrow Ra, Dec, \lambda$ )
  - Ra ( $2 \times float$ ) + Dec ( $2 \times float$ ) +  $\lambda$  ( $2 \times float$ ) + data ( $2 \times float$ ) + error ( $2 \times float$ ) + bpm (`int`) [**requires**  $\approx 11 \times float$ ]
4.  $\Rightarrow$  `hdrl_imagelist` (resample Ra, Dec,  $\lambda \rightarrow x, y, \lambda$ )
  - $\Rightarrow$  data ( $2 \times float$ ) + error ( $2 \times float$ ) + bpm ( $2 \times unsigned\ char \approx 0.5 \times float$ ) [**requires**  $\approx 4.5 \times float$ ]

After reading the data [step 1.] and inserting the data into an `hdrl` imagelist [step 2.] the data of step 1. can be deleted. The total image consumption of the transition of step 2 and step 3 depends on the data. If you have multi extension frames like for VIRCAM you can populate the table extension by extension and therefore you are mostly dominated by the size of your table. If you can not work extension by extension, e.g. if your input is a MUSE cube you can delete the `hdrl` imagelist of step 2 only after you fully populate the cpl table of step 3. This means that you have to keep the data involved in step 2 and step 3 in memory at the same time<sup>17</sup>. Then, once the table is filled, you may delete the `hdrl` imagelist created in step 2. As written above, the size of the

<sup>14</sup>usually 4 bytes

<sup>15</sup>usually 4 bytes

<sup>16</sup>usually 1 bytes

<sup>17</sup>see section 5.1 for a possibility to address this problem



resampled image [step 4.] strongly depends on the use-case so here we assume that the resampled image is a mosaic without gaps in between (no stacking). Then for transition between steps 3 and 4 one needs to have the table and the resampled output image in memory at the same time.

Usually, memory (RAM) requirements are dominated by the data allocated in step 2 + 3 or step 3 + 4. This means that as a rule of thumb one would need [ $\approx 11 + 4.5 = 15.5 \times \text{float} \times \text{number of input pixels}$ ] of memory.

This shows that if your data requires a certain amount of RAM to be stored as raw data you need **at least 6 times more memory**:  $3 \times \text{float} \Rightarrow 15.5 \times \text{float}$ , to perform the corresponding resampling. Please note, that in case you do not have error images/cubes and/or bpm images/cubes in input you have to assume that they also would be there, when you calculate the maximum amount of memory consumption as previously described. Even if you pass NULL to the `hdrl_imagelist_create()` function, it will then create an error and bpm image on the fly. In other words, if you only have information on the input data (no error and no bpm), you need **at least 18 times more memory** to do the resampling.

## 4.15 Limiting Magnitude

The limiting magnitude is one of the mandatory header keywords in the Phase 3 archival standard<sup>18</sup> and characterizes the depth of an observation. According to the ESO Phase 3 standard, it is defined as the magnitude of a unresolved source whose flux is 5 times the noise background, i.e. the magnitude of a point like source detected with  $\frac{S}{N} = 5$ .

### 4.15.1 Algorithm - short description

The limiting magnitude of an image is determined as follows:

- The input image `image` is convolved with a Gaussian kernel specified by the input parameters `fwhm` (the FWHM of the kernel), `kernel_size_x`, and `kernel_size_y` (the extension of the kernel in x-direction and y-direction in pixel). To minimize border effects/artifacts during convolution, the input image gets enlarged by a border depending on the kernel size. The parameter `image_extend_method` can be used to specify the algorithm to compute the values of the out-of-bounds pixels during convolution (`HDRL_IMAGE_EXTEND_NEAREST` → use the nearest border pixel, `HDRL_IMAGE_EXTEND_MIRROR` → mirror the image with respect to the border). Masked pixels are ignored in the convolution, i.e. the kernel is normalized ignoring masked pixels.
- Compute the `mode` as described in section B.4 of the convolved image.
- Consider all the non-masked pixels in the convolved image below the `MODE` (let's define them as valid pixels), and compute the noise as:

$$NOISE = 1.4826 \cdot MAD(\text{valid pixels}) \cdot c$$

The factor  $c = \frac{1}{\sqrt{1-\frac{2}{\pi}}} \approx 1.6588967$  accounts for the fact that HDRL computes the statistics on half of the distribution (assuming a Gaussian noise distribution), MAD is the median absolute deviation, and

<sup>18</sup>see <https://www.eso.org/sci/observing/phase3.html> for the ESO Phase 3 Standard



1.4826 is the scaling factor to convert the MAD in the standard deviation (for a Gaussian distribution). If  $MAD=0$ , then:

$$NOISE = STDDEV(\text{valid pixels}) \cdot c$$

where STDEV indicates the standard deviation.

- Compute the limiting magnitude as:

$$ABMAGLIM = -2.5 \log_{10}(5 \cdot NOISE \cdot 4\pi\sigma^2) + ZPT$$

$$\text{with } \sigma = \frac{FWHM}{\sqrt{4 \ln(4)}} \approx \frac{FWHM}{2.35482}$$

## 4.15.2 Functions

The *High Level Data Reduction Library* implements the following function to derive the limiting magnitude:

```
cpl_error_code hdrl_maglim_compute(  
    cpl_image          * image,  
    double             zeropoint,  
    double             fwhm,  
    cpl_size           kernel_size_x,  
    cpl_size           kernel_size_y,  
    hdrl_image_extend_method image_extend_method,  
    hdrl_parameter     * mode_parameter,  
    double             * limiting_magnitude);
```

## 4.15.3 Inputs

The image where the limiting magnitude should be derived and its zeropoint is passed to the function by the `image` and the `zeropoint` argument.

The `fwhm` parameter sets the full width at half maximum of the Gaussian convolution kernel whereas the extension of the kernel in x-direction and y-direction (in pixel) is set by `kernel_size_x`, and `kernel_size_y`, respectively. The parameter `image_extend_method` specifies the algorithm to compute the values of the out-of-bounds pixels during convolution (`HDRL_IMAGE_EXTEND_NEAREST` → use the nearest border pixel, `HDRL_IMAGE_EXTEND_MIRROR` → mirror the image with respect to the border).

The `hdrl` parameter `mode_parameter` controls the computation of the mode of the distribution and is created by the `hdrl_collapse_mode_parameter_create()` function. The parameter of this function (`histo_min`, `histo_max`, `bin_size`, `mode_method`, and `error_niter`) are described in section [B.4](#)

Finally, the computed limiting magnitude is returned as `limiting_magnitude`



## 4.15.4 Outputs

The result of the function is the limiting magnitude of the image stored in the function parameter `limiting_magnitude`

## 4.15.5 Example

The following code is a example how to use HDRL to compute the limiting magnitude from an image.

```
/* initialize the parameters of a the mode computation */  
  
double histo_min = 10.;  
double histo_max = 1.; /* min > max -> autoset by algorithm */  
double bin_size = 0.; /* bin_size <= 0 -> autoset by algorithm */  
cpl_size error_niter = 0; /* analytic error */  
hdrl_mode_type mode_method = HDRL_MODE_MEDIAN; /* very robust */  
  
hdrl_parameter * mode_parameter = NULL;  
mode_parameter = hdrl_collapse_mode_parameter_create(histo_min,  
                                                    histo_max,  
                                                    bin_size,  
                                                    mode_method,  
                                                    error_niter);  
  
/* initialize other parameters of the limiting magnitude computation */  
  
double zeropoint = 24.3;  
double fwhm = 3.50;  
cpl_size kernel_size_x = 10; /* about 3 times fwhm */  
cpl_size kernel_size_y = 10; /* about 3 times fwhm */  
hdrl_image_extend_method convolution_boundary = HDRL_IMAGE_EXTEND_MIRROR;  
double limiting_magnitude = 0.;  
  
/* compute the limiting magnitude */  
  
hdrl_maglim_compute(image, zeropoint, fwhm, kernel_size_x,  
                   kernel_size_y, convolution_boundary,  
                   mode_parameter, &limiting_magnitude);
```



## 4.16 Barycentric correction

The algorithm derives the barycentric correction of an observation, i.e. the wavelength shift to apply to a spectrum to compensate for the motion of the observer with respect to the barycenter of the solar system, by using the [ERFA](#) (Essential Routines for Fundamental Astronomy) library. ERFA is a C library containing key algorithms for astronomy, and is based on the [SOFA library](#) published by the International Astronomical Union (IAU).

### 4.16.1 Algorithm - short description

The implemented algorithm uses the ERFA function [eraApc0130](#) to calculate the barycentric correction of an observation. For details on the implemented low level algorithm please visit the above mentioned web pages and links therein.

A comparison between the algorithm implemented in the *High Level Data Reduction Library* and the one implemented in the ESPRESSO pipeline shows a very good agreement. For this about 7000 IDPs from 2021 were re-analyzed with the *High Level Data Reduction Library* implementation and the differences read as follows:

- Mean difference : 0.036 m/s
- Median difference: 0.055 m/s
- Standard deviation: 0.281 m/s
- MAD (median absolute deviation): 0.217 m/s
- Maximum deviation: 1.573 m/s

### 4.16.2 Functions

The *High Level Data Reduction Library* implements the following function to derive the barycentric correction:

```
cpl_error_code hdrl_barycorr_compute (  
    double          ra,  
    double          dec,  
    const cpl_table * eop_table,  
    double          mjdots,  
    double          time_to_mid_exposure,  
    double          longitude,  
    double          latitude,  
    double          elevation,  
    double          pressure,  
    double          temperature,  
    double          humidity,  
    double          wavelength ,  
    double          * barycorr);
```



## 4.16.3 Inputs

The different **input** function parameters are described as follows:

- `ra`: Target right ascension (J2000) in [deg]
- `dec`: Target declination (J2000) in [deg]
- `eop_table`: `cpl_table` containing the earth orientation parameter
- `mjdobs`: Start of observation in [days]
- `time_to_mid_exposure`: Time to mid exposure, e.g. `EXPTIME/2.` in [s]
- `longitude`: Telescope geodetic longitude (+ = East ) in [deg]
- `latitude`: Telescope geodetic latitude (+ = North) in [deg]
- `elevation`: Telescope elevation above sea level in [m]
- `pressure`: Pressure at the observer in [hPa == mbar]
- `temperature`: Ambient temperature at the observer in [deg C]
- `humidity`: Relative humidity at the observer [range 0 - 1]
- `wavelength`: Observing wavelength in [micrometer]

Please note that the pressure, temperature, humidity, and the wavelength parameters are only tested with a value of 0. No tests with other values were performed.

The `eop_table` contains the [Earth Orientation Parameter](#) as a function of time (`MJD-OBS`) and can be downloaded from [IERS](#). For this purpose the *High Level Data Reduction Library* offers the 2 functions `hdrl_download_url_to_buffer()` and `hdrl_eop_data_totable()`. The first function allows you to download the full EOP table from IERS and the second function extracts and converts the relevant information into a fits table. The latter contains the appropriate columns and can be passed to the `hdrl_barycorr_compute()` function.

```
char * hdrl_download_url_to_buffer (  
    const char * url,  
    size_t      * data_length);
```

This function allows you to download a url into a c data buffer with `url` being the url to download from. The length of the buffer is stored in the function parameter `data_length`. The url to download the EOP data should be `https://datacenter.iers.org/products/eop/rapid/standard/finals2000A.data`

Please note, that this function is not threadsafe, to the extent that it may only be called from the main thread, with no other threads running. So as long as `esorex` or alike is not using threads it still may be called from within a recipe before the the recipe itself, or `hdrl` launches any additional threads.

```
cpl_table * hdrl_eop_data_totable (  
    const char * eop_data,  
    cpl_size    data_length);
```

From the above listed url<sup>19</sup>, this function extracts the relevant information from the c data buffer and converts it

<sup>19</sup><https://datacenter.iers.org/products/eop/rapid/standard/finals2000A.data>



into a fitsfile. In this function, the `eop_data` is the downloaded data buffer and `data_length` the length of the buffer.

As the time resolution of the EOP table is only one day, the parameters are linearly interpolated to have the most accurate values at the time of observation.

#### 4.16.4 Outputs

The **computed barycentric correction** is stored in the function parameter `barycorr` with the units [m/s].

#### 4.16.5 Example

The following code is a skeleton example on how to use HDRL to compute the barycentric correction of an observation. For a detailed implementation please see the `esotk_barycorr` recipe for the barycentric correction computation and the `esotk_eop` recipe for the EOP table creation.

```
/* Load EOP file - it was created by another dedicated recipe */
/* For detail on how to create the EOP file see the esotk_eop recipe */

cpl_table * eop_table =
    cpl_table_load(cpl_frame_get_filename(eop_frame), 1, 0);

/* Read the primary header */
cpl_propertylist * header =
    cpl_propertylist_load(cpl_frame_get_filename(cur_frame), 0);

double ra          = cpl_propertylist_get_double(header, "RA");
double dec         = cpl_propertylist_get_double(header, "DEC");
double mjdots     = cpl_propertylist_get_double(header, "MJD-OBS");
double exptime    = cpl_propertylist_get_double(header, "EXPTIME");
double longitude  = cpl_propertylist_get_double(header, "ESO_TEL_GEOLON");
double latitude   = cpl_propertylist_get_double(header, "ESO_TEL_GEOLAT");
double elevation  = cpl_propertylist_get_double(header, "ESO_TEL_GEOELEV");
double pressure   = 0.0
double temperature = 0.0
double humidity   = 0.0
double wavelength = 0.0

double barycorr = 0;
hdrl_barycorr_compute(ra, dec, eop_table, mjdots, exptime/2.,
                    longitude, latitude, elevation,
                    pressure, temperature, humidity, wavelength,
                    &barycorr);

cpl_msg_info (cpl_func, "Barycentric_correction:_%g_m/s", barycorr);
```



## 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 4GiB 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);
```



## 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



## B Statistical Estimators

### B.1 Arithmetic mean - Arithmetic weighted mean

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

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i. \quad (14)$$

For a random variable  $X$  that is normally distributed ( $N(\mu, \sigma)$ ), the sample mean is the most efficient estimator of the population mean  $\mu$ . 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

$$\hat{\sigma}^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2. \quad (15)$$

If  $X$  has variance  $\sigma$  then the average (14) has variance

$$\sigma_{\text{av}}^2 = \frac{\sigma^2}{N}. \quad (16)$$

This relation can be used in place of Eq. (15) if an *a priori* estimate of the noise in the images is available and is the same for all images (homoscedastic case).

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

$$\bar{x} = \left( \sum \frac{x_i}{\sigma_i^2} \right) \left( \sum \frac{1}{\sigma_i^2} \right)^{-1} \quad (17)$$

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

$$\frac{1}{\sigma_{\text{av}}^2} = \sum_{i=1}^N \frac{1}{\sigma_i^2}. \quad (18)$$

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.

### B.2 Median

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

$$x_{\text{med}} = x_{\frac{N+1}{2}} \quad \text{if } N \text{ is odd} \quad (19)$$

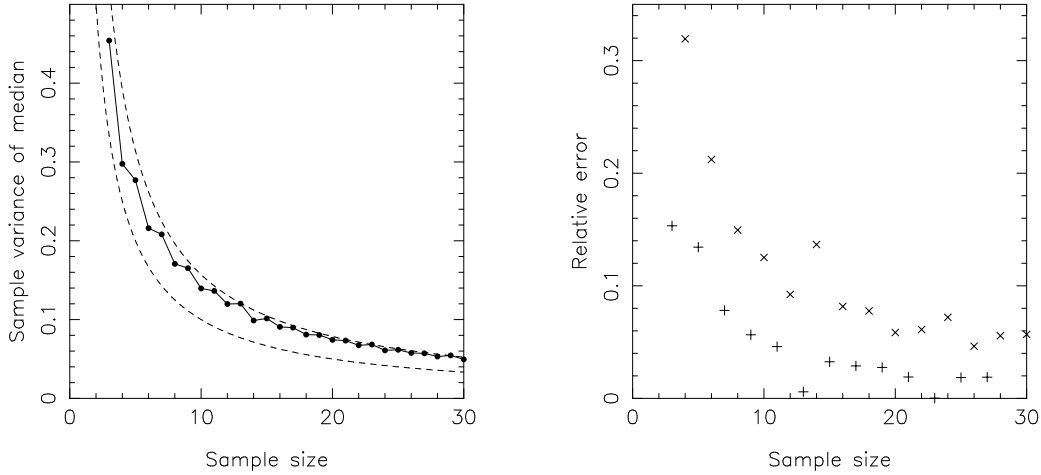


Figure B.2.1: Sample variance of the median as a function of sample size (left panel). The lower dashed curve is the expected variance of the arithmetic mean, the upper dashed curve the asymptotic variance of the median. The right panel shows the relative error compared to the asymptotic variance. Crosses are for even sample sizes, pluses for odd sample sizes.

and

$$x_{\text{med}} = \frac{1}{2}(x_{\frac{N}{2}} + x_{\frac{N}{2}+1}) \quad \text{if } N \text{ is even.} \quad (20)$$

This definition for the median for even-sized samples ensures that the median is an unbiased estimator of the population mean  $\mu$  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 [7] and exact values for a few sample sizes and a number of parent distributions are given in [9]. 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, [3]) in the Gaussian case is

$$\sigma_{\text{med}}^2 = \frac{\pi}{2} \sigma_{\text{av}}^2 = \frac{\pi \sigma^2}{2N}. \quad (21)$$

In Fig. B.2.1, 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

$$\frac{\pi/2N - \hat{\sigma}_{\text{med}}^2}{\pi/2N}, \quad (22)$$

is plotted in the right hand panel of Fig. B.2.1. 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 [9]), whereas for even sized samples this threshold is not crossed until  $N \gtrsim 12$ . However, as mentioned above, the asymptotic value in Eq. (21) 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

$$x_{\text{wmed}} = \begin{cases} \frac{x_j + x_{j+1}}{2}, & \text{if } \sum_{i=1}^j w_i = 0.5 \\ x_j, & \text{if } \sum_{i=1}^j w_i > 0.5 \quad \text{and} \quad \sum_{i=1}^{j-1} w_i < 0.5 \end{cases} \quad (23)$$

### 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  $\kappa$  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  $\sigma$  is given in terms of the MAD through

$$\sigma = \text{MAD} \times 1.4826. \quad (24)$$

$\kappa\sigma$  clipping has one parameter, the clipping threshold  $\kappa$ . The value of  $\kappa$  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  $\kappa$ .

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

$$\sigma_{\kappa\sigma}^2 = \frac{1}{N_{\text{good}}^2} \sum_{\text{good}} \sigma_i^2. \quad (25)$$

## B.4 Mode

The mode (or modal value) of a distribution of values is defined as the value of the distribution that appears more often (the maximum of the function distribution). The *High Level Data Reduction Library* implements the mode computation for the case of a discrete distribution.

In the case of a discrete distribution, one can define the “modal class” (the position of the highest peak of the histogram), and the “modal frequency” (the value of the highest peak of the histogram), which are related to the mode of the underlying continuous distribution. The value of the modal frequency depends from the chosen bin size used to create the histogram and from statistical noise. A large bin size helps in finding a high and unique modal frequency, but decreases the resolution. A small bin increases the resolution of the modal class but decreases the modal frequency and therefore increases the chance that, because of the statistical noise, the highest peak in the histogram is not related to the mode of the underlying distribution.

Concerning the determination of the error associated to the mode estimator, the mode method is special as HDRL **calculates the error from the data** and is not doing pure **error propagation** as all the other statistical estimators listed above. The error calculation depends on the parameter `error_niter`. If the parameter is set to 0, HDRL estimates the error analytically as described below. If the parameter is larger than 0, the error is calculated by a bootstrap Montecarlo simulation from the input data with the value of the parameter specifying the number of simulations. In this case the input data are perturbed with the bootstrap technique `error_niter` times and the mode is derived for each simulation. From this modes the standard deviation is calculated and returned as error.

The *High Level Data Reduction Library* implementation in the supported case of discrete distributions depends on the following parameters:

- `histo_min`: This parameter defines the lower bound of the histogram, i.e. the minimum value of the pixels that are used to derive the histogram<sup>20</sup>. Please note that if the lower bound (`histo_min`) is set to the same or to a higher value than the upper bound (`histo_max`), both parameters are automatically determined from the data by taking the minimum value for `histo_min` and the maximum value for `histo_max`.<sup>21</sup>
- `histo_max`: This parameter defines the upper bound of the histogram, i.e. the maximum value of the pixels that are used to derive the histogram<sup>20</sup>. Please note that if the upper bound (`histo_max`) is set to the same or to a lower value than the lower bound (`histo_min`), both parameters are automatically determined from the data by taking the minimum value for `histo_min` and the maximum value for `histo_max`.<sup>22</sup>

<sup>20</sup>If `histo_min < histo_max`, `nbin = floor((histo_max - histo_min) / bin_size) + 1`, data points outside the range [`histo_min`, `histo_max`] are removed and, to be consistent with GSL, `histo_min` is kept to the user value and `histo_max = histo_min + nbin * bin_size`

<sup>21</sup>If `histo_min >= histo_max`, the real minimum internally used is always `histo_min - bin_size / 2`

<sup>22</sup>If `histo_min >= histo_max`, the real maximum internally used is always `histo_max + bin_size / 2`



- `bin_size`: This parameter defines the size of the histogram bin. Please note that if the value is set to value less or equal to 0, the parameter is automatically determined from the data.
- `mode_method`: This parameter defines the method used to compute the mode. Currently there are three methods implemented with an enum:
  - `HDRL_MODE_MEDIAN`: This method is the most robust method and can/should be used for very asymmetric point distributions. For more details see below.
  - `HDRL_MODE_WEIGHTED`: This method can/should be used with asymmetric distributions. For more details see below.
  - `HDRL_MODE_FIT`: This method give accurate results for symmetric distributions but is also more likely to fail. For more details see below.
- `error_niter`: This parameter defines the algorithm to compute the error of the mode as, contrary to all the other implemented statistical estimators, the mode **calculates the error from the data** and not by **error propagation**. Moreover, it is also special in the sense that one can choose between an analytical error estimation and one based on a bootstrap simulation. If the parameter is set to 0, HDRL is doing an analytical error estimation. If the parameter is larger than 0, the error is obtained by a bootstrap Montecarlo simulation from the input data with the value of the parameter specifying the number of simulations. In this case the input data are perturbed with the bootstrap technique and the mode is calculated `error_niter` times. From this modes the standard deviation is determined and returned as error.

In details, the implemented algorithm is performing the following operations:

1. Pixels marked as bad (bpm) are removed from the input data.
2. HDRL derives/sets the size of the histogram bins (binsize) as follows:
  - If `bin_size` is set to 0, then binsize is automatically defined as:
$$\text{binsize} = 2 \cdot 3.49 \cdot \Sigma / N_{\text{TOT}}^{1/3}$$
where
$$\Sigma = \begin{cases} \text{MAD}(X) / .67449 & , \text{if } \text{MAD}(X) \neq 0 \\ \text{STDDEV}(X) & , \text{if } \text{MAD}(X) = 0 \end{cases}$$
and  $N_{\text{TOT}}$  is the number of elements of  $X$ . MAD is the Median Absolute Deviation and STDDEV is the Standard Deviation (assuming a degree of freedom of 1). The formula above represents Scott's rule for determining the binsize of an histogram, multiplied by 2 and with STDDEV replaced by MAD. The use of the MAD instead of the standard deviation makes the algorithm more robust to the presence of outliers. The factor  $1/0.67449$  is needed to convert MAD into standard deviation (this is strictly only valid for a Gaussian distribution).
  - If `bin_size` is larger than 0, then `binsize = bin_size`.
3. Builds a histogram, considering only points in  $X$  within `max` and `min`. Let's define  $v_i$  as the position of the lower limits of the  $i$ -th bin (a.k.a. "class") and  $h_i$  as the number of elements in the  $i$ -th bin (a.k.a. "frequency"). The vectors  $v_i$  and  $h_i$  are outputs,  $i$  running from 0 to  $N - 1$  (do not confuse  $N$  with  $N_{\text{TOT}}$ ).
4. Compute the mode  $m$  and its error according to the specified algorithm - see section [B.4.1](#), section [B.4.2](#), and section [B.4.3](#)



**B.4.1 Mode algorithm: HDRL\_MODE\_FIT**

1. Locate the highest value of  $h$ . Define the lower limit of the modal class, the value of  $v$  that corresponds to  $h$ , i.e.  $v_M$  with  $M$  the index of the highest bin. If there are  $N$  adjacent bins with the same value of  $h$  proceed as follows:
  - (a) Take the mean of the indices of the  $n$  bins ( $i_1, i_2, \dots, i_n$ ) and round it to the closest integer value  $i_m$  and define  $M = i_m$ .
  - (b) If there are  $n$  non-adjacent bins with the same value of  $h$ , identify the subgroup of  $m$  adjacent bins with lower  $v$  and proceed as before.
2. Collect “at most” 2 bins below  $v_M$ , and 2 bins above it. Define  $x = [v_{M-2}, v_{M-1}, v_M, v_{M+1}, v_{M+2}]$  and  $y = [h_{M-2}, h_{M-1}, h_M, h_{M+1}, h_{M+2}]$ . If  $M \leq 0$  or  $M \geq N - 3$  be conservative, e.g. if  $M = 0$  then  $x = [v_0, v_1, v_2]$ .
3. If there are at least three points in  $x, y$ , then fit a parabola, otherwise the routine will return with an appropriate error code. Define the mode as the value that corresponds to the maximum of the parabola. In formulas, if  $p(x) = a_0 + a_1x + a_2x^2$  is the equation of the parabola ( $x$  is continuous variable) and  $p(x_M)$  is the maximum of the parabola, then the mode is  $m = x_M + \text{binsize}/2$ .

The error on the coefficients of the fit can be used to estimate the error on the mode. The position of the maximum of the parabola is  $x_M = -a_1/(2 \cdot a_2)$ . Its error can be obtained via error propagation:

$$(\Delta x_M)^2 = \left(\frac{\Delta a_1}{2a_2}\right)^2 + \left(\frac{a_1 \Delta a_2}{2a_2^2}\right)^2 + \text{COVAR\_TERM}$$

COVAR\_TERM is an additional term due to the fact that variables  $a_2$  and  $a_1$  are not independent:

$$\text{COVAR\_TERM} = 2 \cdot \frac{-1}{2 \cdot a_2} \frac{a_1}{2a_2^2} \cdot \text{Covar}(a_2, a_1)$$

$\text{Covar}(a_2, a_1)$  is the term of the covariance matrix. The covariance matrix is defined as the inverse of the following matrix (parabolic case):

$$\begin{bmatrix} \sum w_i & \sum w_i x_i & \sum w_i x_i^2 \\ \sum w_i x_i & \sum w_i x_i^2 & \sum w_i x_i^3 \\ \sum w_i x_i^2 & \sum w_i x_i^3 & \sum w_i x_i^4 \end{bmatrix}$$

Because we do not consider the errors on the individual points, we can set  $w_i = 1$ . HDRL actually uses GSL to compute the fit coefficients  $a_i$ , and the associated errors (taking into account of the covariance).

4. If the fit does not converge, or the mode  $m$  computed above is more than  $\text{binsize}/2$  outside the maximum bin, HDRL exit with corresponding error message. In formulas, if

$$|v_M + \text{binsize}/2 - m| > \text{binsize}$$

HDRL issues an appropriate error message.

Figure B.4.1 illustrates the method.

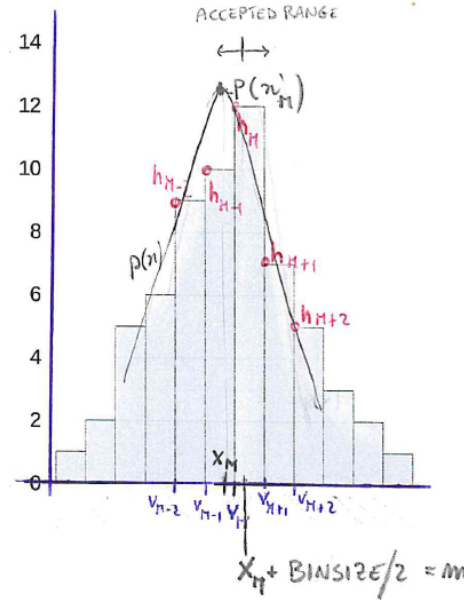


Figure B.4.1: Graphic representation of the “fit” method to compute the mode of an histogram.

**B.4.2 Mode algorithm: HDRL\_MODE\_WEIGHTED**

1. Locate the highest value of  $h$ . Define the lower limit of the modal class, the value of  $v$  that corresponds to  $h$ , i.e.  $v_M$  with  $M$  the index of the highest bin. If there are  $N$  adjacent bins with the same value of  $h$  proceed as follows:
  - (a) Take the mean of the indices of the  $n$  bins ( $i_1, i_2, \dots, i_n$ ) and round it to the closest integer value  $i_m$  and define  $M = i_m$ .
  - (b) If there are  $n$  non-adjacent bins with the same value of  $h$ , identify the subgroup of  $m$  adjacent bins with lower  $v$  and proceed as before.
2. If  $n$  is even, define  $v' = v_M - \text{binsize}/2$ , if  $n$  is odd define  $v' = v_M$ .
3. Collect one bin before ( $h_0$ ) and one after ( $h_2$ ) the group of  $n$  bins with highest value; in formulas  $v_{M-1-\text{int}(n/2)} \cdot v_{M+1+\text{int}(n/2)}$ , where  $\text{int}(n/2)$  is the integer part (if  $n = 3$  then  $\text{int}(n/2) = 1$ ). Define  $f_0 = h_0$  and  $f_2 = h_2$ . If  $M = 0$  then  $f_0 = 0$ . If  $M = N - 1$  then  $f_2 = 0$ .
4. Define  $d1 = f1 - f0$ ,  $d2 = f1 - f2$  and the weight  $w = d1/(d1 + d2)$ .
5. Define the mode  $m = v_M + \text{binsize} \cdot w$ . If  $w = 0$  or infinite, then set  $w = 0.5$ .

Error on the mode is computed via error propagation, assuming that the errors on  $d1$  and  $d2$  are given by

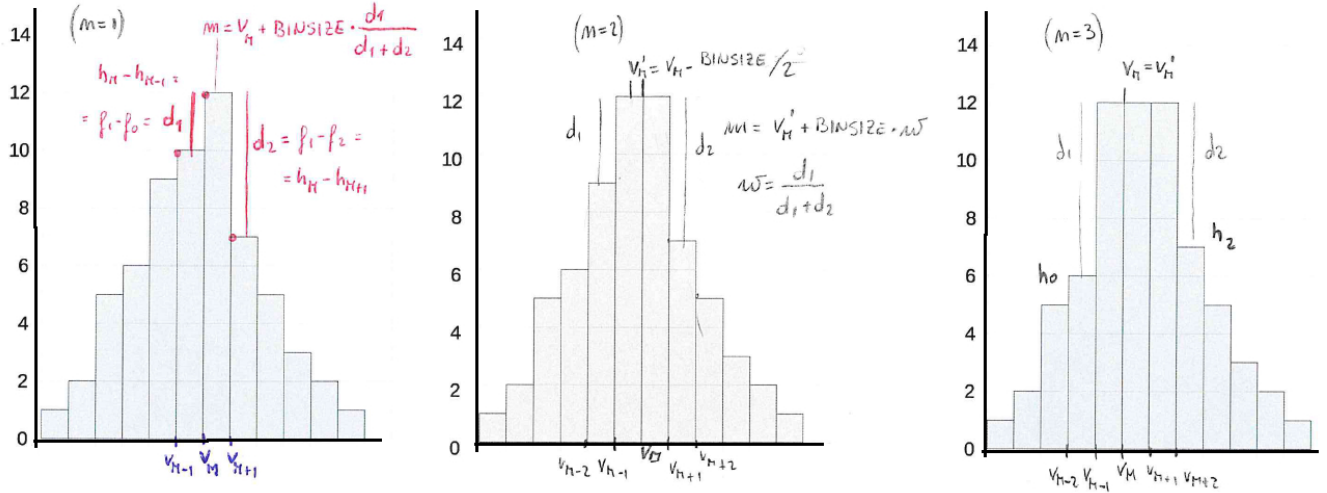


Figure B.4.2: Graphic representation of the “weight” method to compute the mode of an histogram. Three cases are show: single modal class (n=1, left panel), two modal classes (n=2, central panel), and three modal classes (n=3, right panel).

Poisson statistics ( $\Delta f_i = \sqrt{f_i}$ )

$$\begin{aligned}
 \Delta d_1 &= \sqrt{f_1 + f_0} \\
 \Delta d_2 &= \sqrt{f_1 + f_2} \\
 \Delta m &= \text{binsize} \cdot \Delta w \\
 \Delta w^2 &= \left( \frac{d_2 \Delta d_1}{(d_1 + d_2)^2} \right)^2 + \left( \frac{d_1 \Delta d_2}{(d_1 + d_2)^2} \right)^2
 \end{aligned} \tag{26}$$

Figure B.4.2 illustrates the method with  $n=1,2$ , and 3.

### B.4.3 Mode algorithm: HDRL\_MODE\_MEDIAN

This algorithm is suited for strongly asymmetric distributions (as Gamma distributions).

1. Locate the highest value of  $h$ . Define the lower limit of the modal class, the value of  $v$  that corresponds to  $h$ , i.e.  $v_M$  with  $M$  the index of the highest bin. If there are  $N$  adjacent bins with the same value of  $h$  proceed as follows:
  - (a) Take the mean of the indices of the  $n$  bins ( $i_1, i_2, \dots, i_n$ ) and round it to the closest integer value  $i_m$  and define  $M = i_m$ .
  - (b) If there are  $n$  non-adjacent bins with the same value of  $h$ , identify the subgroup of  $m$  adjacent bins with lower  $v$  and proceed as before.



2. Define the mode as the median of the values of  $v_M < X < v_M + binsize$ . The error on the median is the standard deviation of the  $X$  values.



## C Bad Pixel Determination

### C.1 Detailed Description of the various Algorithms

#### C.1.1 Bad-pixel detection on a single image

The `hdr1_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 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.1.2 shows the results on simulated flat fields.

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

The `hdr1_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 ≥ 10`

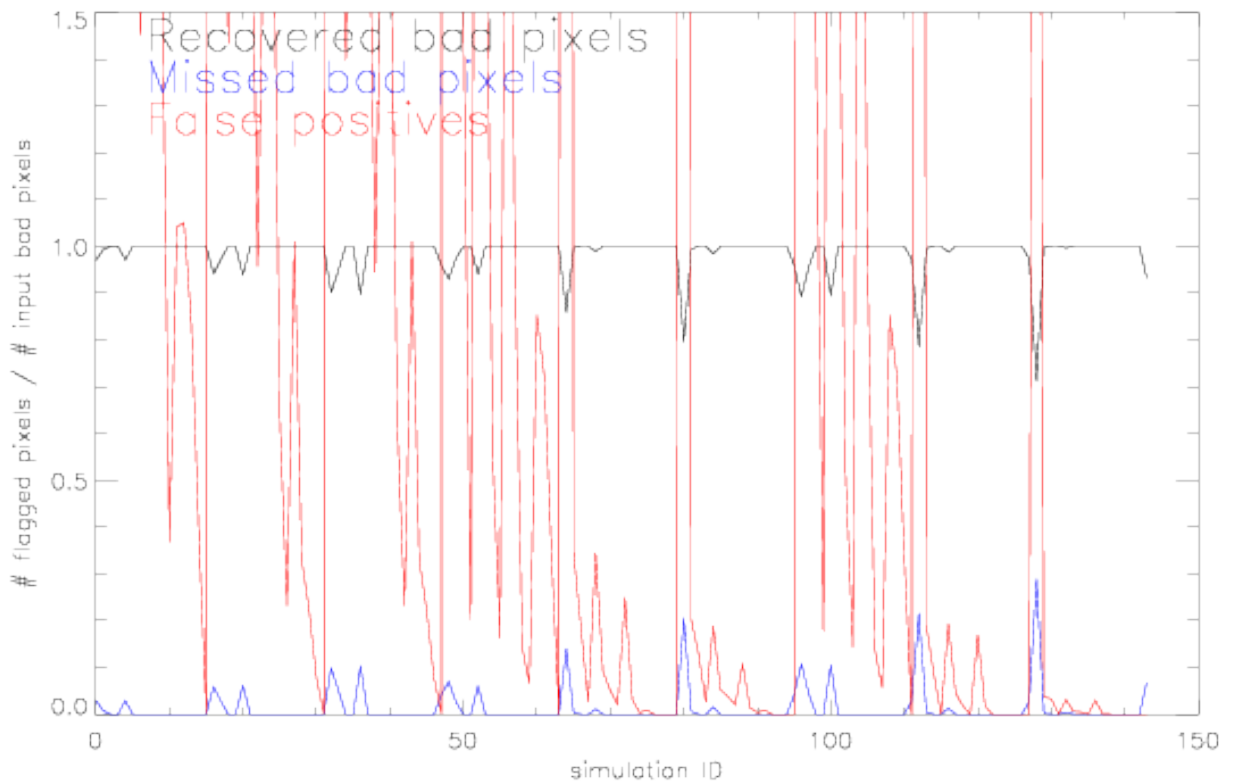


Figure C.1.1: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields using method 1 of the `hdr1_bpm_2d` recipe. The best performances are obtained setting `-bpm.smooth_x/y =49`, `-bpm.kappa_low/high > 5`, and `-pfx/y =3`. A large fraction 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).

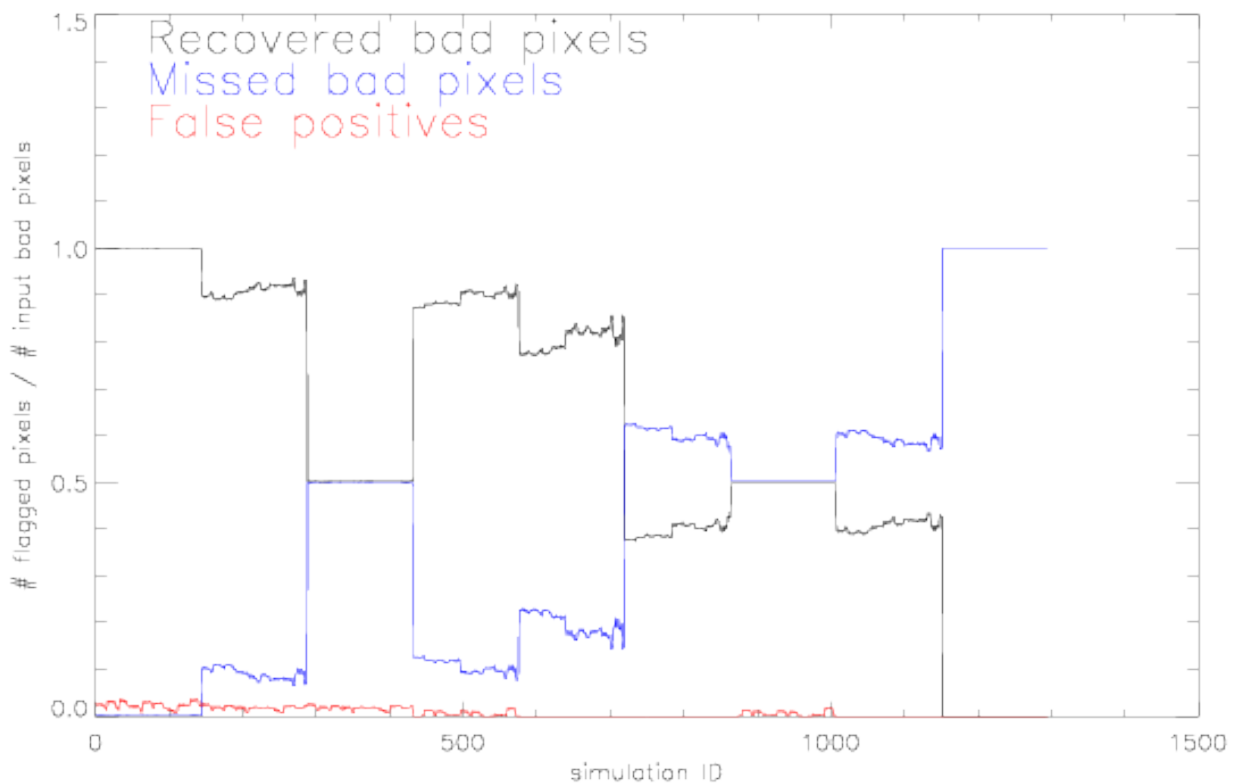


Figure C.1.2: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields using method 2 of the `hdr1_bpm_2d` recipe. The best performances are obtained setting `-bpm.kappa_low/high = 3` (run ID < 100). For larger values, the number of detected bad pixels decreases.

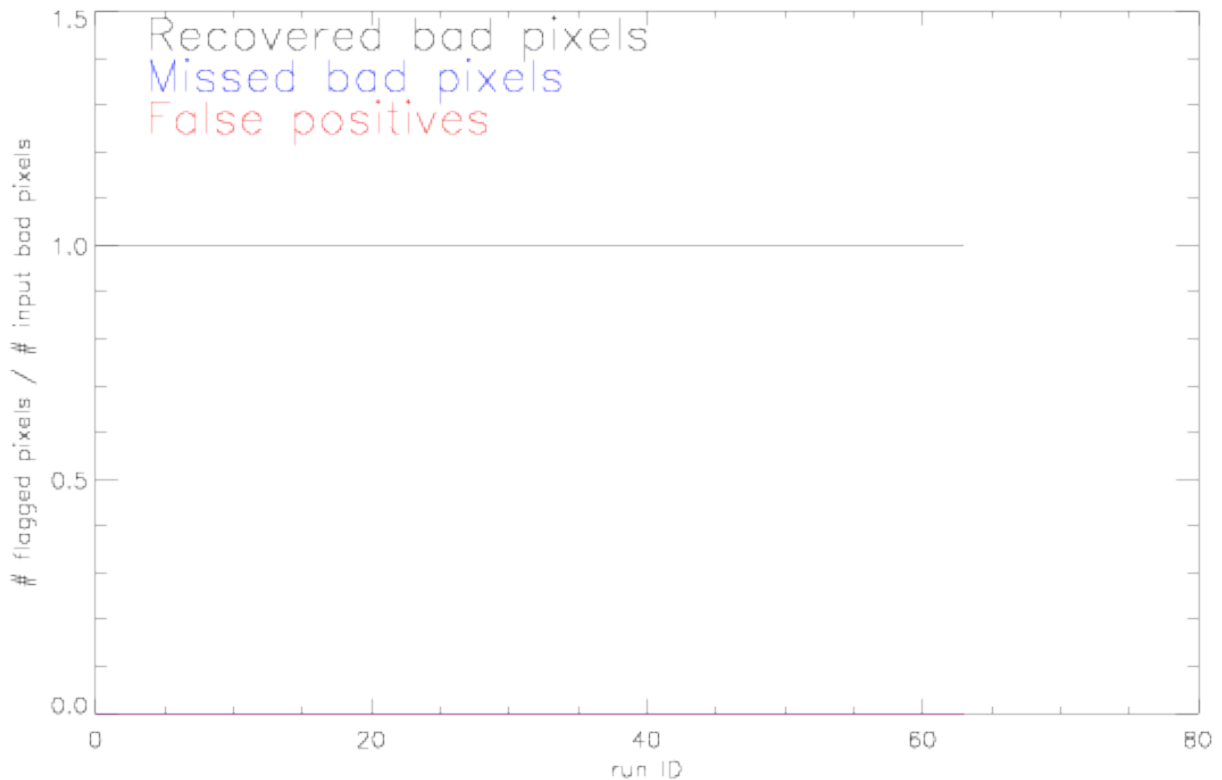


Figure C.1.3: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 2 of the `hdr1_bpm_3d` recipe.

(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.1.3 and C.1.4 show the tests on methods 2 and 3 on noiseless simulated stack of flat fields.

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

The `hdr1_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.

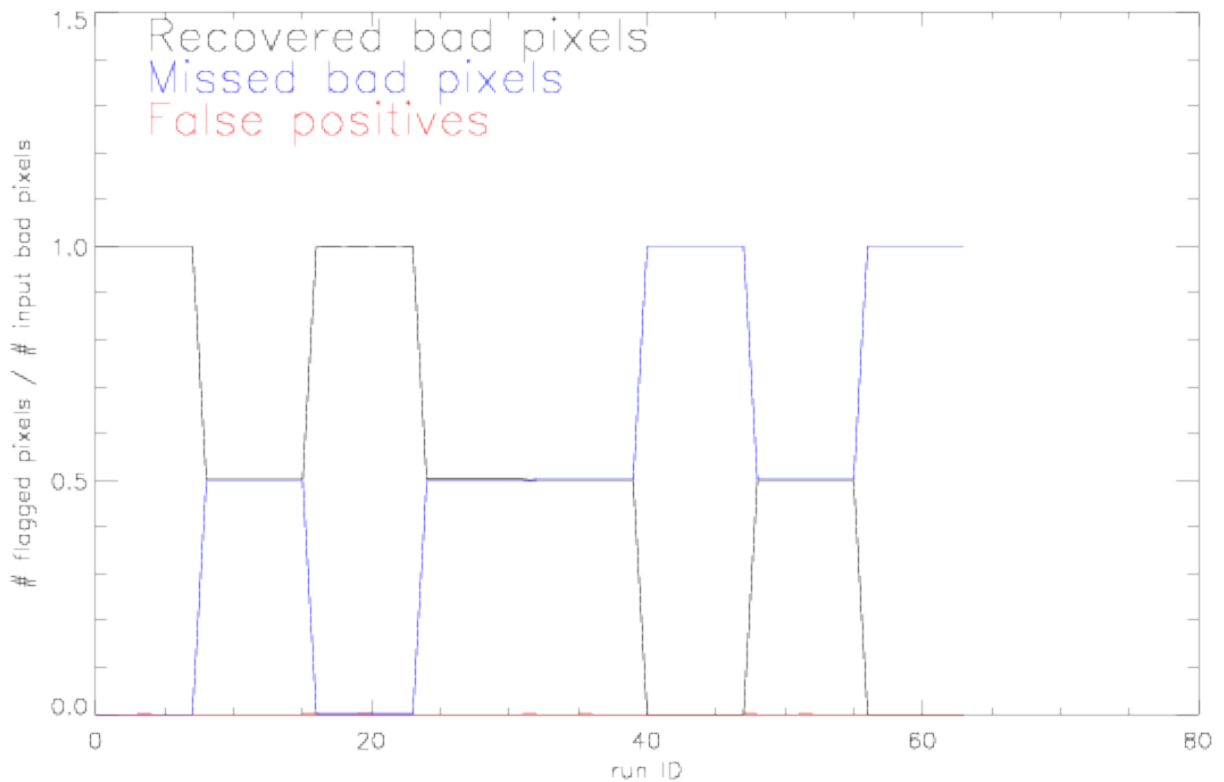


Figure C.1.4: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 3 of the `hdr1_bpm_3d` recipe. 50% of input bad pixels are missed if `-bpm.kappa_low/high > 10`.



It is therefore advisable to remove cosmic rays on individual frames, before running `hdr1_bpm_fit` on a sequence of images.

The `hdr1_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.1.5, C.1.6, and C.1.7 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 `hdr1_bpm_fit`. Figure C.1.8 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 default 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 ([10]). 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 [10].

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

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

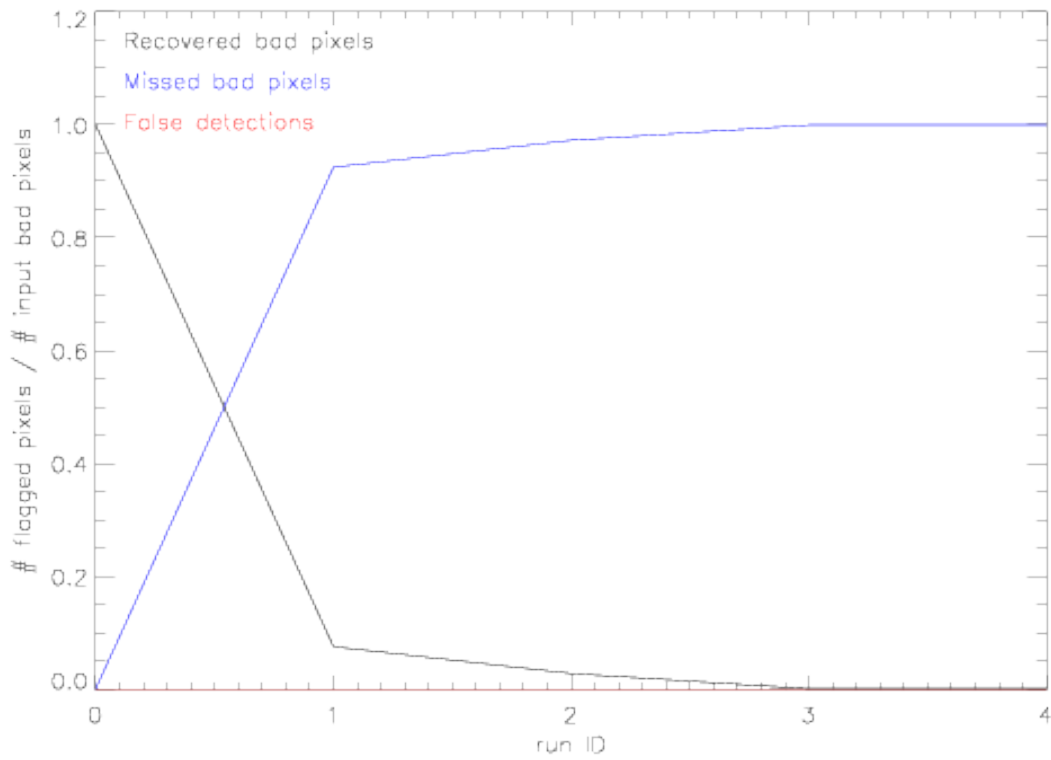


Figure C.1.5: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 2 (`-rel-rchi2`) of the `hdr1_bpm_fit` recipe. The best performance (run ID=0) corresponds to `-rel-rchi2 = 1`.

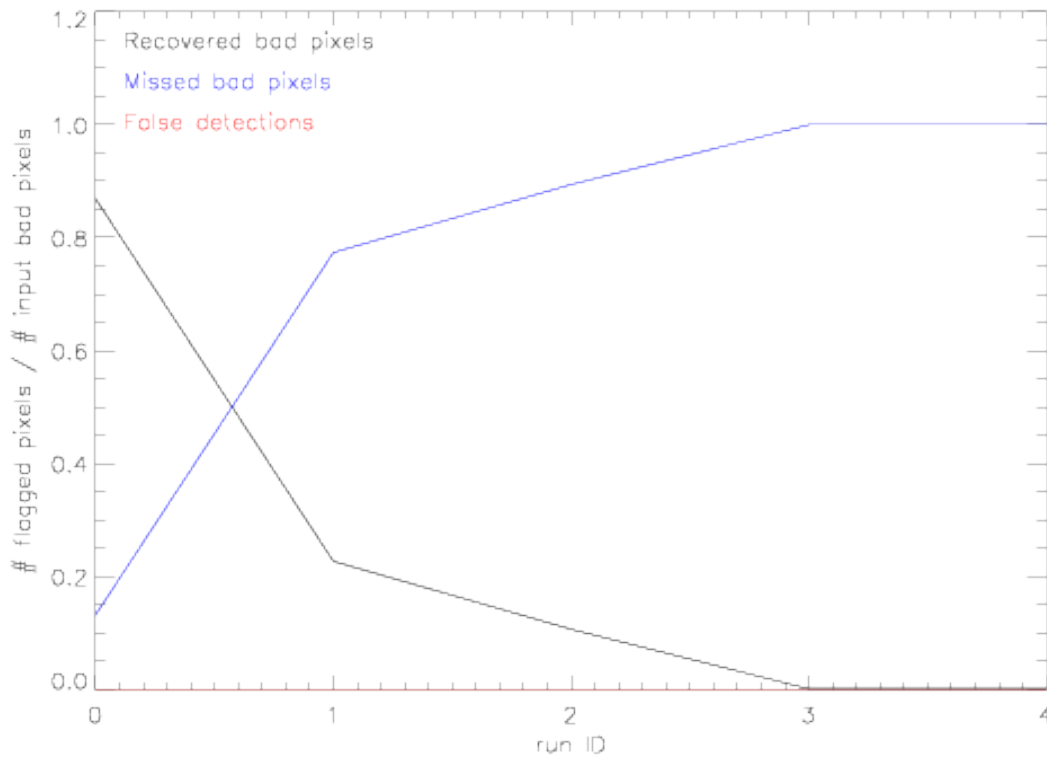


Figure C.1.6: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 3 (`-rel-coef`) of the `hdr1_bpm_fit` recipe. The best performance (run ID=0) corresponds to `-rel-coef = 1`.

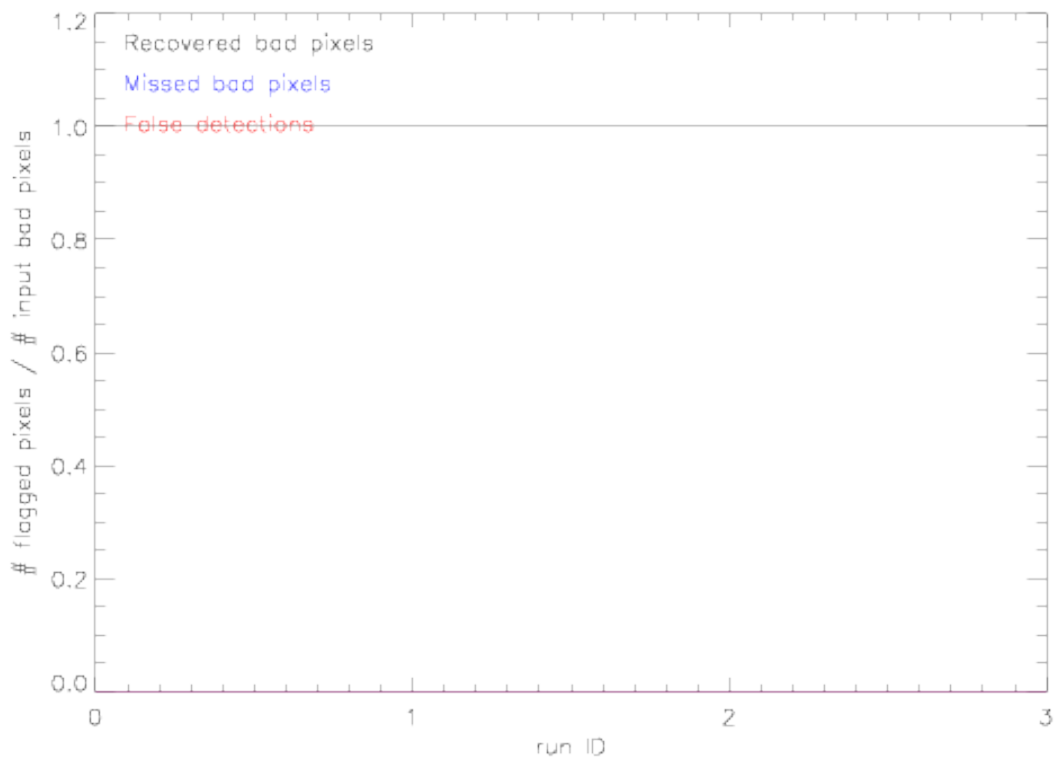


Figure C.1.7: Fraction of recovered bad pixels (black), missed bad pixels (blue) and false detections (red) on simulated flat fields, using method 4 (`-pval`) of the `hdr1_bpm_fit` recipe.

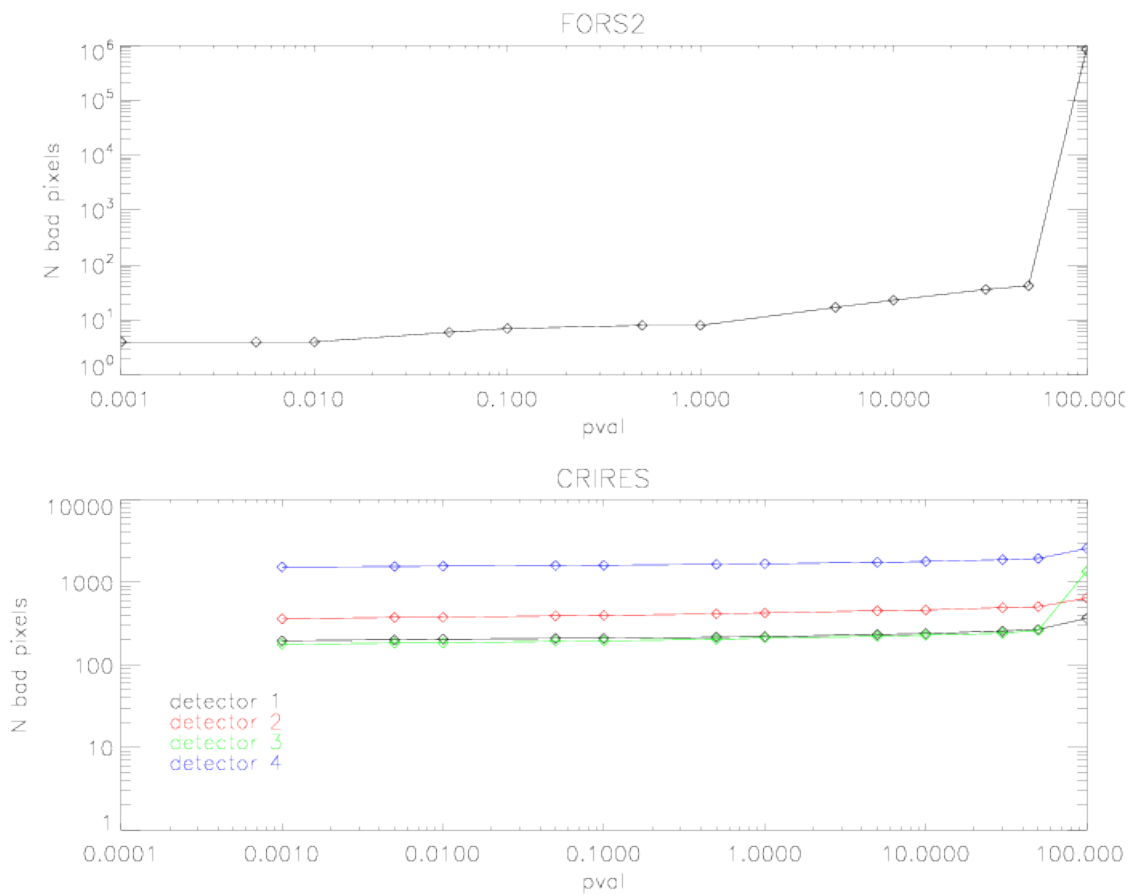


Figure C.1.8: Number of flagged bad pixels with `hdr1_bpm_fit`, method 4 as a function of the `-pval` on a series of internal flats for FORS2 (upper panel) and CRILES (lower panel). The number of detected bad pixels does not vary significantly within  $0.001 < -pval < 0.1$ .



-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  $89.4\% \pm 0.4\%$  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  $6.8\% \pm 0.3\%$  over the full range of cosmic ray densities.

As is evident in figure [C.1.9](#), 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 detection efficiency is not notably worse and is, on average, at a level of  $89.1\% \pm 0.4\%$ . The difference between

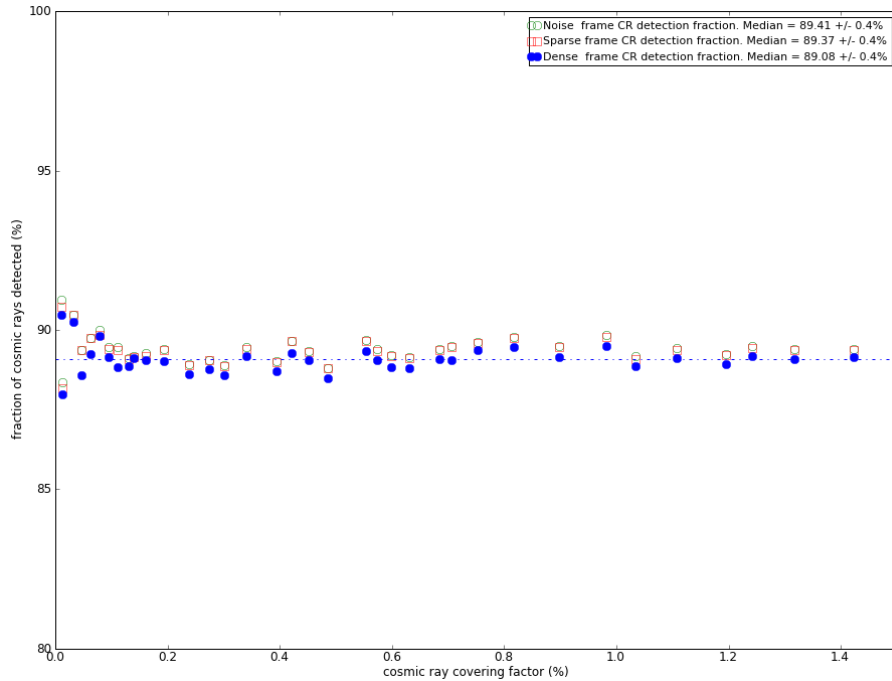


Figure C.1.9: The detection efficiency of *hdrldemo bpm lacosmic* versus the cosmic ray covering factor (in percent). The efficiency (given in percent) is measured from the number of cosmic rays detected by *hdrldemo bpm lacosmic* when applied to the pure noise frame (open green circles), the sparsely 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).

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.1.9 is given by the fraction of input cosmic rays not detected (see figure C.1.10). The total count of undetected cosmic rays (figure C.1.10, 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.1.11). 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.

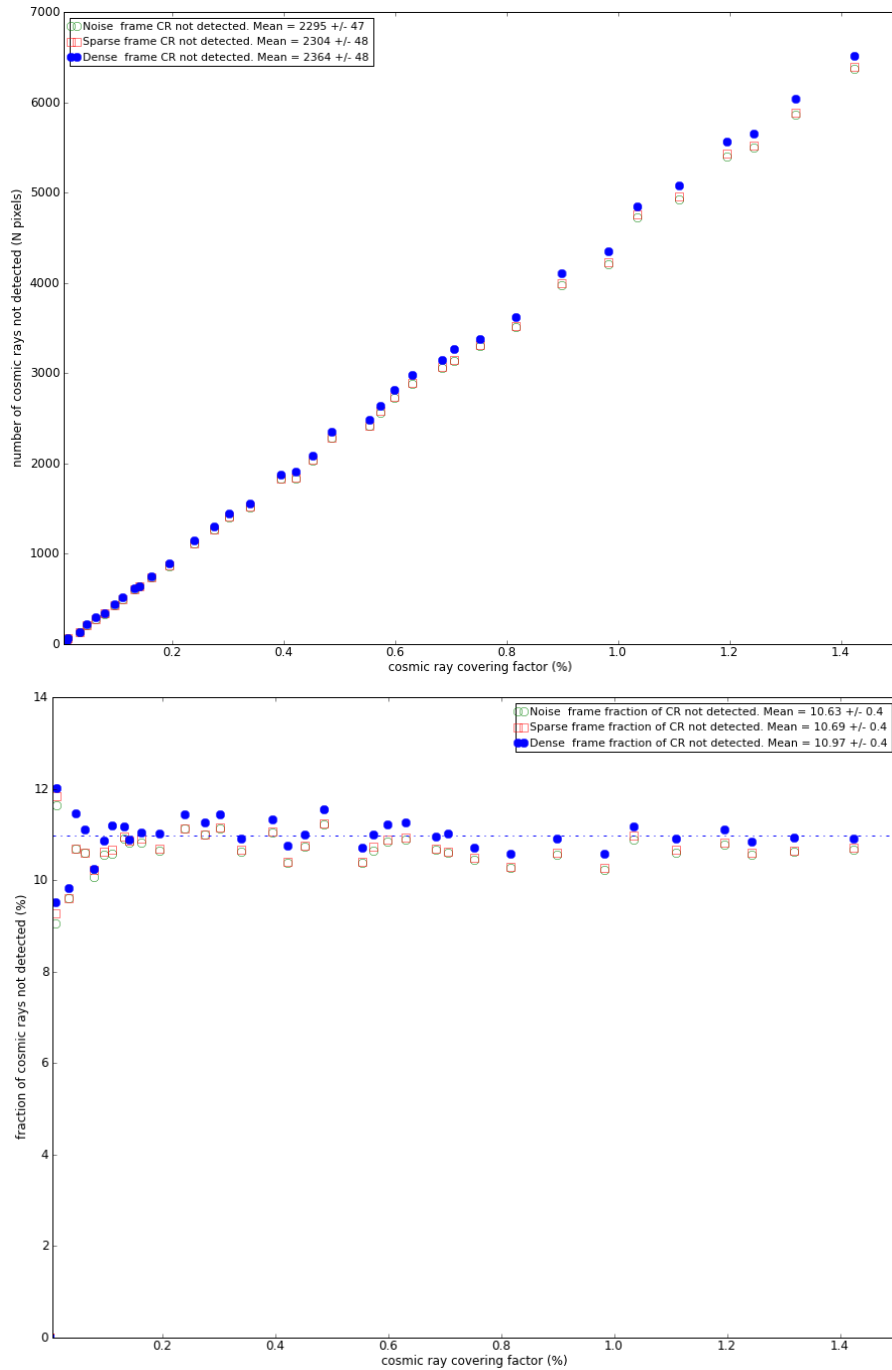


Figure C.1.10: The number of cosmic rays **not** detected (top panel) and the non-detection fraction (bottom panel) versus the cosmic ray covering factor (given in percent).

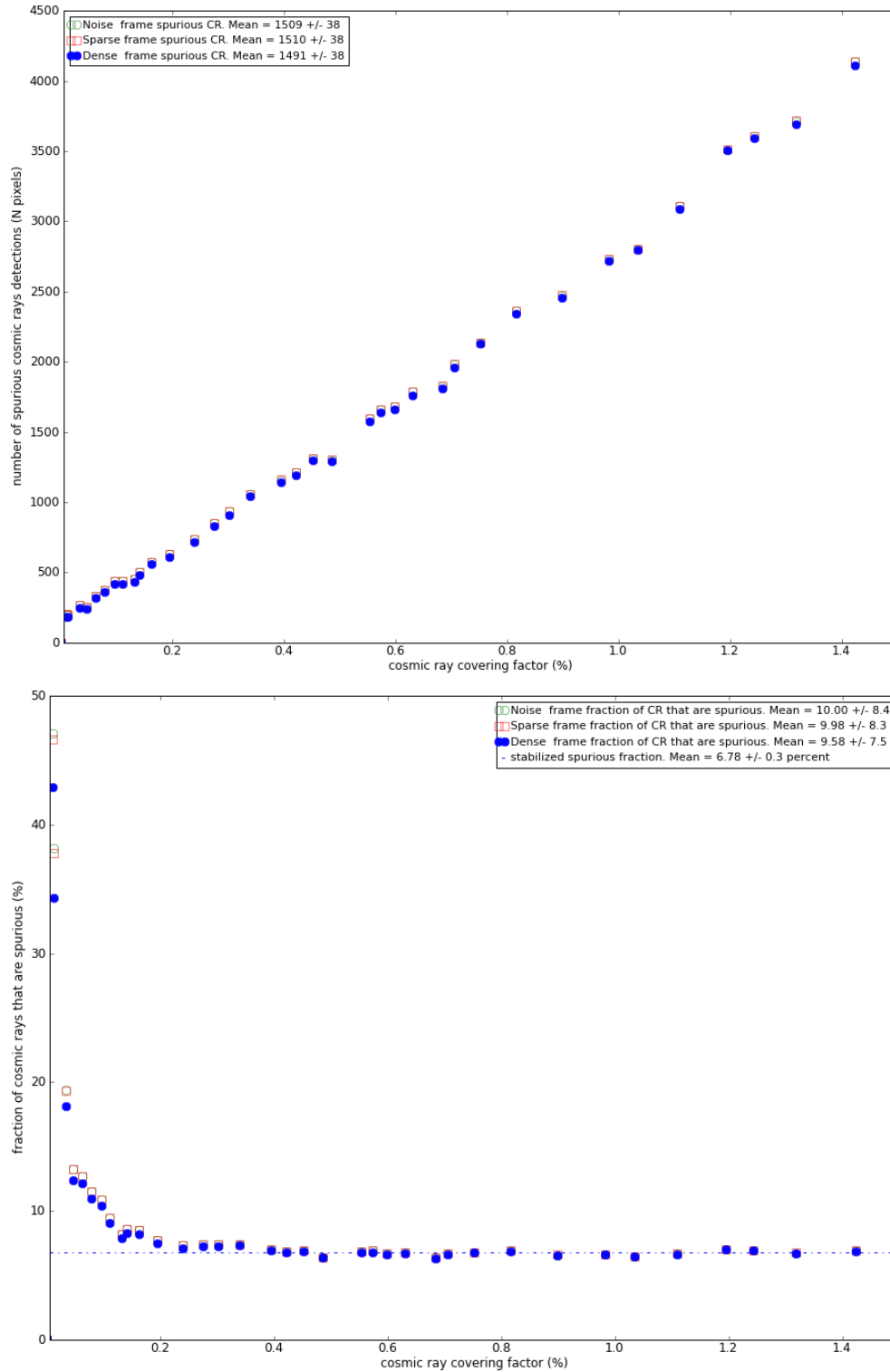


Figure C.1.11: The number of spurious cosmic ray detections (top panel) and the spurious fraction (bottom panel) versus the cosmic ray covering factor (given in percent).



## **D Masterfringe Computation and Removal**

### **D.1 Detailed Description of the de-fringing Algorithm**

TO BE WRITTEN (SDPG)



## **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)



## F Efficiency Calculation

### F.1 Introduction

In order to determine the efficiency of an instrument (including telescope and detector) spectrophotometric standard stars are observed on photometric nights and with setups that collect as much as possible (e.g. wide slits for slit spectrographs). After correcting for atmospheric effects, photon energy, and mirror collecting area the ratio between the observed spectrum and the reference spectrum of the standard star will provide the efficiency of the system.

The observed spectra of the standard stars are first corrected for atmospheric effects:

1. The continuous *atmospheric extinction* is usually corrected with a standard extinction curve obtained for a given observatory, scaled with the airmass of the observation. During non-photometric conditions the actual extinction may be higher than the one tabulated in the extinction curve. The impact of this correction decreases with increasing wavelength.
2. Wavelength above about 600 nm<sup>23</sup> are affected by *telluric absorption lines*. If these are not corrected the efficiency will contain a mixture of instrument response and atmospheric response, especially if the reference spectra are free from telluric lines (space-based data or model spectra). Reference spectra for many standard stars used for ESO instruments (e.g. UVES, FORS2, VIMOS) contain telluric lines (e.g. [5, 4]). For such reference spectra a telluric correction of the observed spectra is useless and one should instead interpolate the efficiency across the regions of significant telluric absorption.

Possibly the wavelength scale between the observed standard star spectrum and the reference one needs to be aligned, either to correct differences in radial velocity between the observed and the reference spectrum in case of model spectra (e.g. X-shooter, SINFONI) and/or to correct shifts introduced by imprecise positioning of the flux standard star in the slit (e.g. VIMOS). If the differences in wavelength become significant for a given resolution the division of the reference spectrum by the observed one will create pseudo-P Cyg profiles, which may distort the efficiency.

These requirements result in the algorithmic steps described in Sect. 4.9.

### F.2 Testing

Once the response determination has been implemented in HDRL and incorporated in the SINFONI pipeline QCG tested it by comparing the resulting values to the SPIFFI commissioning report (VLT-TRE-MPE-14720-8003) and found good agreement. The efficiency is trended at [http://www.eso.org/observing/dfo/quality/SINFONI/reports/HEALTH/trend\\_report\\_EFFICIENCY\\_HC.html](http://www.eso.org/observing/dfo/quality/SINFONI/reports/HEALTH/trend_report_EFFICIENCY_HC.html)

---

<sup>23</sup>Ozone absorption affects the bluest part of ground-based data at about 310–330 nm, but these are noticeable only in high S/N data.



## G Response Calculation

### G.1 Introduction

In order to determine the spectroscopic response of an instrument (including telescope and detector) spectrophotometric standard stars are observed with the same setup as the science targets<sup>24</sup>. Spectrophotometric standard stars are preferably observed during photometric conditions to reduce the importance of atmospheric effects. After correcting for atmospheric effects the ratio between the reference spectrum of the standard star and the observed spectrum will provide the response of the system.

If the science and/or standard star data are observed during non-photometric conditions an absolute flux calibration is impossible. For slit spectra the use of a narrow slit introduces slit losses that are hard to correct. In many cases, however, only a *relative* flux calibration is required, namely a response curve that allows the user to correct the observed science data for bumps and wiggles that were not corrected by the flat fielding process.

The observed spectra of the standard stars are first corrected for atmospheric effects:

1. The continuous *atmospheric extinction* is usually corrected with a standard extinction curve obtained for a given observatory, scaled with the airmass of the observation. During non-photometric conditions the actual extinction may be higher than the one tabulated in the extinction curve. The impact of this correction decreases with increasing wavelength.
2. Wavelength above about 600 nm<sup>25</sup> are affected by *telluric absorption lines*. If these are not corrected the response curve will contain a mixture of instrument response and atmospheric response, especially if the reference spectra are free from telluric lines (space-based data or model spectra). Reference spectra for many standard stars used for ESO instruments (e.g. UVES, FORS2, VIMOS) contain telluric lines (e.g. [5, 4]). For such reference spectra a telluric correction of the observed spectra is useless and one should instead interpolate the response across the regions of significant telluric absorption.

Possibly the wavelength scale between the observed standard star spectrum and the reference one needs to be aligned, either to correct differences in radial velocity between the observed and the reference spectrum in case of model spectra (e.g. X-shooter, SINFONI) and/or to correct shifts introduced by imprecise positioning of the flux standard star in the slit (e.g. VIMOS). If the differences in wavelength become significant for a given resolution the division of the reference spectrum by the observed one will create pseudo-P Cyg profiles, which will distort the response.

These requirements result in the algorithmic steps described in Sect. 4.10.

### G.2 Testing

Once the response determination has been implemented in HDRL and incorporated in the SINFONI pipeline we tested it by calibrating one standard star with the response curve from another (observed with the same setup, but not necessarily during the same night). From the resulting cube we extracted the one-dimensional spectrum with

<sup>24</sup>For slit spectrographs one generally uses a wide slit for flux standard stars to capture all flux.

<sup>25</sup>Ozone absorption affects the bluest part of ground-based data at about 310–330 nm, but these are noticeable only in high S/N data.

QFitsView (using a radius of 11 pixels and sum) and corrected it with a background spectrum extracted with the same aperture. The difference of the two spectra was then compared to the reference flux for the standard star. Figures G.2.1, G.2.2, and G.2.3 show example results. Due to the small number of observations only limited tests were possible, which showed no problems that cannot be explained by transparency differences between the observations of the standard stars.

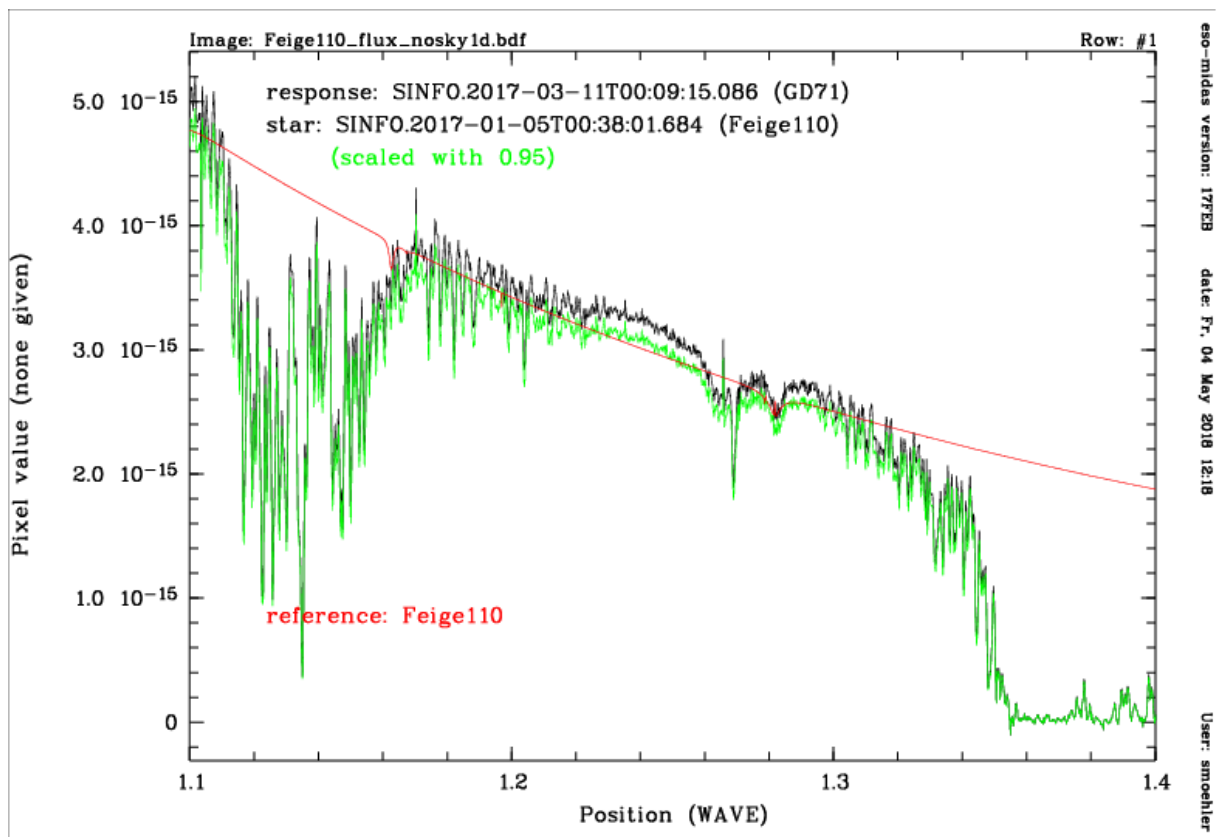


Figure G.2.1: The  $J$  spectrum of the flux standard stars Feige 110 calibrated with a response curve from the flux standard star GD71. The two stars were observed about 2 months apart on nights of unknown quality. Scaling the flux-calibrated spectrum of Feige 110 by 95% achieves good agreement with its reference data.

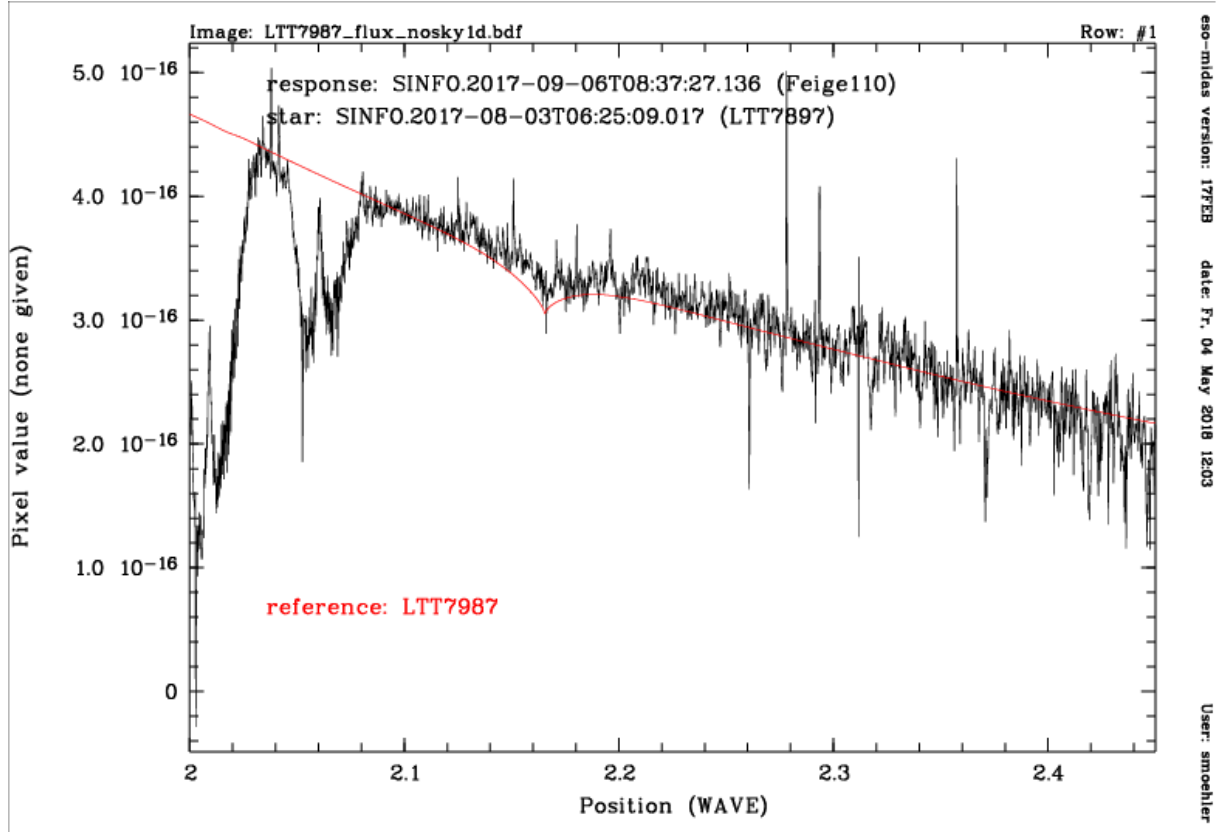


Figure G.2.2: The  $K$  spectrum of the flux standard stars LTT7987 calibrated with a response curve from the flux standard star LTT7987. The two stars were observed about 1 month apart on nights of unknown quality. Nevertheless the flux-calibrated spectrum of LTT7987 agrees well with its reference data.

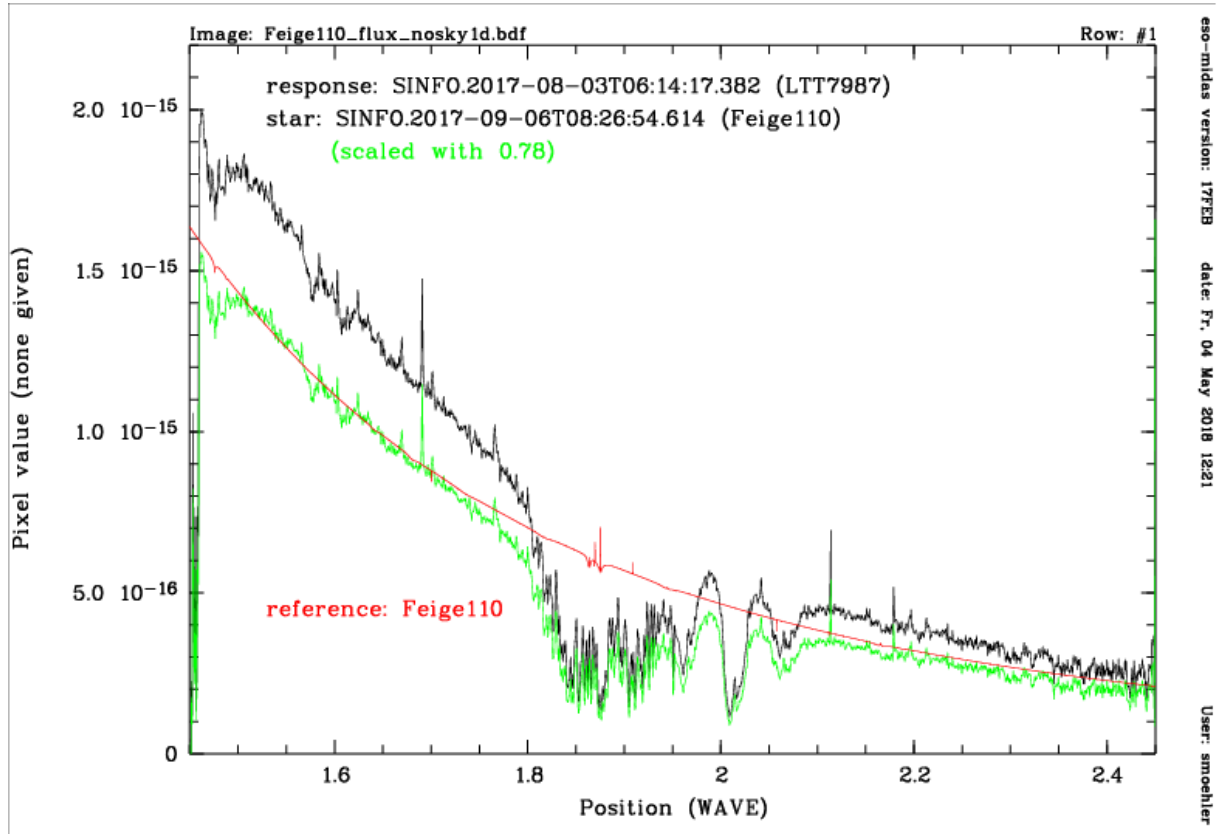


Figure G.2.3: The H+K spectrum of the flux standard stars Feige 110 calibrated with a response curve from the flux standard star LTT 7987. The two stars were observed about 1 month apart on nights of unknown quality. Scaling the flux-calibrated spectrum of Feige 110 by 78% achieves good agreement with its reference data.



## 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](#) and [H.3.2](#)).

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](#)).

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.



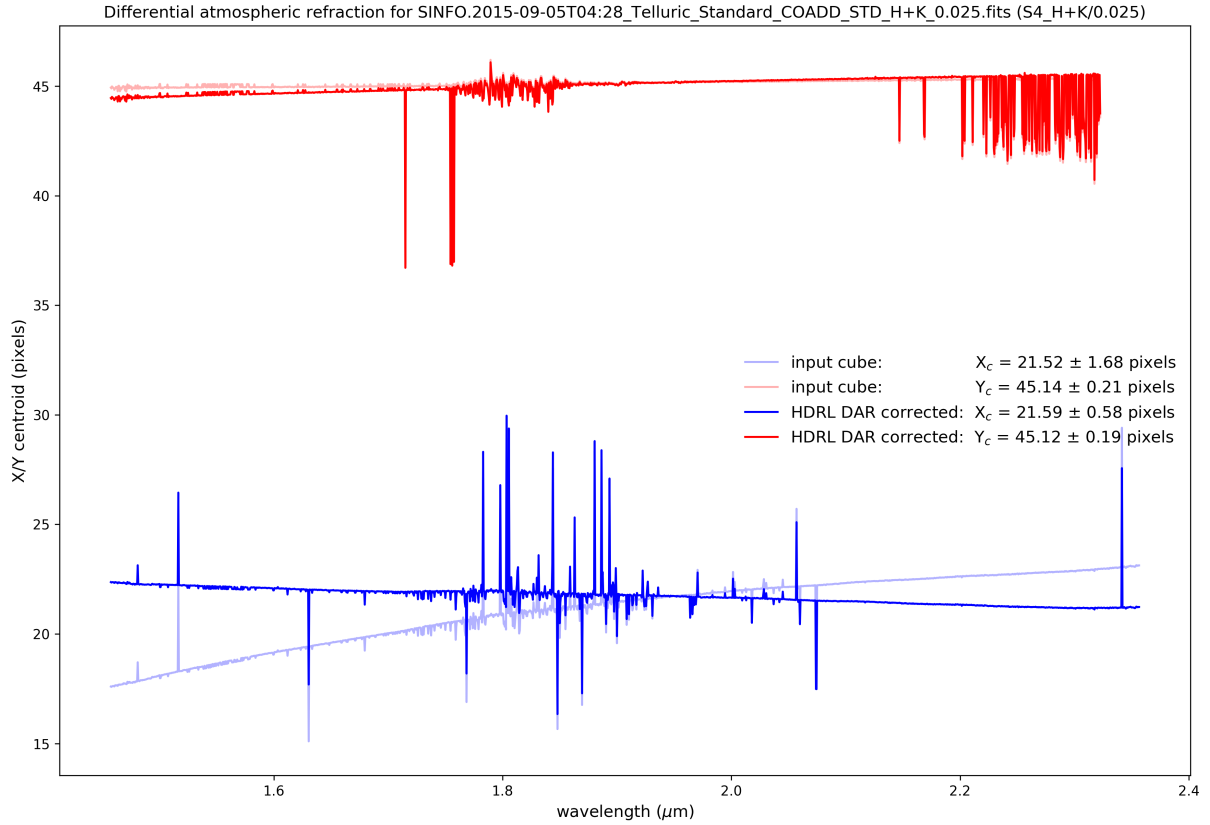
Table H.2.0: Parameter Ranges Covered by SINFONI Data

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

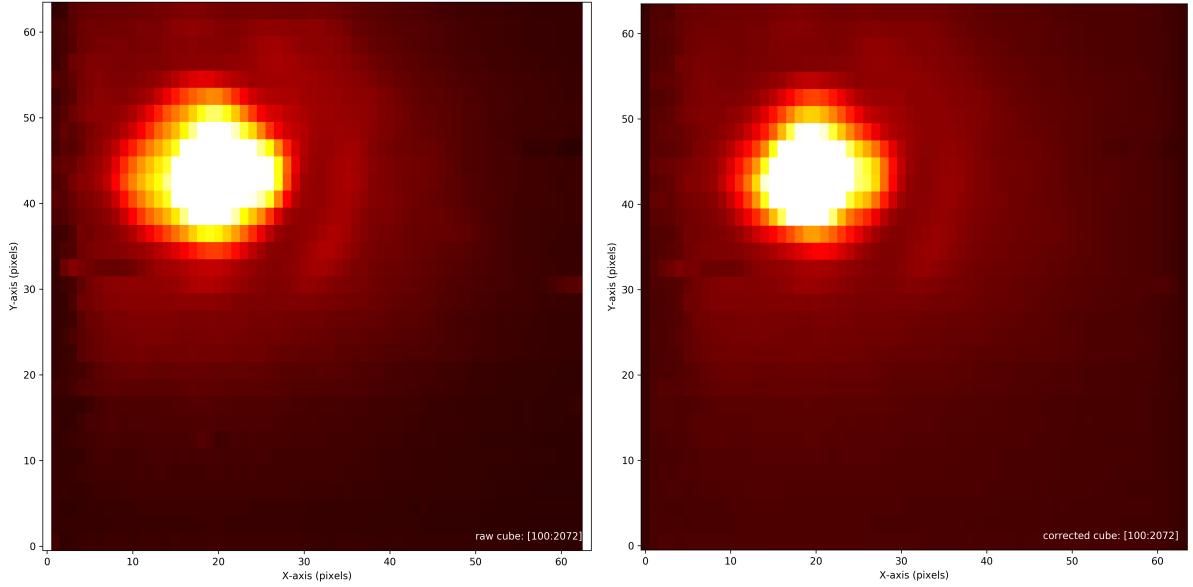
## H.3 Testing the *hdrl\_dar* Differential Atmospheric Refraction Routines with SINFONI Data

### 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. 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.3.2. In both plots, a significant qualitative improvement is apparent.



**Figure H.3.1:** A line trace of the source centroid position as a function of wavelength for the SINFONI data cube SINFONI.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 ( $Y_c$ ) centroids, 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.



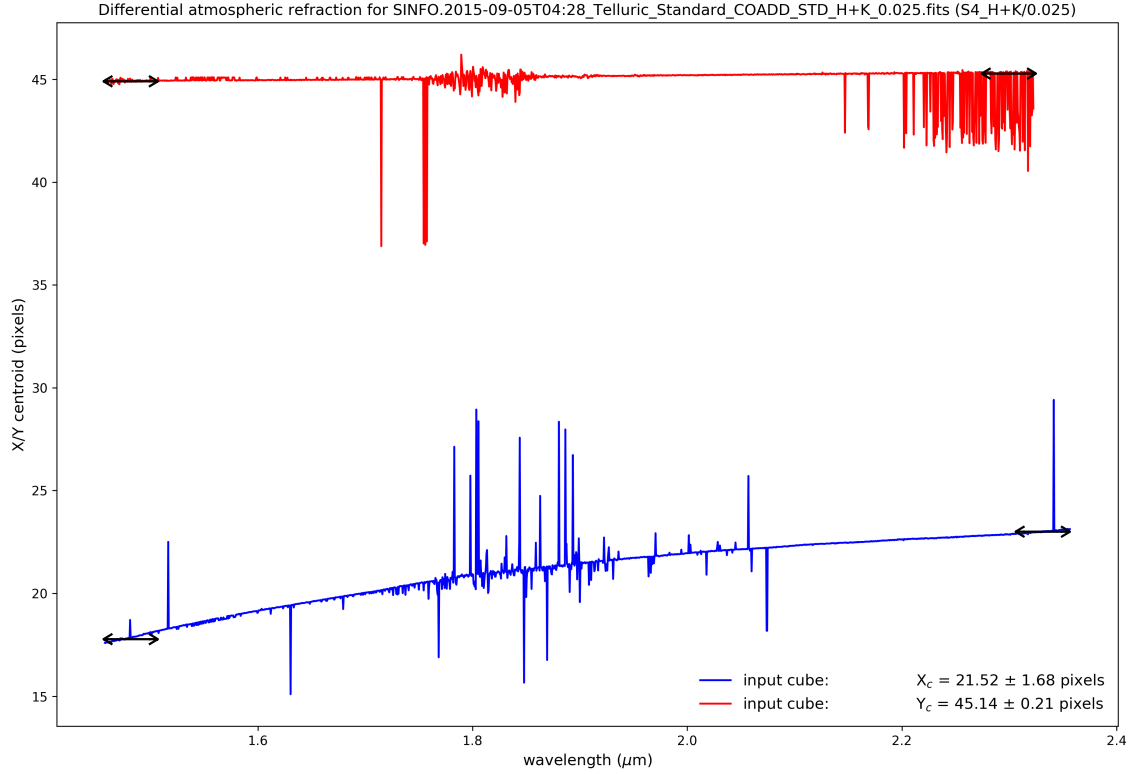
**Figure H.3.2: Left Panel:** A median collapse of the same raw data cube shown as a trace in Figure H.3.1.  
**Right Panel:** The same median collapsed cube, integer-pixel-corrected for the differential atmospheric refraction. The improvement between the raw and corrected data cubes is visually apparent. 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 *hdl\_dar*, we define a global atmospheric refraction based the source position at the end points of each SINFONI data cube (see Figure H.3.3). 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:

$$\Delta X_c = \langle X_c \rangle_{100} - \langle X_c \rangle_{N_{max}-100} \quad (27)$$

$$\Delta Y_c = \langle Y_c \rangle_{100} - \langle Y_c \rangle_{N_{max}-100} \quad (28)$$



**Figure H.3.3:** 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  $\Delta X_c$  and  $\Delta Y_c$ .

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 *hdrd\_dar*:

$$\Delta X_c(\text{data}) - \Delta X_c(\text{theory}) = 0.29 \pm 0.63 \text{ pixels} \quad (29)$$

$$\Delta Y_c(\text{data}) - \Delta Y_c(\text{theory}) = 0.02 \pm 0.19 \text{ pixels} \quad (30)$$

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  $\Delta X_c$  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 *hdrd\_dar*, can be summarised in a surface density plot displaying  $\Delta X_c(\text{data}) - \Delta X_c(\text{theory})$  vs.  $\Delta Y_c(\text{data}) - \Delta Y_c(\text{theory})$ . This plot is shown in Figure H.3.4.

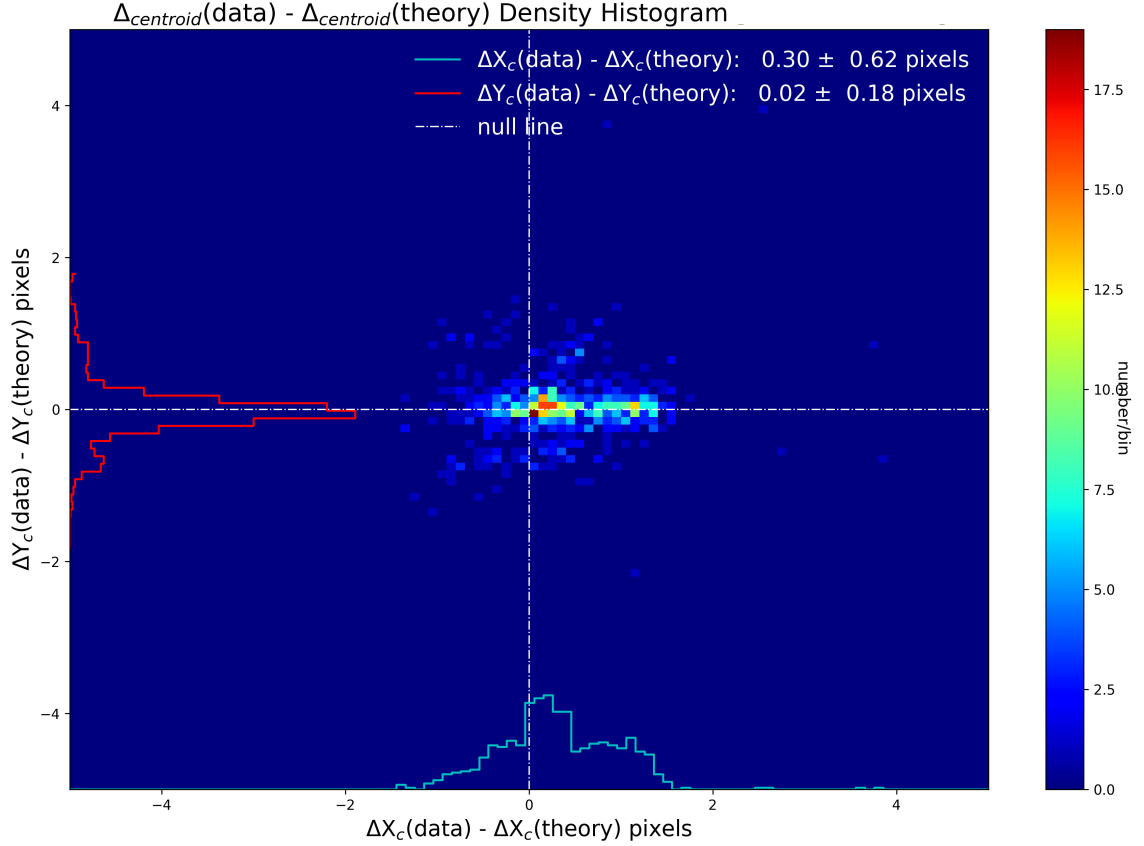


Figure H.3.4: Surface density plot of  $\Delta X_c(\text{data}) - \Delta X_c(\text{theory})$  vs.  $\Delta Y_c(\text{data}) - \Delta Y_c(\text{theory})$  for the 754 SINFONI data cubes analysed. Note that the differing widths of the  $\Delta X_c$  and  $\Delta Y_c$  histograms is due to the fact that 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 ( $\langle X_c \rangle$ ,  $\langle Y_c \rangle$ ). 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:

$$X_c \text{ residual} = | \langle X_c(\text{data}) \rangle - [X_c(\text{data}) + X_{\text{shift}}(\text{theory})] | \quad \text{pixels} \quad (31)$$

$$Y_c \text{ residual} = | \langle Y_c(\text{data}) \rangle - [Y_c(\text{data}) + Y_{\text{shift}}(\text{theory})] | \quad \text{pixels} \quad (32)$$

A surface density plot of these residuals are shown in Figure H.3.5.

$$X_c \text{ residual} = 0.15 \pm 0.13 \quad \text{pixels} \quad (33)$$

$$Y_c \text{ residual} = 0.04 \pm 0.03 \quad \text{pixels} \quad (34)$$

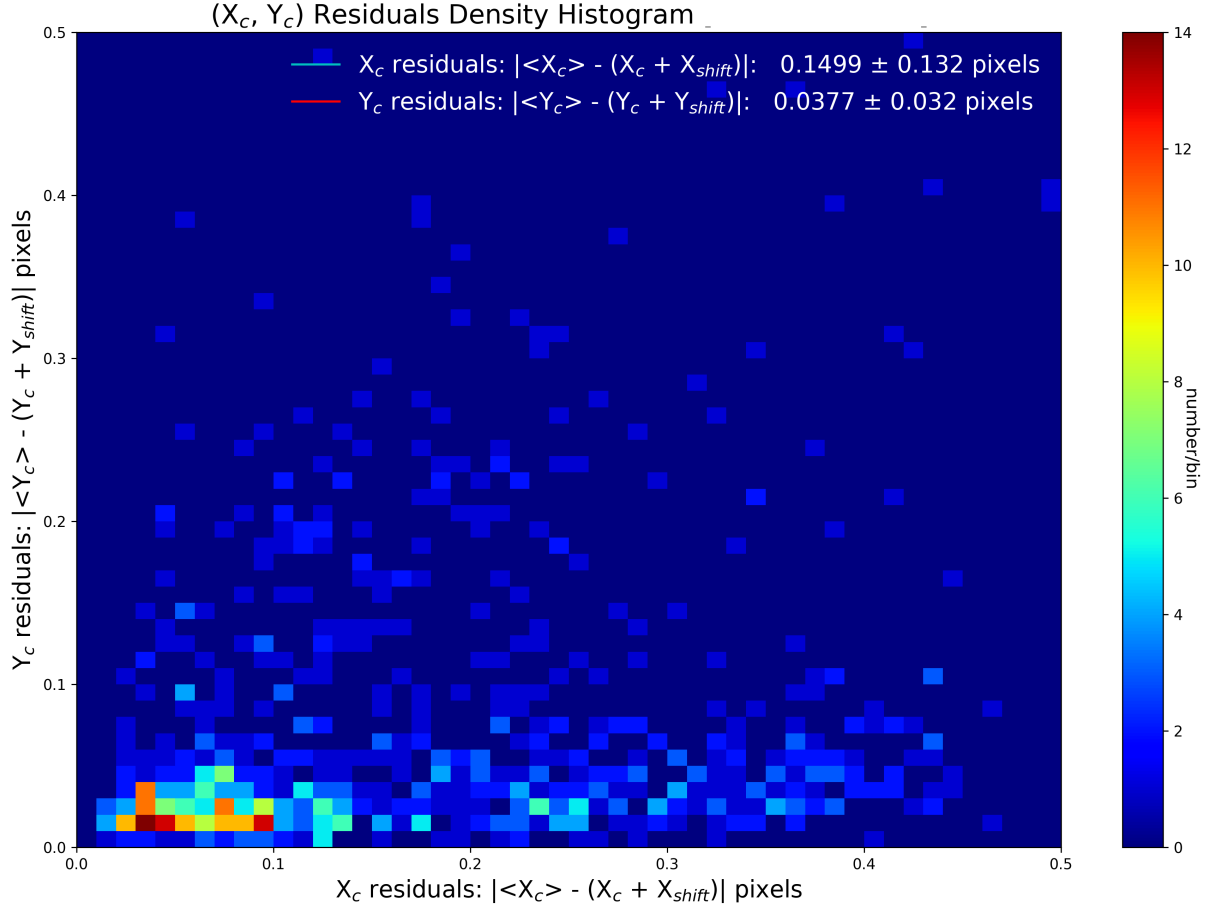


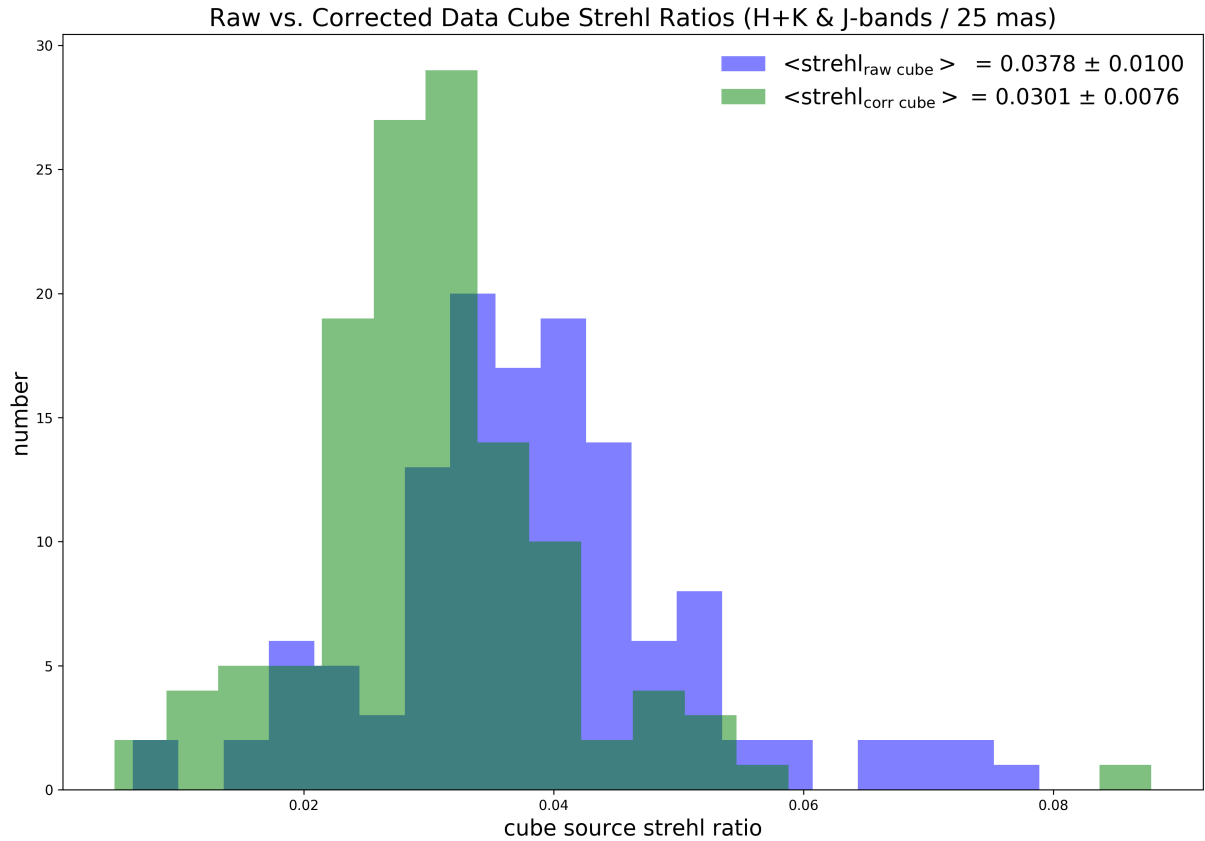
Figure H.3.5: Surface density residual plot of  $|\langle X_c(\text{data}) \rangle - [X_c(\text{data}) + X_{\text{shift}}(\text{theory})]|$  vs.  $|\langle Y_c(\text{data}) \rangle - [Y_c(\text{data}) + Y_{\text{shift}}(\text{theory})]|$  for the 753 SINFONI data cubes analysed. Note that the differing widths of the  $\Delta X_c$  and  $\Delta Y_c$  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 (HDRLEMO\_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.3.2). For each image, *SExtractor* was run using a detection and analysis threshold of  $3\sigma$ . The *hdrl\_strehl* routine was then run on the same image using a flux-radius set to  $1.5 \times \text{FWHM}$  computed by *SExtractor*. The bkg-radius-low was set to  $2.0 \times \text{FWHM}$  and the bkg-radius-high was set to  $3.0 \times \text{FWHM}$ . 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.3.6. 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.3.6:** 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.

## I Fixed Pattern Noise

The pattern noise (PN, hereafter) is an almost regular pattern that appears on the images. Most likely it is due to electronic interference. Perhaps, the simplest PN example is the odd-even column/row effect – a pattern in which consecutive columns/rows have systematically different levels. Despite being called **fixed**, the PN is transient and can vary significantly between frames of a sequence.

The PN not easily detectable by simple visual image inspection: a “good” image (Fig. I.0.1, left) is not much different from a “bad” image (Fig. I.0.1, right), because the amplitude of the pattern may easily be smaller than the random noise level. In the special case when the pattern is aligned along the columns/rows, compacting the image in the direction of alignment can help to make the PN visible, but this is not the possible in general.

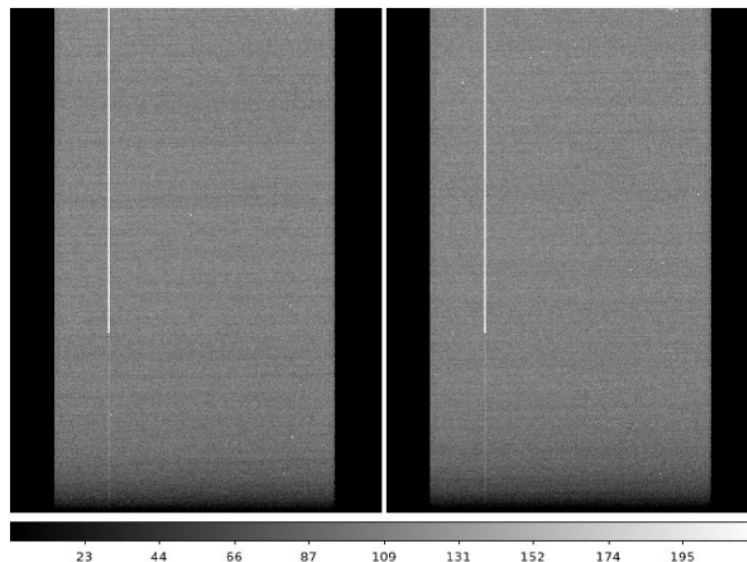


Figure I.0.1: Raw GIRAFFE bias images. The vertical stripe is a detector artifact. Here and in the next figures, except for Fig. I.0.2, the left and the right panels use the same stretch to facilitate easier comparison. Here and further the units are ADU.

Fortunately, the PN shows prominently as a set of structures in the power spectra. The power spectrum of an ideal image, without any pattern, is a perfect delta function with a width of one pixel (Fig. I.0.2). The power concentrated in that single peak reflects the simple pixel-to-pixel sensitivity variation that is corrected by flat fielding. Other peaks and structures in the power spectrum indicate the presence of systematic noises, e.g. patterns.

However, if the analysed image is a science frame and contains astronomical objects, these objects themselves will introduce peaks in the power spectrum that correspond to variations on the scales of these objects. Therefore, the search for PN should be performed on calibration frames, e.g. biases.

The function described here calculates the 2D power spectrum of an input image, then masks a region around the peak corresponding to the pixel-to-pixel variations, and then calculates on the remaining area of the power spectrum the root mean square (RMS) and the mean-absolute-deviation (MAD). A high RMS indicates the presence of power in variations of any spatial scale and type of PN. The MAD is used as a reference level – it indicates the typical level of the random variations of any scale.

Summarizing, the advantage of this diagnostics is that it is sensitive to the PN of any spatial scale or spatial pattern type, and it is not sensitive to the pixel-to-pixel variations.

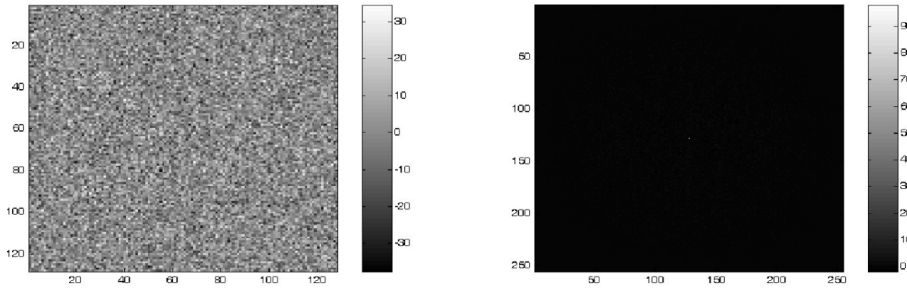


Figure I.0.2: **Left Panel:** Random noise image with no patterns. **Right Panel:** Its power spectrum (delta function at the centre).

Disclaimer: if the described power spectrum analysis is applied directly on real raw data, it may be affected by any structures such as cosmetic defects (e.g. bad columns/areas) and count decrease at the edges (e.g. due to vignetting). These changes are systematic in nature and they will show prominently in the power spectra, making the signatures of the transient PN almost invisible (Fig. I.0.3).

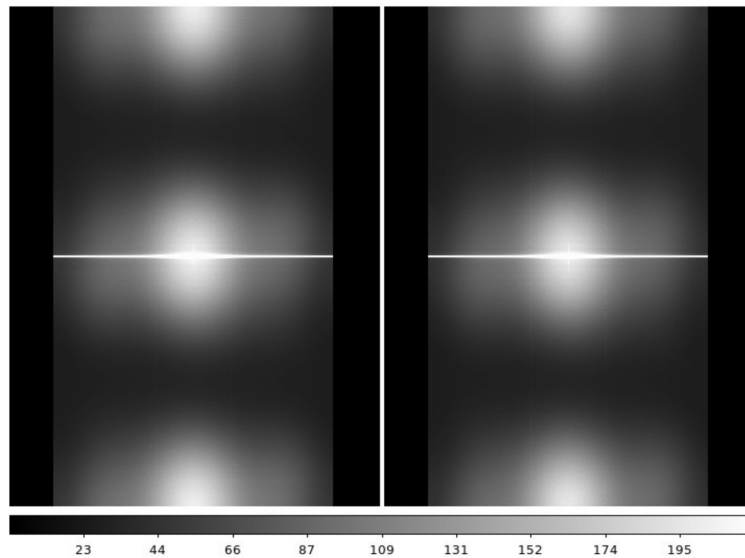


Figure I.0.3: Power spectra of the raw GIRAFFE bias images.

Therefore, some pre-reduction may be necessary: removing the large artefacts from the images, repairing the cosmic rays (in our case – replacing the affected pixels with averages from their neighbours), and taking out any stable known global patterns like any permanent gradients in the illumination or bias level. Figure I.0.4 shows an example of a “repaired” GIRAFFE images).

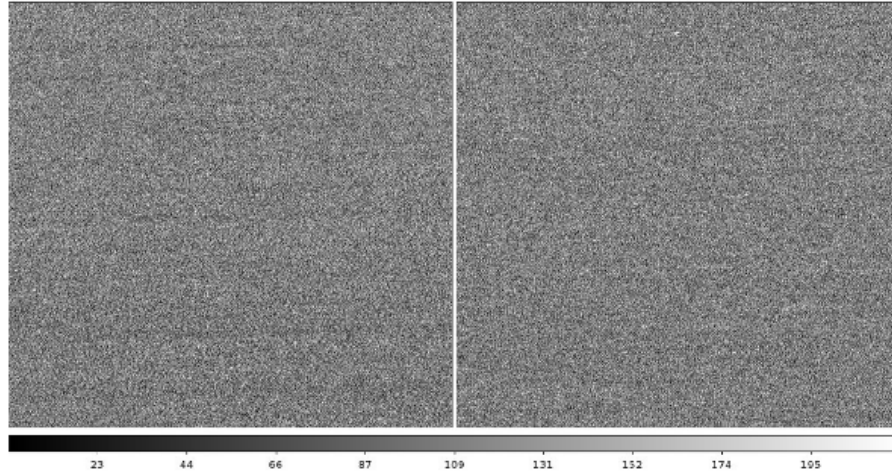


Figure I.0.4: “Repaired” GIRAFFE bias images.

The power spectra of the “repaired” images shows great improvement as seen in Fig. I.0.5 – compare the weak power present on the left image, and strong power on the right image.

The time scale of the PN variations must also be taken into account in this analysis, especially if one wants to apply the detection algorithm on combined master frames of some kind, e.g. on master biases. If the PN varies too quickly, the combination into master bias will average over many different types of PN and the resulting image will be unrealistically “clean”. In such cases the analysis should be carried out on individual raw images. It is important to investigate the PN variation time scale before adopting a procedure for a particular instrument/detector.

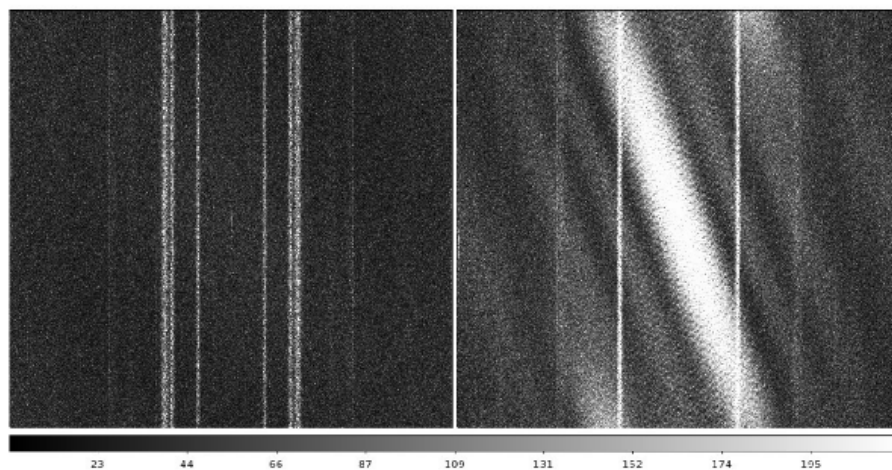


Figure I.0.5: Power spectra of the “repaired” GIRAFFE bias. The image on the left shows no pattern noise, the image on the right is affected by pattern noise.



## J Image and Cube Resampling

### J.1 Introduction

The *hdrldemo\_interpolation* implementation consists of a select number of general-use interpolation algorithms for sub-pixel re-gridding in the co-addition of 2D images (to be used in a stacking recipe added to the ESOTK package) and in the reconstruction of 3D data cubes. The recipes can take images or cubes as input, interpolate them to a common grid and, for the 2D images, co-add the re-gridded images using stacking functions already available in HDRL.

General interpolation methods can be relevant to any ESO instrument, however, the core application for the interpolation routines will be the combination of multiple, dithered frames to a sub-pixel, re-gridded reference frame. This has applications for both 2D image combination (FORS2) and for 3D data cube reconstruction with the integral field spectrographs, HARMONI, ERIS/SPIFFIER, MUSE, and KMOS.

The interpolation palette of *hdrldemo\_resample* consists of the resampling methods NEAREST, LINEAR, QUADRATIC, RENKA, LANCZOS, and DRIZZLE. The specific attributes of these interpolation algorithms were specified in the original proposal ([8]) and include the following:

1. retain good image quality following the interpolation. For example, their use in a pipeline should not increase the FWHM of a Nyquist-sampled Gaussian by more than 20% (see, for example, [2]);
2. conserve the detected flux over the sampled scale;
3. allow an estimate of the interpolation error, which is used to compute the variance;
4. retain the correct pixel status (i.e. propagate bad pixels or exclude them by performing a sigma clipping or another rejection method);
5. can be, when possible, multi-threaded in order to keep the computation times reasonable;

The goal of this report is to verify the use of the six interpolation methods on various types of data (synthetic images, HAWK-I images, strongly rotated images, and 3D data cubes), and to test whether all specific goals have been achieved.

### J.2 Testing the *hdrldemo\_resample* Routines Using Synthetic Data

To test the *hdrldemo\_resample* routines a series of synthetic exposures were created and interpolated. This data includes both single images and a series of 10 dithered exposures.

The synthetic images are made using a dedicated Python routine (*synthetic\_images.py*) that uses IRAF the functions *iraf.mknoise* to create the image background, *iraf.starlist* to create a list of stars with given distributions and brightness limits, and *iraf.mkobject* to convert this list to images with a *Moffat* profile, a given exposure time, and a given magnitude zero point. In order to test the changes that the interpolation introduces to the images, only stellar sources were added to the synthetic images. The format of the synthetic images is a three-plane MEF consisting of the image data, the image error, and the quality map. A summary of the attributes of the synthetic images include:

size: 1024×2048 pixels  
background: 500 ADU  
gain: 2.5 e<sup>-</sup>/ADU  
readnoise: 5.0 e<sup>-</sup>  
N<sub>sources</sub>: 1000  
exposure time: 100 sec.  
mag zeropoint: 14.0  
N<sub>dithers</sub>: 10  
dither\_step: 30 pixels

These single and 10 dithered images were then resampled using each of the methods available in *hdrldemo\_resample*, NEAREST, LINEAR, QUADRATIC, RENKA, LANCZOS, and DRIZZLE, and each was executed with both a loop distance of 1 and 3 pixels (`-method.loop-distance`). The sources within the interpolated images are then compared to the sources in the original input images.

An example of two representative *hdrldemo\_resample* products are shown in figure J.1.

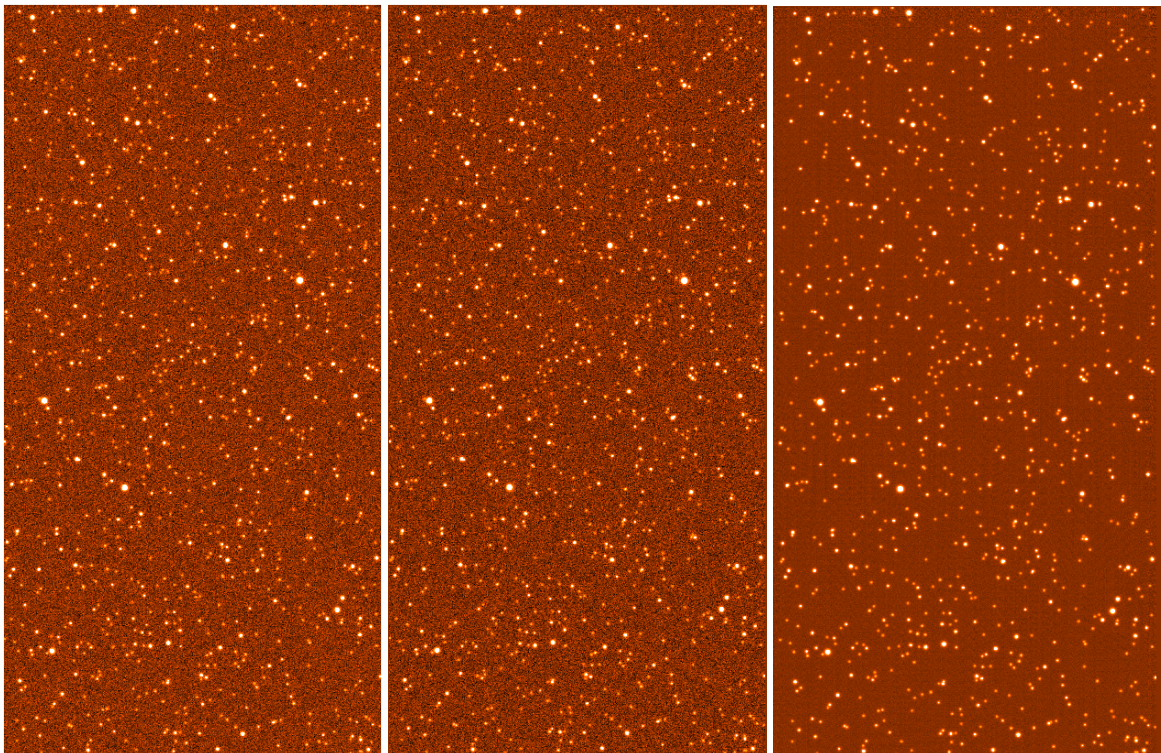


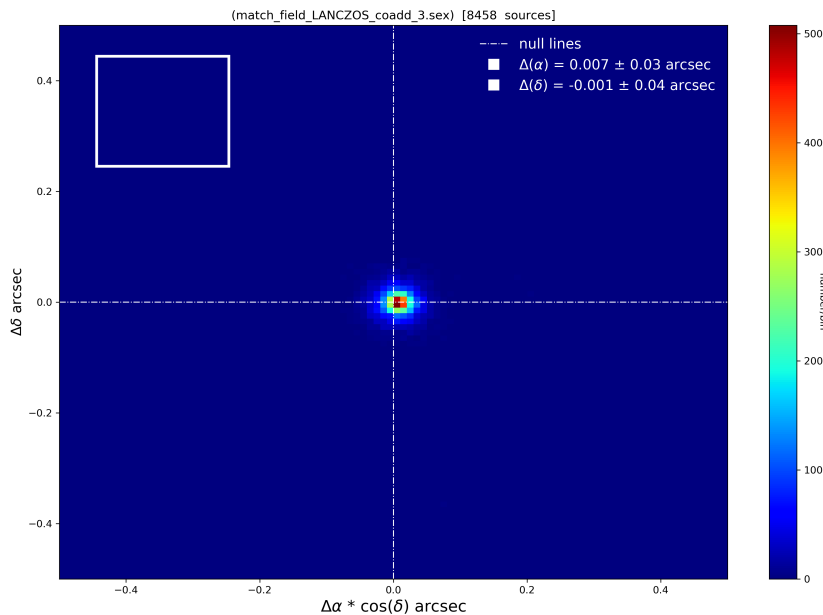
Figure J.1: **Left Panel:** An input synthetic image with 1000 stellar sources.

**Centre Panel:** The resampled single synthetic frame using *hdrldemo\_resample* with `-method=DRIZZLE` and `-method.loop-distance=3`.

**Right Panel:** The resampled tile of 10 synthetic frame using *hdrldemo\_resample* with `-method=DRIZZLE` and `-method.loop-distance=3`.

The comparison of the data products is done using a python script (`compare_image_sex.py`). This script runs SExtractor ([1]) on both images with a `-DETECT_THRESH` and `-ANALYSIS_THRESH` of  $5.0\sigma$  and a `-DETECT_MINAREA` of 10 pixels. A spherical, nearest-neighbour match within 1 arcsec is then done between the sources of the two resultant SExtractor catalogues. This allows us to compare the attributes (position  $(\alpha, \delta)$ , magnitude, FWHM, and ellipticity) of the ensemble of extracted sources from the input synthetic images to the same sources once they have been resampled by each of the six interpolation methods.

In all five methods of image resampling, the astrometric quality of the sources is retained following interpolation. An example of this is shown in Figure J.2. Here, the absolute positions of the sources in the 10 synthetic images are compared to the absolute positions of the sources in the image resample using the lanczos method. Here, the median  $\Delta\alpha * \cos(\delta) = 0.007 \pm 0.03 \text{ arcsec}$  and  $\Delta\delta = -0.001 \pm 0.04 \text{ arcsec}$  with the cloud of offsets spread symmetrically about the origin. Considering that the synthetic image pixel size is 0.20 pixels/arcsec, the astrometric accuracy is retained to better than a  $\frac{1}{5}$  of one pixel (0.04 pixels).



**Figure J.2:** the astrometric quality of the `hdrldemo_resample` routines as measured by comparing the more than 8,000 sources in the synthetic input images, with those in the resampled (LANCZOS) image tile. The standard deviation of the  $\Delta\alpha * \cos(\delta)$  and  $\Delta\delta$  distributions is 0.04 arcsec. This is significantly less than one pixel (0.20 pixels/arcsec). The pixel size is indicated by the white square in the top left corner.

Similarly, the photometric quality is also retained following interpolation. Figure J.3 is typical of the synthetic interpolated frames, with a source magnitude match between the individual images and the resampled (lanczos) image tile of  $\Delta(mag) = 0.006 \pm 0.05$  magnitudes.

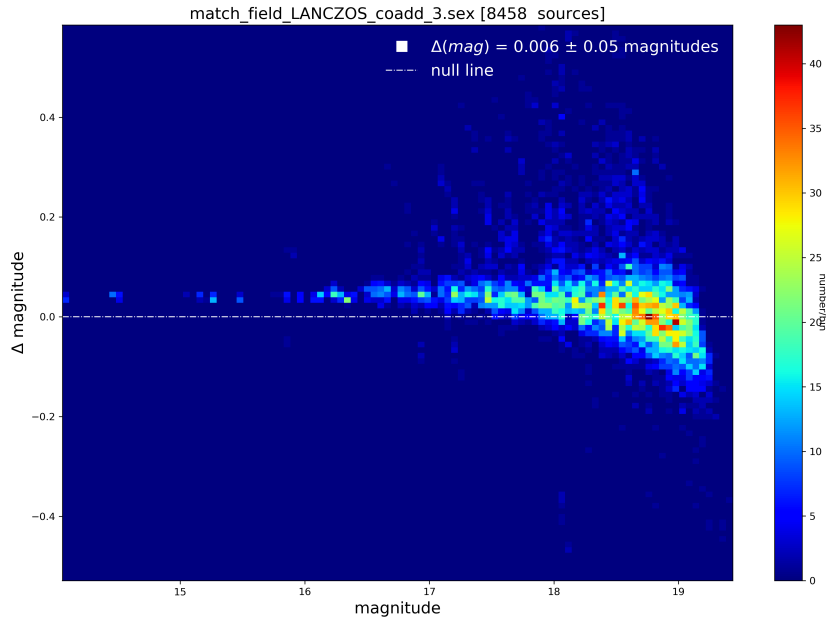


Figure J.3: the photometric quality of the *hdrldemo\_resample* routines as measured by comparing the more than 8,000 sources in the original input synthetic images with those in the resampled (lanczos) image tile. The standard deviation of the  $\Delta(\text{mag})$  distribution is 0.05 magnitudes.

As can be seen in figure J.4, the resampling (in this case using `-method=LINEAR`) causes a slight increase in the FWHM of the sources. This is expected for any interpolation method. As can be seen in the left panel of Figure J.4, the resampling tightens the distribution of FWHM and slightly increases it. This is true for all resampling methods tested.

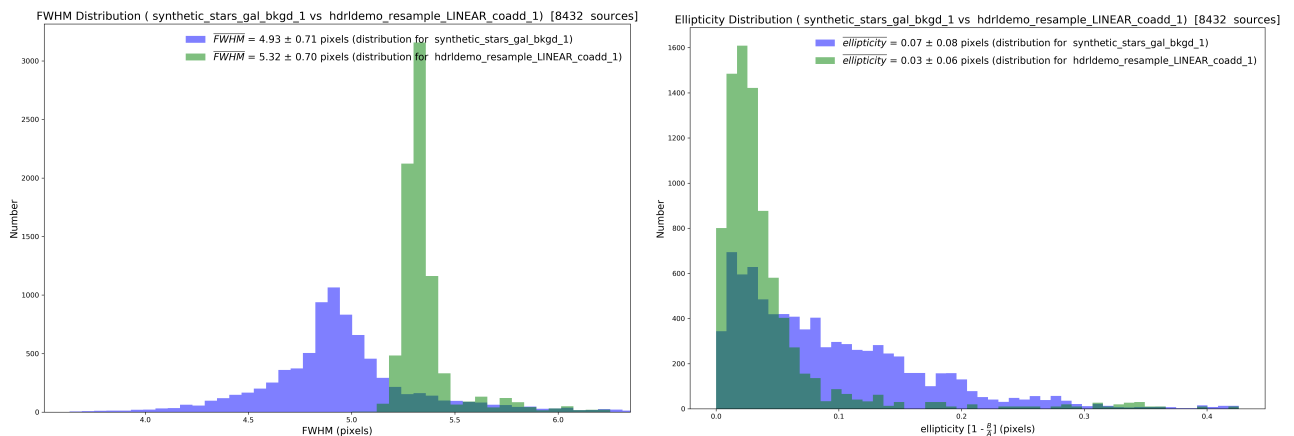


Figure J.4: A comparison of FWHM and ellipticity for the sources in the synthetic input images (blue histograms) and those after resampling (green histograms). The resampling was done with `-method=LINEAR` and `-method.loop-distance=1`.  
**left panel:** The FWHM distribution of the 8,000 sources in the synthetic field. As expected, the median FWHM, after resampling, increases slightly. The increase in FWHM is less than 8%.  
**right panel:** The distribution of ellipticities. Here, the resampling has circularised the sources. This, too, is as expected.



A summary of the source attributes between the input frames and the resampled images is given in Table J.2.0, in which the source attributes of all six resampling methods are compared for both a single synthetic image input and a resampling of 10 synthetic images.

The largest change in the average position of the sources is  $\Delta\alpha=0.1$  arcsec and  $\Delta\delta=-0.008$  arcsec for the resampling `method=NEAREST`. Since this is still only one half of a pixel, this is still acceptable. The largest change in the average magnitude can be found in the `LINEAR` resampling and corresponds to a small shift of  $\Delta\text{mag.} = 0.06$  magnitudes. The largest change in FWHM is seen for `method=LINEAR`, in which the FWHM of the synthetic stars increase by 1.8 pixels (37%). A measure of the source ellipticities shows a congruent feature. In all resampling methods the ellipticities are decreased and the distribution is tightened, as the interpolation has the general effect of circularising the source images.

### J.3 Results from HAWK-I Globular Cluster Images

To test the `hdrldemo_resample` routines a series of large-field HAWK-I exposures were processed. This data includes the globular cluster fields mapping M30 and NGC288. The data for M30 is from 2015-10-31 with `OBS.ID=200203309`, while the data for NGC288 is from 2016-11-22 with `OBS.ID=200203307`. Both data sets have `PROG.ID=60.A-9800(L)`.

All data has been processed using the HAWK-I pipeline version 2.4.6, with the intermediate pipeline science products (`BASIC_CALIBRATED_SCI` and `BASIC_VAR_MAP`) acting as input for the `hdrldemo_interpolation` and image combination routines. Each data set consisted of 25, calibrated science frames (dark-corrected, gain-corrected, flat-fielded, and astrometrically and photometrically calibrated), with each being a MEF of four detectors for a total of 100 individual exposures. These 100 images were resampled using each of the methods available in `hdrldemo_resample`, `NEAREST`, `LINEAR`, `QUADRATIC`, `RENKA`, `LANCZOS`, and `DRIZZLE`, and each was executed with both a loop distance of 1 and 3 pixels (`-method.loop-distance`). The resulting image tiles are then compared to the original input images.

The comparison of the data products is done using a python script (`compare_image_sex.py`). This script runs SExtractor ([1]) on both images with a `-DETECT_THRESH` and `-ANALYSIS_THRESH` of  $5.0\sigma$  and a `-DETECT_MINAREA` of 10 pixels. A spherical, nearest-neighbour match within 1 arcsec is then done between the sources of the two resultant SExtractor catalogues. This allows us to compare the attributes (position ( $\alpha, \delta$ ), magnitude, FWHM, and ellipticity) of the ensemble of extracted sources from the original input images to the same sources once they have been resampled by each of the six interpolation methods.

#### J.3.1 NGC288

An example of the `hdrldemo_resample` product is shown in figure J.5. A comparison with the combined frames from the HAWK-I pipeline (combined with `hawki_science_postprocess` show qualitatively very similar results.

Table J.2.0: A Summary of Comparisons Between Input Synthetic Data Images and Resultant Interpolation Images

synthetic_image	Interpolation method (LD) <sup>1</sup>	$N_{frames}^2$	Nmatch	$\Delta\alpha$ (arcsec)	$\Delta\delta$ (arcsec)	$\Delta(mag)$	$\sigma\Delta(mag)$	FWHM1 <sup>3</sup> (pixels)	FWHM2 <sup>4</sup> (pixels)	ellip1 <sup>3</sup>	ellip2 <sup>4</sup>
synthetic_stars_gal_bkgd_single	DRIZZLE (1)	single	893	0.000	-0.007	-0.005	0.014	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	DRIZZLE (3)	single	893	0.000	-0.007	-0.005	0.014	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	LANCZOS (1)	single	892	0.000	-0.003	-0.005	0.016	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	LANCZOS (3)	single	892	0.000	-0.004	-0.005	0.015	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	LINEAR (1)	single	895	0.000	-0.007	0.003	0.026	4.92	5.03	0.07	0.07
synthetic_stars_gal_bkgd_single	LINEAR (3)	single	892	0.000	-0.006	0.020	0.041	4.92	5.64	0.07	0.06
synthetic_stars_gal_bkgd_single	NEAREST (1)	single	893	0.000	-0.007	-0.005	0.014	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	NEAREST (3)	single	893	0.000	-0.007	-0.005	0.014	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	QUADRATIC (1)	single	890	0.000	-0.007	-0.004	0.015	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	QUADRATIC (3)	single	889	0.000	-0.007	-0.004	0.015	4.92	4.93	0.07	0.07
synthetic_stars_gal_bkgd_single	RENKA (1)	single	892	0.000	-0.007	-0.005	0.015	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_single	RENKA (3)	single	892	0.000	-0.007	-0.004	0.014	4.92	4.92	0.07	0.07
synthetic_stars_gal_bkgd_1→10	DRIZZLE (1)	10	8,587	0.001	-0.008	0.024	0.06	4.93	5.01	0.07	0.03
synthetic_stars_gal_bkgd_1→10	DRIZZLE (3)	10	8,587	0.001	-0.008	0.024	0.06	4.93	5.01	0.07	0.03
synthetic_stars_gal_bkgd_1→10	LANCZOS (1)	10	8,619	-0.017	0.003	0.024	0.06	4.93	4.97	0.07	0.03
synthetic_stars_gal_bkgd_1→10	LANCZOS (3)	10	8,629	0.007	-0.001	0.020	0.06	4.93	4.88	0.07	0.03
synthetic_stars_gal_bkgd_1→10	LINEAR (1)	10	8,603	0.056	-0.006	0.040	0.07	4.93	5.32	0.07	0.03
synthetic_stars_gal_bkgd_1→10	LINEAR (3)	10	8,508	0.051	-0.005	0.064	0.07	4.93	6.76	0.07	0.03
synthetic_stars_gal_bkgd_1→10	NEAREST (1)	10	8,390	0.099	-0.008	-0.007	0.01	4.93	4.91	0.07	0.07
synthetic_stars_gal_bkgd_1→10	NEAREST (3)	10	8,390	0.099	-0.008	-0.007	0.01	4.93	4.91	0.07	0.07
synthetic_stars_gal_bkgd_1→10	QUADRATIC (1)	10	8,589	0.027	-0.005	0.031	0.07	4.93	5.18	0.07	0.03
synthetic_stars_gal_bkgd_1→10	QUADRATIC (3)	10	8,569	0.018	-0.003	0.042	0.06	4.93	5.74	0.07	0.03
synthetic_stars_gal_bkgd_1→10	RENKA (1)	10	8,581	0.006	-0.005	0.029	0.07	4.93	5.07	0.07	0.03
synthetic_stars_gal_bkgd_1→10	RENKA (3)	10	8,581	0.003	-0.004	0.029	0.07	4.93	5.08	0.07	0.03

<sup>1</sup> The interpolation method and loop-distance used (-method, loop-distance).  
<sup>2</sup> The total number of synthetic input images combined during the resampling.  
<sup>3</sup> FWHM1 and ellip1 refer to the synthetic input images.  
<sup>4</sup> FWHM2 and ellip2 refer to the resampled images.

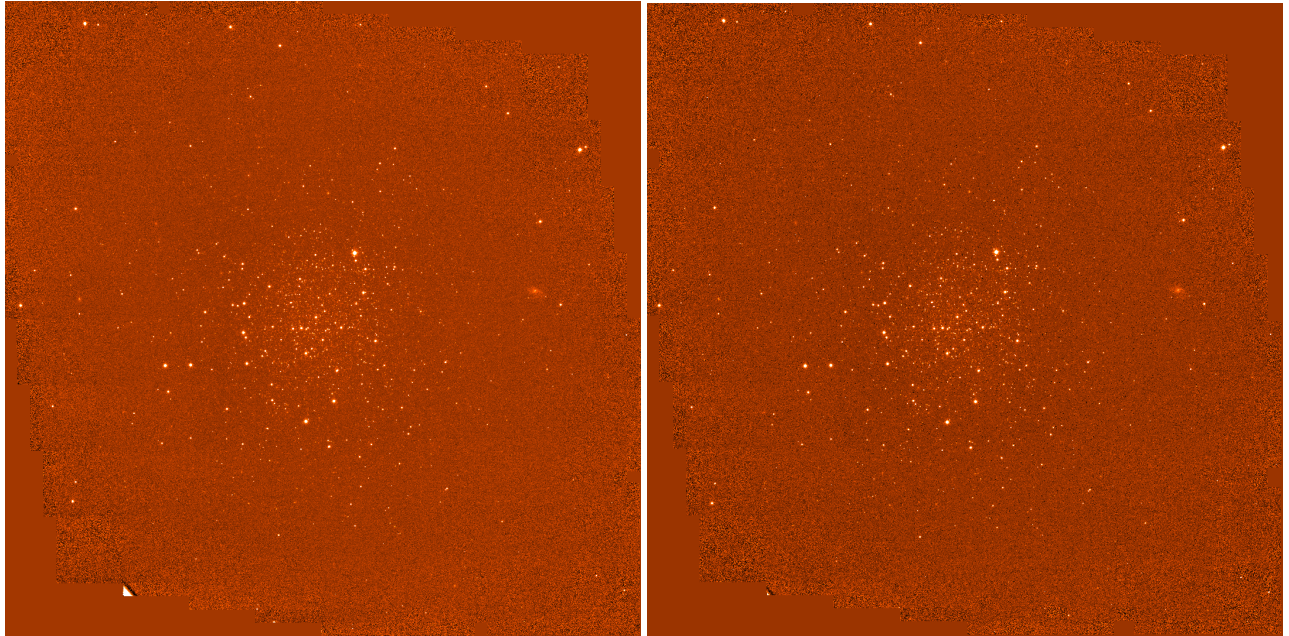


Figure J.5: The final tile image made from 100 HAWK-I images of the NGC 288 field:

**left panel:** as creating by the HAWK-I pipeline v. 2.4.6 routine `hawki_science_postprocess`

**right panel:** as resampled and combined using `hdrldemo_resample` with `-method=DRIZZLE` and `-method.loop-distance=1`.

In all five methods of image resampling, the astrometric quality of the sources is retained following interpolation. An example of this is shown in figure J.6. Here, the absolute positions of the sources in the 100 input HAWK-I images is compared to the absolute positions of the sources in the image resample using the drizzle method. Here, the median  $\Delta\alpha * \cos(\delta) = 0.001 \pm 0.07 \text{ arcsec}$  and  $\Delta\delta = 0.003 \pm 0.07 \text{ arcsec}$  with the cloud of offsets spread symmetrically about the origin. Considering that the HAWK-I pixel size is 0.106 pixels/arcsec, the astrometric accuracy is retained to better than a fraction of one pixel (0.66 pixels).

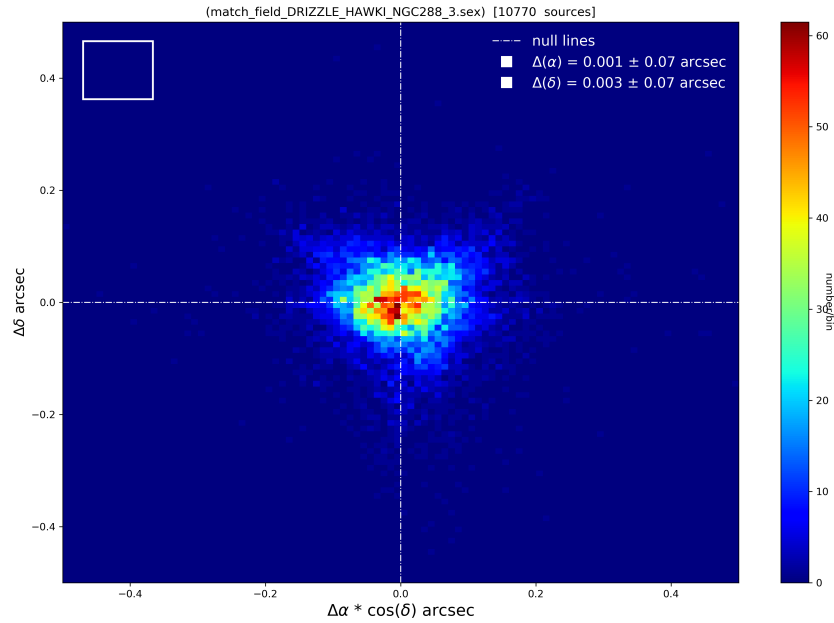


Figure J.6: the astrometric quality of the *hdrldemo\_resample* routines as measured by comparing the more than 10,000 sources in original input HAWK-I images with those in the resampled (drizzle) image tile of Figure J.5. The standard deviation of the  $\Delta\alpha * \cos(\delta)$  and  $\Delta\delta$  distributions is 0.07 arcsec. This is significantly less than one HAWK-I pixel (0.106 pixels/arcsec). The pixel size is indicated by the white square in the top left corner.

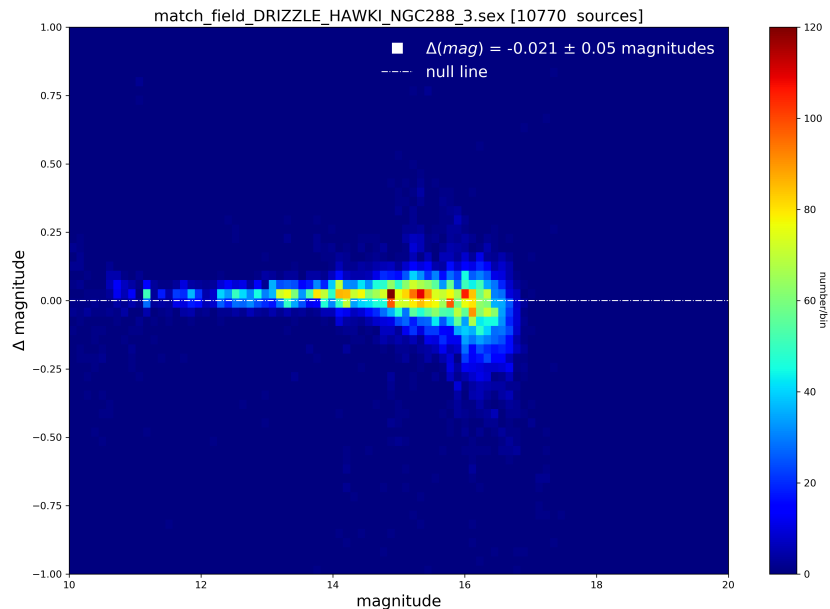


Figure J.7: the photometric quality of the *hdrldemo\_resample* routines as measured by comparing the more than 10,000 sources in original input HAWK-I images with those in the resampled (drizzle) image tile of Figure J.5 (right panel). The standard deviation of the  $\Delta(mag)$  distribution is 0.05 magnitudes.

Similarly, the photometric quality is also retained following interpolation. Figure J.7 is typical of all interpolated

frames, with a source magnitude match between the individual HAWK-I images and the resampled (drizzle) image tile of  $\Delta(mag) = -0.02 \pm 0.05$  magnitudes.

As can be seen in figure J.8, the resampling causes a slight increase in the FWHM of the sources and a decrease in the source ellipticities. This is expected, since the resampling will cause a circularisation and homogenisation of the source shapes.

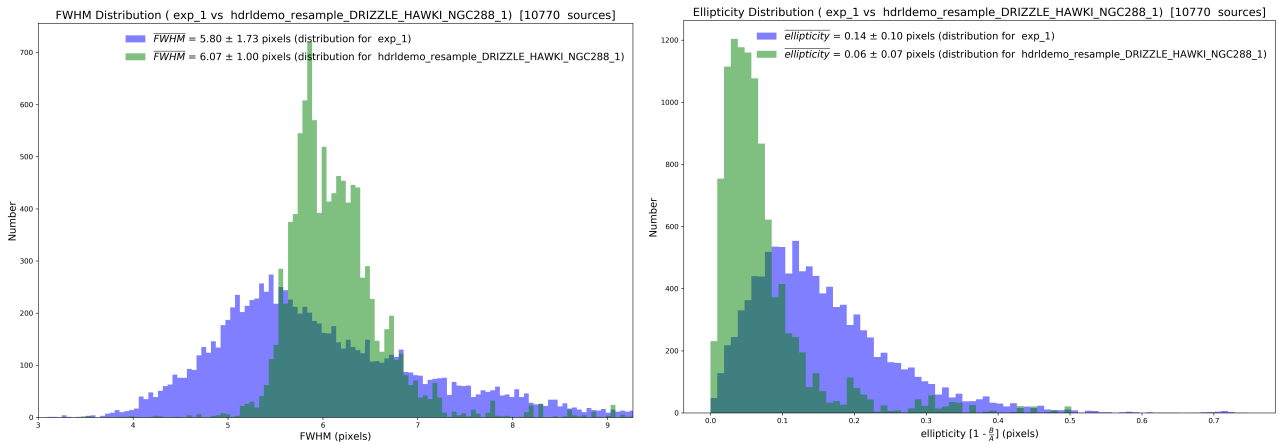


Figure J.8: A comparison of FWHM and ellipticity for the sources in the HAWK-I input images (blue histograms) and those after resampling (green histograms). The resampling was done with `-method=DRIZZLE` and `-method.loop-distance=1`.

**left panel:** The FWHM distribution of the 10,000 sources in the NGC288 field. As expected, the median FWHM, after resampling, increases slightly. Since the increase in FWHM is less than 5%, this is acceptable.

**right panel:** The distribution of ellipticities. Here, the resampling has circularised the sources. This, too, is as expected.

## J.3.2 M30

The analysis of the denser globular cluster M30 shows similar good results when the individual input HAWK-I images are compared to the resampled results. An example of the `hdrldemo_resample` product (`-method=DRIZZLE` and `-method.loop-distance=1`) is shown in figure J.9.

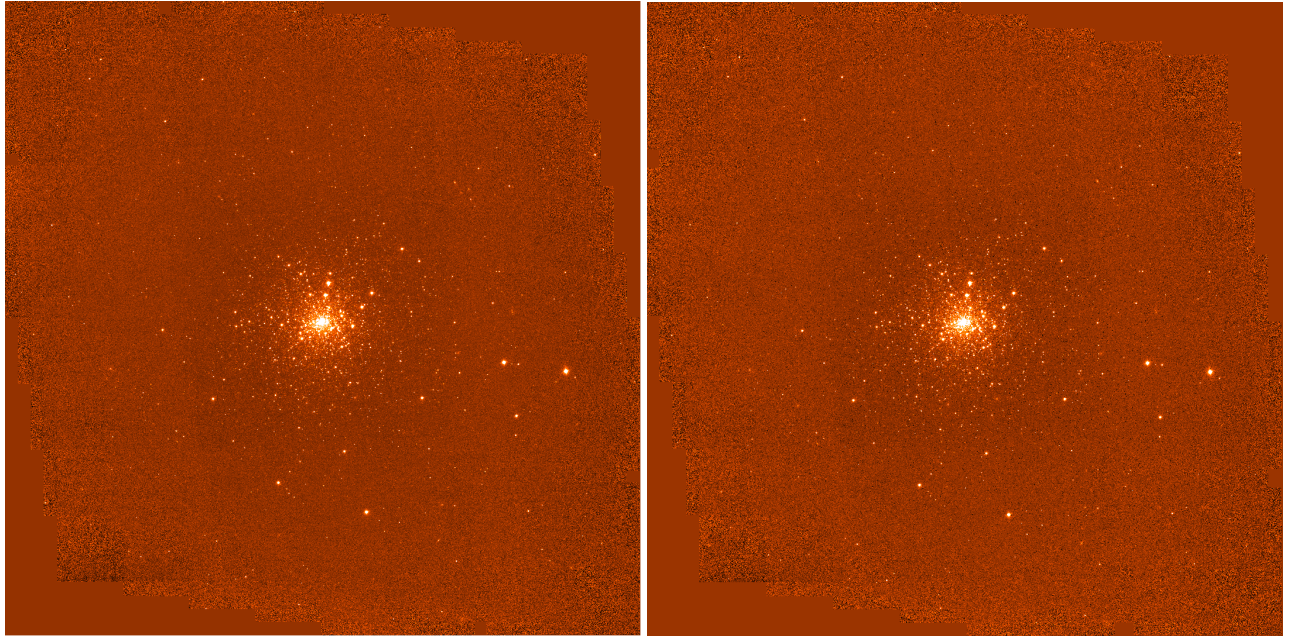


Figure J.9: The final tile image made from 100 HAWK-I images of the M30 field:

**left panel:** as creating by the HAWK-I pipeline v. 2.4.6 routine `hawki_science_postprocess`

**right panel:** as resampled and combined using `hdrldemo_resample` with `-method=DRIZZLE` and `-method.loop-distance=1`.

As in the previous HAWK-I example, all five methods of image resampling retained the astrometric quality following interpolation. This is evident in figure J.10. Here, the absolute positions of the sources in the 100 input HAWK-I images is compared to the absolute positions of the sources in the image resample using the QUADRATIC method. Here, the median  $\Delta\alpha \cdot \cos(\delta) = -0.002 \pm 0.11 \text{ arcsec}$  and  $\Delta\delta = -0.007 \pm 0.12 \text{ arcsec}$  with the cloud of offsets spread symmetrically about the origin. Considering that the HAWK-I pixel size is 0.106 pixels/arcsec, the standard deviation of the astrometric accuracy approximately one pixel.

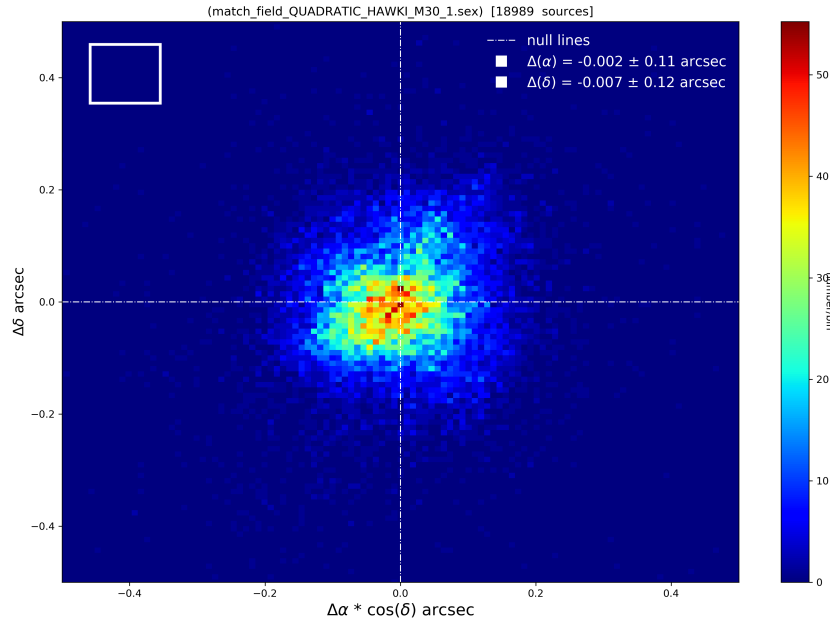


Figure J.10: the astrometric quality of the *hdrldemo\_resample* routines as measured by comparing almost 19,000 sources in the original input HAWK-I images with those in the resampled (QUADRATIC) image tile of Figure J.9. The standard deviation of the  $\Delta\alpha * \cos(\delta)$  and  $\Delta\delta$  distributions is 0.12 arcsec. This is approximately the size of one HAWK-I pixel (0.106 pixels/arcsec). The pixel size is indicated by the white square in the top left corner.

Similarly, the photometric quality is also retained following interpolation. Figure J.11 is typical of all interpolated frames, with a source magnitude match between the individual HAWK-I images and the resampled (LANCZOS) image tile of  $\Delta(mag) = -0.023 \pm 0.08$  magnitudes.

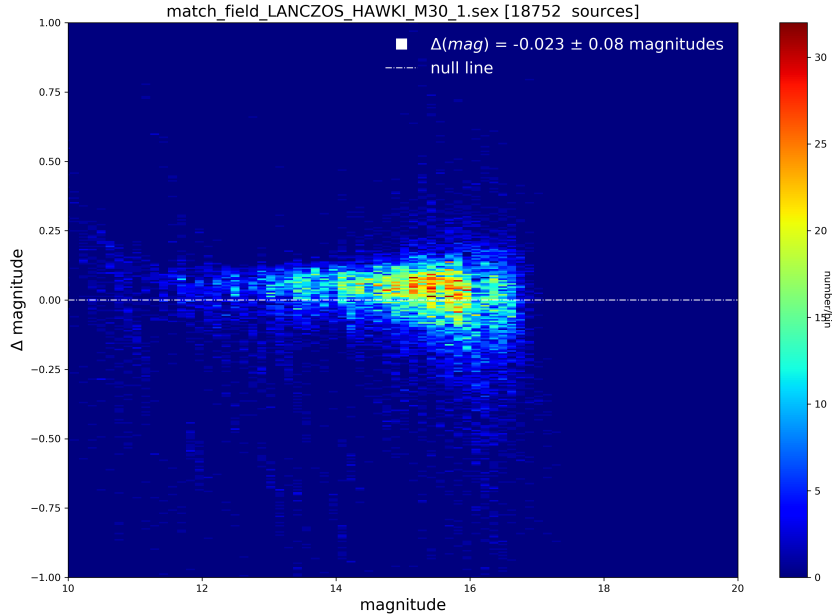


Figure J.11: the photometric quality of the *hdrldemo\_resample* routines as measured by comparing the more than 18,000 sources in the original input HAWK-I images with those in the resampled (LANCZOS) image tile of Figure J.9 (right panel). The standard deviation of the  $\Delta(mag)$  distribution is 0.08 magnitudes.

As can be seen in figure J.12, the resampling causes a slight increase in the FWHM of the sources and a decrease in the source ellipticities. This is expected, since the resampling will cause a circularisation and homogenisation of the source shapes.

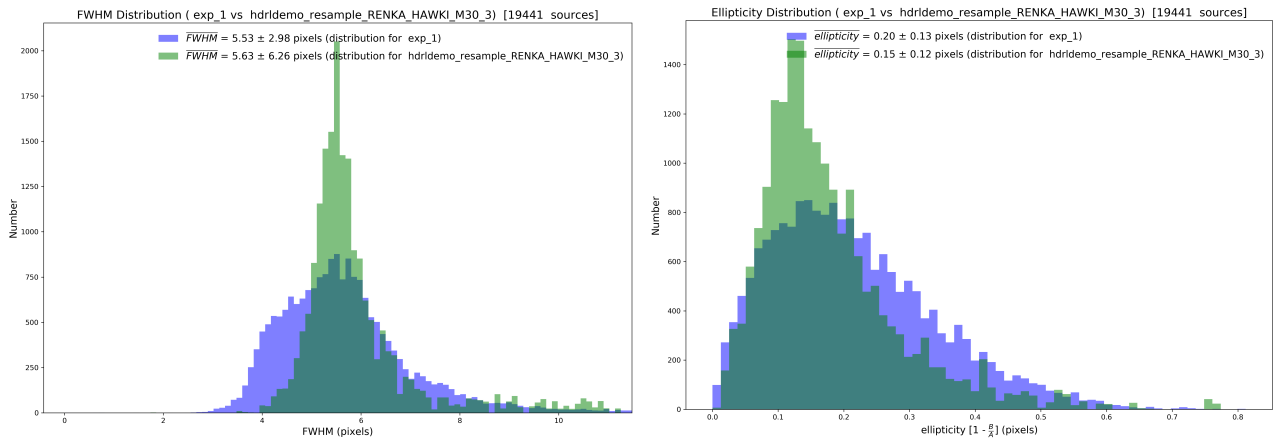


Figure J.12: A comparison of FWHM and ellipticity for the sources in the HAWK-I input images (blue histograms) and those after resampling (green histograms). The resampling was done with `-method=RENKA` and `-method.loop-distance=3`.

**left panel:** The FWHM distribution of the 19,441 sources in the M30 field. As expected, the median FWHM, after resampling, increases slightly. Since the increase in FWHM is less than 5%, this is acceptable.

**right panel:** The distribution of ellipticities. Here, the resampling has circularised the sources. This, too, is as expected.

A summary of the source attributes between the input frames and the resampled images is given in Table J.3.0.

Table J.3.0: A Summary of Comparisons Between Input HAWK-I Cluster Data Images and Resultant Interpolation Images

Image	Interpolation method (LD) <sup>1</sup>	$N_{frames}^2$	Nmatch	$\Delta\alpha$ (arcsec)	$\Delta\delta$ (arcsec)	$\Delta(mag)$	$\sigma\Delta(mag)$	FWHM1 <sup>3</sup> (pixels)	FWHM2 <sup>4</sup> (pixels)	ellip1 <sup>3</sup>	ellip2 <sup>4</sup>
NGC 288	DRIZZLE (1)	100	10,770	0.001	0.003	-0.021	0.05	5.80	6.07	0.14	0.06
	DRIZZLE (3)	100	10,770	0.001	0.003	-0.021	0.05	5.80	6.07	0.14	0.06
	LANCZOS (1)	100	10,768	0.001	0.003	-0.025	0.05	5.80	6.01	0.14	0.05
	LANCZOS (3)	100	10,760	0.000	0.003	-0.029	0.05	5.80	5.93	0.14	0.05
	LINEAR (1)	100	10,794	0.002	0.003	-0.012	0.05	5.80	6.33	0.14	0.06
	LINEAR (3)	100	10,782	0.003	0.004	0.015	0.06	5.80	7.56	0.14	0.06
	NEAREST (1)	100	9,879	-0.001	0.002	-0.044	0.06	5.80	5.82	0.13	0.12
	NEAREST (3)	100	9,879	-0.001	0.002	-0.044	0.06	5.80	5.82	0.13	0.12
	QUADRATIC (1)	100	10,739	0.001	0.004	-0.023	0.05	5.80	6.14	0.14	0.06
	QUADRATIC (3)	100	10,724	0.002	0.004	-0.015	0.05	5.80	6.52	0.14	0.06
RENKA (1)	100	10,718	0.000	0.004	-0.030	0.05	5.80	6.04	0.14	0.07	
RENKA (3)	100	10,718	0.000	0.004	-0.030	0.05	5.80	6.04	0.14	0.07	
M 30	DRIZZLE (1)	100	18,678	-0.002	-0.006	-0.020	0.08	5.51	5.53	0.19	0.14
	DRIZZLE (3)	100	18,678	-0.002	-0.006	-0.020	0.08	5.51	5.53	0.19	0.14
	LANCZOS (1)	100	18,752	-0.002	-0.007	-0.023	0.08	5.51	5.49	0.19	0.14
	LANCZOS (3)	100	18,852	-0.002	-0.006	-0.026	0.08	5.51	5.42	0.19	0.14
	LINEAR (1)	100	18,187	-0.002	-0.005	-0.008	0.09	5.50	5.77	0.19	0.13
	LINEAR (3)	100	16,999	-0.001	-0.004	0.025	0.09	5.50	6.97	0.19	0.13
	NEAREST (1)	100	19,189	-0.006	-0.004	-0.052	0.11	5.59	5.75	0.20	0.21
	NEAREST (3)	100	19,189	-0.006	-0.004	-0.052	0.11	5.59	5.75	0.20	0.21
	QUADRATIC (1)	100	18,989	-0.002	-0.007	-0.026	0.09	5.52	5.66	0.19	0.14
	QUADRATIC (3)	100	18,499	-0.003	-0.005	-0.016	0.09	5.51	5.97	0.19	0.13
RENKA (1)	100	19,413	-0.003	-0.007	-0.034	0.09	5.53	5.63	0.20	0.15	
RENKA (3)	100	19,441	-0.003	-0.007	-0.034	0.09	5.53	5.63	0.20	0.15	

<sup>1</sup> The interpolation method and loop-distance used (`-method.loop-distance`).

<sup>2</sup> The total number of synthetic input images combined during the resampling.

<sup>3</sup> FWHM1 and ellip1 refer to the individual, HAWK-I input images.

<sup>4</sup> FWHM2 and ellip2 refer to the resampled images.

#### J.4 Results from Strongly Rotated HAWK-I Images

In order to test for any flexure in the HAWK-I instrument, an observing block (OB) was designed that allows the camera co-rotator to travel through  $160^\circ$  while pointing at the same field of view. This data set was obtained on the night of 2015-08-28 and has programme id 60.A-9283 (A) and the observing block id 200346422. It consists of 20 separate exposures of four detectors, with each pair of images rotated by  $20^\circ$  for a total of  $160^\circ$ . Clearly, a complex data set consisting of 80 distinct detector frames with such a large overall rotation is an ideal stress test in order to see how well the *hdrldemo\_resample* routines perform.

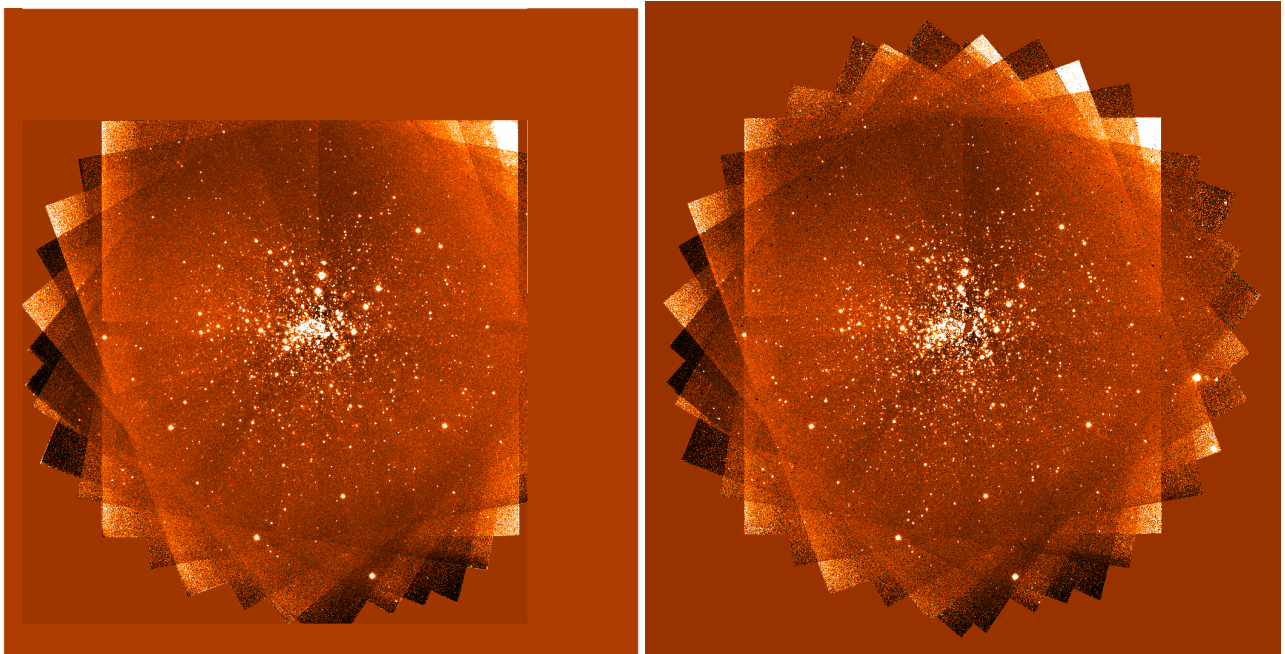


Figure J.13: The final tile image made from 80 HAWK-I flexure frames:

**left panel:** Created by the HAWK-I pipeline v. 2.4.6 routine `hawki_science_postprocess`

**right panel:** Resampled and combined using *hdrldemo\_resample* with `-method=DRIZZLE` and `-method.loop-distance=1`. The only major difference between the two images is that the HAWK-I pipeline trims the tile, whereas *hdrldemo\_resample* creates the a maximal area occupied by the WCS from all input images.

The  $20 \times 4$  separate exposures were resampled using each of the methods available in *hdrldemo\_resample*, NEAREST, LINEAR, QUADRATIC, RENKA, LANCZOS, and DRIZZLE, each with a loop distance of 1 and 3 pixels. The resulting image tiles were then compared to the tile resulting from processing the identical data with the HAWK-I pipeline version 2.4.6.

As in the previous HAWK-I example, all five methods of image resampling retained the astrometric quality following interpolation. This is evident in figure J.14. Here, the absolute positions of the sources in the 80 input HAWK-I images is compared to the absolute positions of the sources in the image resample using the LINEAR method. Here, the median  $\Delta\alpha \cdot \cos(\delta) = -0.004 \pm 0.13 \text{ arcsec}$  and  $\Delta\delta = -0.011 \pm 0.14 \text{ arcsec}$  with the cloud of offsets spread symmetrically about the origin. Considering that the HAWK-I pixel size is 0.106 pixels/arcsec, the standard deviation of the astrometric accuracy approximately one pixel.

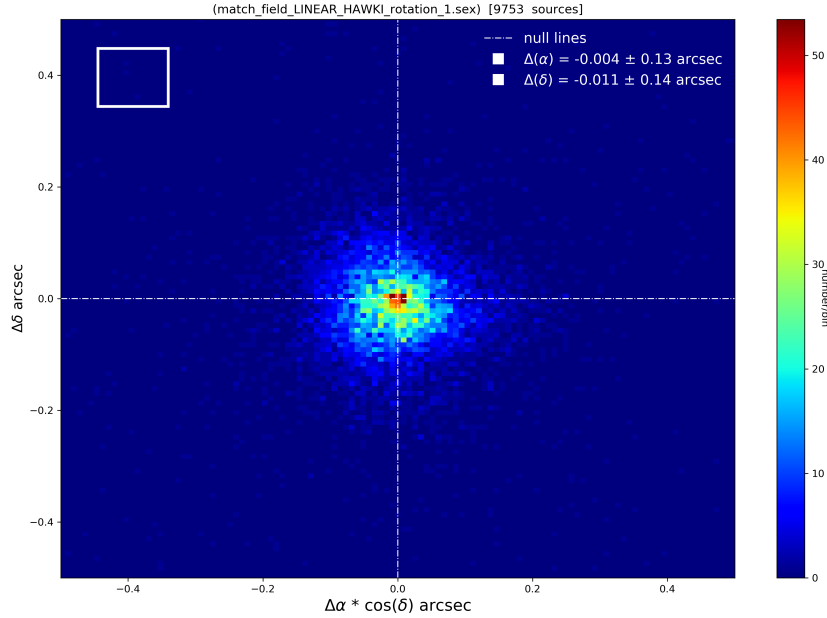


Figure J.14: the astrometric quality of the *hdrldemo\_resample* routines as measured by comparing 9,753 sources in the original input HAWK-I images with those in the resampled (LINEAR) image tile of Figure J.13. The standard deviation of the  $\Delta\alpha * \cos(\delta)$  and  $\Delta\delta$  distributions is 0.14 arcsec. This is approximately the size of one HAWK-I pixel (0.106 pixels/arcsec). The pixel size is indicated by the white square in the top left corner.

Similarly, the photometric quality is also retained following interpolation. Figure J.15 is typical of all interpolated frames, with a source magnitude match between the individual HAWK-I images and the resampled (LINEAR) image tile of  $\Delta(mag) = -0.18 \pm 0.13$  magnitudes.

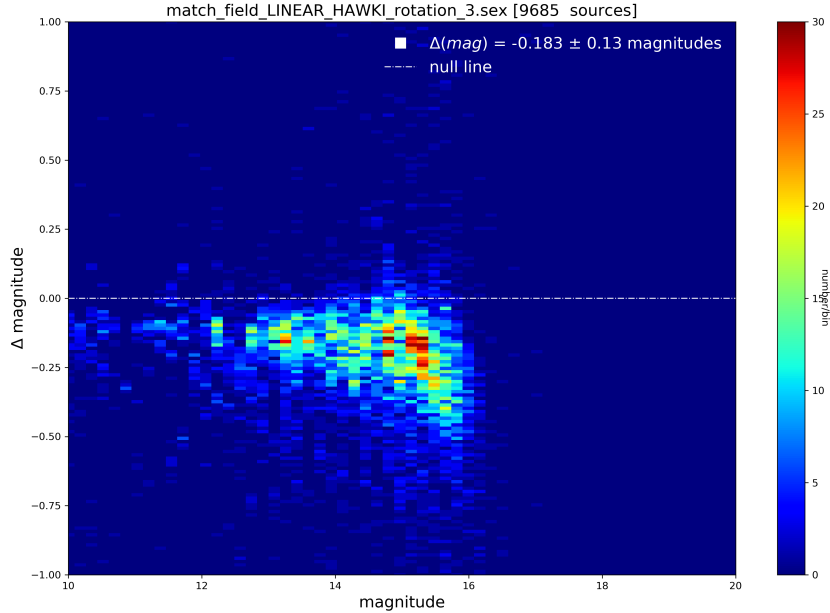


Figure J.15: the photometric quality of the *hdrldemo\_resample* routines as measured by comparing the 9,685 sources in the original input HAWK-I images with those in the resampled (LINEAR) image tile of Figure J.13 (right panel). The standard deviation of the  $\Delta(mag)$  distribution is 0.13 magnitudes.

As can be seen in figure J.16, the resampling causes a slight increase in the FWHM of the sources and a decrease in the source ellipticities. This is expected, since the resampling will cause a circularisation and homogenisation of the source shapes.

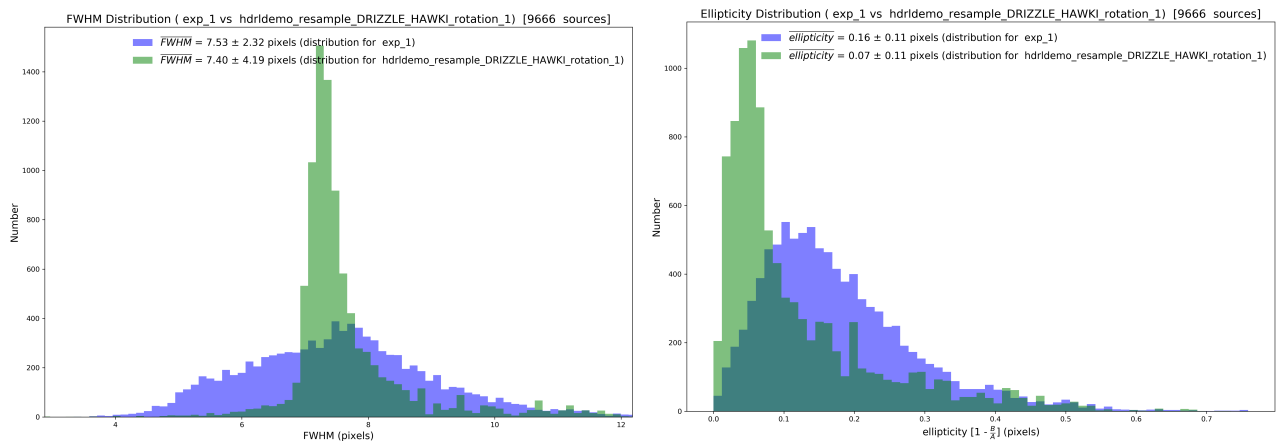


Figure J.16: A comparison of FWHM and ellipticity for the sources in the HAWK-I input images (blue histograms) and those after resampling (green histograms). The resampling was done with `-method=DRIZZLE` and `-method.loop-distance=1`.

**left panel:** The FWHM distribution of the 9,666 sources in the flexure field. The median FWHM, after resampling, decreases slightly. However, this is certainly an artifact of the very broad, ill-defined distribution determined for the input HAWK-I images, being over-estimated.

**right panel:** The distribution of ellipticities. Here, the resampling has circularised the sources. This, too, is as expected.

A summary of the source attributes between the input frames and the resampled images is given in Table J.4.0.

## J.5 The Propagation of Edge Effects with Resampled Images

Astronomical detectors often have edge effects in which the outermost regions of the detector show strongly depressed flux levels. This can be due to a non-uniform illumination over the surface of the detector, or due to masked edges. This is particularly evident in some near infrared detectors.

An edge effect will be strongly enhanced by any resampling method as the algorithm interpolates across such an illumination discontinuity. This can be seen in figure J.17. Here, the images of M30 were resampled without first trimming their edges. This image should be compared to the right hand panel of figure J.9.

The *hdrldemo\_resample* suite will need to include a parameter that allows for the input images to be edge trimmed before being resampled and combined. The parameter should be defined as `-edge-trim=NN`, whereby NN sets the number of pixels trimmed from each of the four sides of each input image. This parameter has already been implemented in the HDRL pipeline kits from version 15 (*hdrldemo-kit-1.3.0-15*).

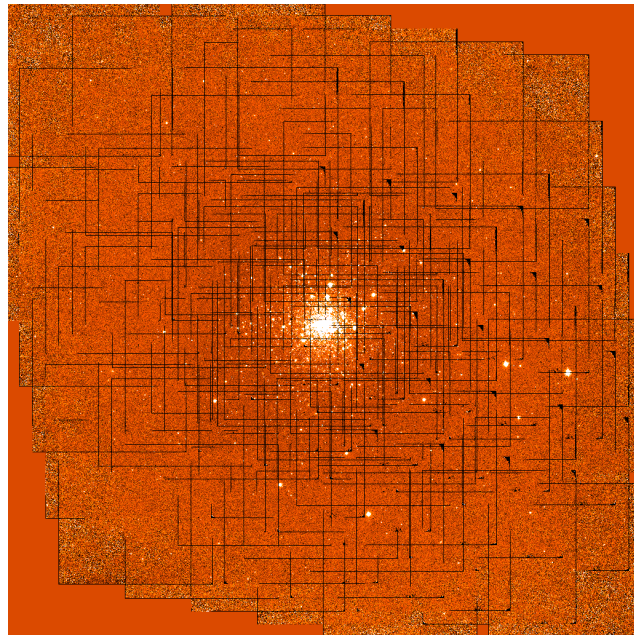


Figure J.17: An example of strong edge effects in the final tile image made from 100 HAWK-I images of the M30 field. This image has been resampled and combined using *hdrldemo\_resample* with `-method=DRIZZLE` and `-method.loop-distance=1`, without trimming the discontinuities at the edge of each input HAWK-I image.

## J.6 Results from Synthetic Images With Right Ascension values At the Meridian and Declination near the Poles

A potential pitfall can occur when images straddle discontinuities in the WCS coordinate system. Therefore, a series of 10 dithered synthetic images were created with a central pixel values set to reference points:

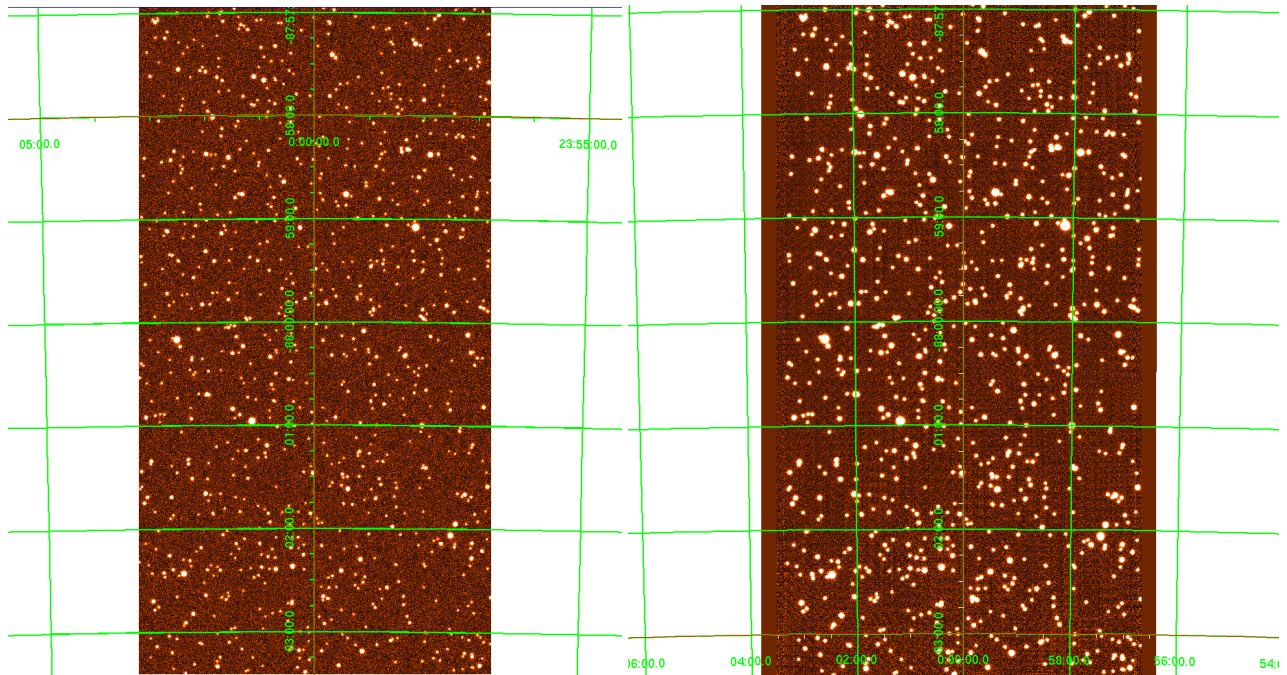
Table J.4.0: A Summary of Comparisons Between Input HAWK-I Flexure Images and Resultant Interpolation Images

Image	Interpolation method (LD) <sup>1</sup>	$N_{frames}^2$	Nmatch	$\Delta\alpha$ (arcsec)	$\Delta\delta$ (arcsec)	$\Delta(mag)$	$\sigma\Delta(mag)$	FWHM1 <sup>3</sup> (pixels)	FWHM2 <sup>4</sup> (pixels)	ellip1 <sup>3</sup>	ellip2 <sup>4</sup>
HAWK-I flexure	DRIZZLE (1)	80	9,666	-0.003	-0.010	-0.23	0.12	7.54	7.40	0.16	0.07
	DRIZZLE (3)	80	9,666	-0.003	-0.010	-0.23	0.12	7.54	7.40	0.16	0.07
	LANCZOS (1)	80	9,675	-0.003	-0.009	-0.23	0.12	7.54	7.36	0.16	0.07
	LANCZOS (3)	80	9,657	-0.003	-0.009	-0.23	0.12	7.54	7.30	0.16	0.07
	LINEAR (1)	80	9,753	-0.004	-0.011	-0.21	0.12	7.54	7.61	0.16	0.09
	LINEAR (3)	80	9,685	-0.004	-0.013	-0.18	0.13	7.54	8.66	0.16	0.11
	NEAREST (1)	80	7,788	-0.001	0.004	-0.26	0.13	7.54	7.79	0.16	0.14
	NEAREST (3)	80	7,788	-0.001	0.004	-0.26	0.13	7.54	7.79	0.16	0.14
	QUADRATIC (1)	80	9,444	-0.003	-0.008	-0.22	0.13	7.54	7.75	0.16	0.08
	QUADRATIC (3)	80	9,470	-0.004	-0.010	-0.21	0.13	7.54	8.00	0.16	0.08
	RENKA (1)	80	9,387	-0.004	-0.007	-0.24	0.12	7.54	7.72	0.16	0.08
	RENKA (3)	80	9,387	-0.004	-0.007	-0.24	0.12	7.54	7.72	0.16	0.08

<sup>1</sup> The interpolation method and loop-distance used (-method . loop-distance).  
<sup>2</sup> The total number of single, HAWK-I input images combined during the resampling.  
<sup>3</sup> FWHM1 and ellip1 refer to the original, HAWK-I input images.  
<sup>4</sup> FWHM2 and ellip2 refer to the resampled images.

$(\text{CRVAL1}, \text{CRVAL2}) = (\alpha_o, \delta_o) = (360^\circ, 88^\circ)$ .

These synthetic images were then resampled and combined using the six available *hdrldemo\_resample* methods. In each case, the resampled frames correctly retain the celestial coordinates of the original input images. This can be seen in figure J.18, in which the ds9 coordinate grid is overlaid on the images and displays the correct transition straddling the  $0^\circ$  meridian.



**Figure J.18:** A synthetic image and its resampled counterpart to test the *hdrldemo\_resample* routines at and near celestial discontinuities

**left panel:** A single synthetic frame with  $(\text{CRVAL1}, \text{CRVAL2}) = (\alpha, \delta) = 360^\circ, -88^\circ$ .

**right panel:** The ten synthetic resampled and combined using *hdrldemo\_resample* with `-method=DRIZZLE` and `-method.loop-distance=1`.

The overlaid ds9 coordinate grid shows that these extreme  $(\alpha, \delta)$  values are handled correctly.

## J.7 Testing the *hdrldemo\_resample* Routines Using MUSE Data Cubes

Tests on MUSE datacubes were performed on dithered observations of two targets: TNJ2009-304 (a protocluster at  $z \sim 3$ ) and FCC 184 (an early-type galaxy in the Fornax cluster).

The cube of TNJ2009-304 includes the combination of 6 dithered exposures (MUSE.2016-06-01T06:09:25.499.fits, MUSE.2016-06-01T07:05:35.072.fits, MUSE.2016-06-01T08:24:39.641.fits, MUSE.2016-06-01T09:19:35.838.fits, MUSE.2016-06-08T06:06:02.858.fits, and MUSE.2016-06-08T06:58:50.486.fits).

The cube of FCC 184 includes the combination of 10 dithered exposures (MUSE.2017-01-26T02:57:56.013.fits, MUSE.2017-01-26T03:16:12.338.fits, MUSE.2017-01-26T03:30:08.819.fits, MUSE.2017-01-26T03:49:23.358.fits, MUSE.2017-01-26T04:03:20.921.fits, MUSE.2017-11-15T06:57:38.494.fits, MUSE.2017-11-15T07:15:56.735.fits, MUSE.2017-11-15T07:29:49.482.fits, MUSE.2017-11-15T07:48:08.059.fits, and MUSE.2017-11-15T08:02:02.941.fits).

First, the data were reduced and combined with the MUSE pipeline (v2.8.3). This process foresees the combi-



Table J.7.0: Point spread function of point-like sources in the reconstructed field of views

ID	X pxl	Y pxl	RA [°]	DEC [°]	MOFFAT_MUSE pxl	MOFFAT_HDRL pxl
				TNJ2009-304		
1	40.48	285.49	302.4580383	-30.6613064	4.73	4.75
2	97.99	190.29	302.4543457	-30.6665955	4.49	4.56
3	168.09	214.34	302.4497986	-30.6652584	4.53	4.61
4	262.61	215.91	302.4436951	-30.6651707	5.78	5.85
5	254.20	189.76	302.4442444	-30.6666241	4.30	4.41
6	155.89	143.03	302.4505920	-30.6692200	4.78	4.82
				FCC184		
7	194.40	355.39	54.2386971	-35.4991646	4.50	4.59

nation of so-called pixel-tables, each of them represent the fully reduced exposure in detector space. The direct combination of pixel tables implies one single regridding process of the data.

The MUSE pipeline also produced the regridded cubes of the individual exposures, which we then combined (after taking into account spatial offsets in RA/DEC) with the HDRL recipe *hdrldemo\_resample*. This process implies two regriddings: the first is done by the MUSE pipeline to “convert” each pixel table into a datacube. The second is done by HDRL to regrid each cube into a common WCS before combining them.

The combination was done using the same parameters as in MUSE; drizzle method with downscaling factor 0.6 (spatial and spectral dimension), and loop-distance =1 pixel. The variance extension in the MUSE individual datacubes was also used.

Also, we set the starting and ending RA/DEC in the HDRL recipe so that the final cube spatially matches the MUSE combined cube. In this way a direct 1-to-1 spaxel comparison is possible.

The goals are to compare the quality of the spatial resolution and spectra obtained with two approaches; a direct combination of reduced pixel-tables with the MUSE pipeline, and the combination of individual pre-regridded cubes with HDRL.

### J.7.1 Spatial resolution

Figure J.19 compares the white-lamp images obtained by collapsing the MUSE and HDRL datacubes along the wavelength axis. The HDRL cubes are smoother in the spatial direction, as a consequence of the two-regridding steps that are involved in the process. Table J.7.0 compares the MOFFAT point spread function (measured with IRAF) of point-like sources measured on the reconstructed images. On average, the HDRL cubes have FWHM 1-2% higher than the corresponding source in the MUSE cube.

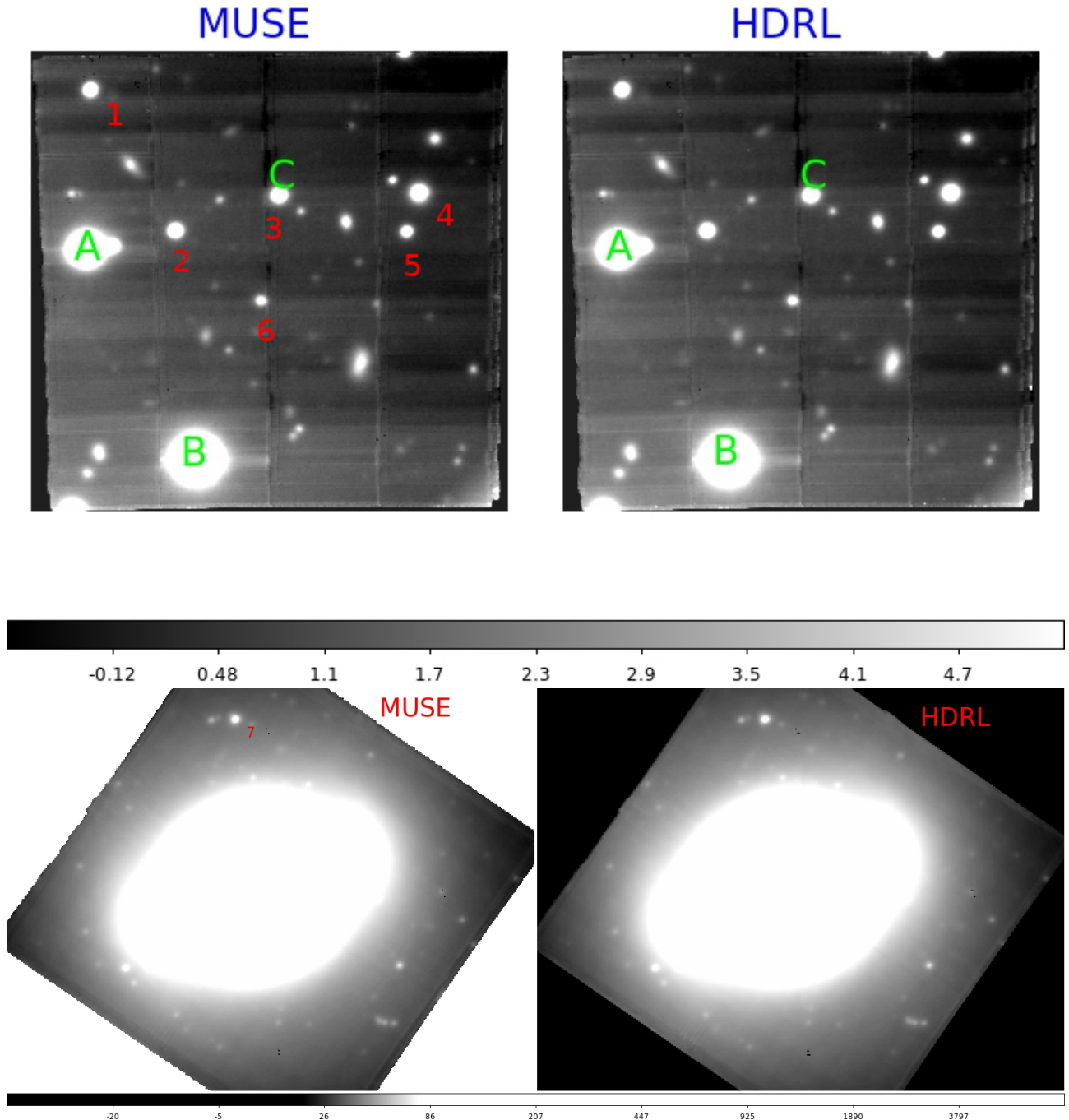


Figure J.19: Comparison between the reconstructed fields of view obtained by combining individual reduced pixel tables with the MUSE pipeline (left) and by combining individual reduced cubes with `hdrl_demo_resample` (right). The figure on the top refers to TNJ2009-304, the figure on the bottom to FCC 184. The numbers indicate the sources we measured the Moffat PSF in Table J.7.0. The letters indicate the sources we compare in the spectra of figures J.20 to J.24.

## J.7.2 Difference in spectra

In this section we quantify the differences in the spectral properties due to the different resampling scheme. The analysis is done for the 3 sources in the field of TNJ2009-304, marked as A, B, and C in Figure J.19.

The first effect of the smearing in the spatial resolution is that in the HDRL cube the flux is distributed on a larger area than in the MUSE cube. Therefore, at each spaxel the flux level is different; in the studied sources this difference can be as large as 5% (see figure J.20), but it obviously depends on the spatial profile, the difference being larger for more peaked profiles.

We then measured the changes in the integrated flux of individual sources; we compared the flux of some sources integrated on a 5x5 box around the source peak. This comparison is shown in figures J.21–J.23.

As shown in these figures, there are small difference between spectra of the same source extracted from the MUSE and HDRL datacubes, of the order of 1-2%. The S/N ratio of the spectra extracted from the cube produced by HDRL is a few percent higher than the S/N of spectra extracted from datacubes combined with the MUSE pipeline; this is expected, as the two regridding process also smooths the spectra along the wavelength direction.

Figure J.24 shows the comparison between the spectra of source A on a wavelength region around prominent absorption lines (CaT). The figure shows that, while the continuum of the two spectra is more or less the same (a small offset  $< 0.5\%$  is visible), the shape of the absorption lines differ by  $\approx 1\%$ . This means that the combination of datacubes (that implies 2 resamplings) can “damage” the shape of line features by  $\approx 1\%$  with respect to the combination of reduced pixel tables.

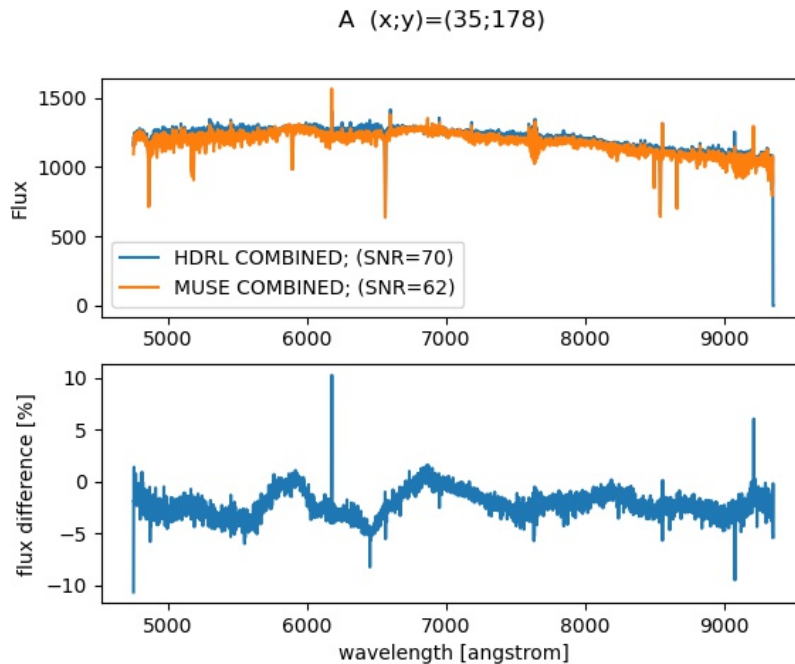


Figure J.20: Top panel. Comparison between the central spaxel of source A shown in J.19 in the cube obtained by combining individual reduced pixel tables with the MUSE pipeline (left orange) and the one obtained by combining individual reduced cubes with `hdr_demo_resample` (blue). Bottom panel, the difference in percentage between the 2 spectra of the top panel.

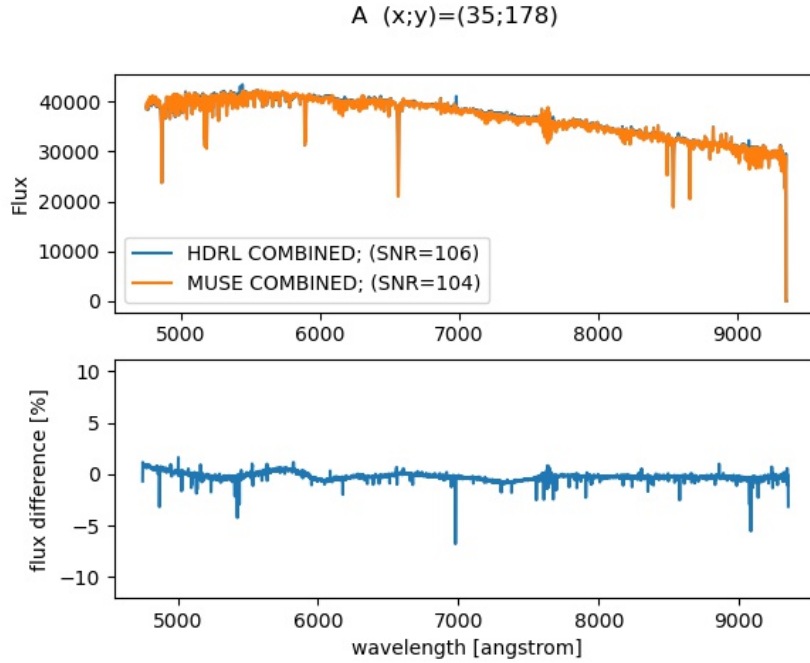


Figure J.21: Top panel. Comparison between the spectrum of source A shown in J.19 in the cube obtained by combining individual reduced pixel tables with the MUSE pipeline (left orange) and the one obtained by combining individual reduced cubes with `hdl_demo_resample` (blue). The spectrum is obtained by integrating a 7x7 box around the spaxel indicated in the image. Bottom panel, the difference in percentage between the 2 spectra of the top panel.

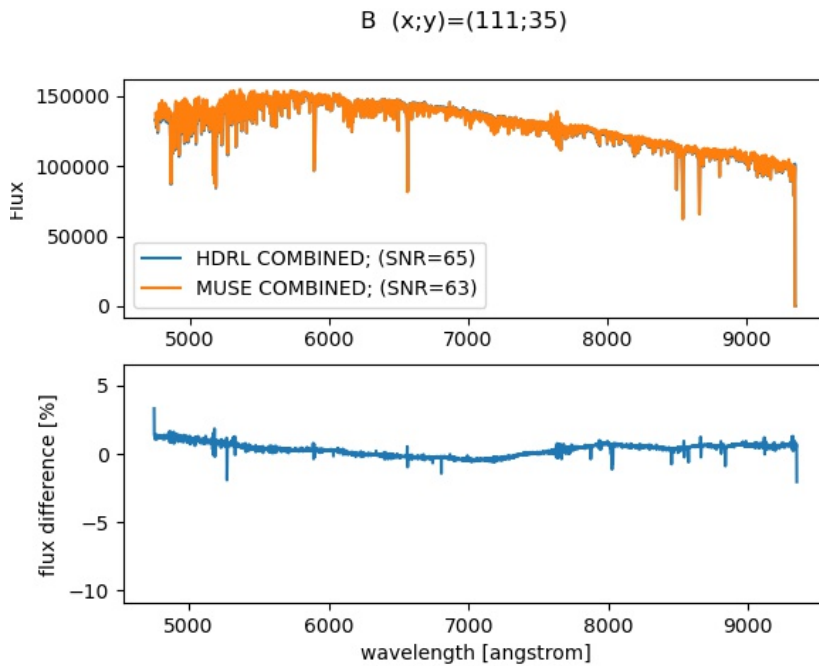


Figure J.22: As in figure J.21, but for source B in figure J.19.

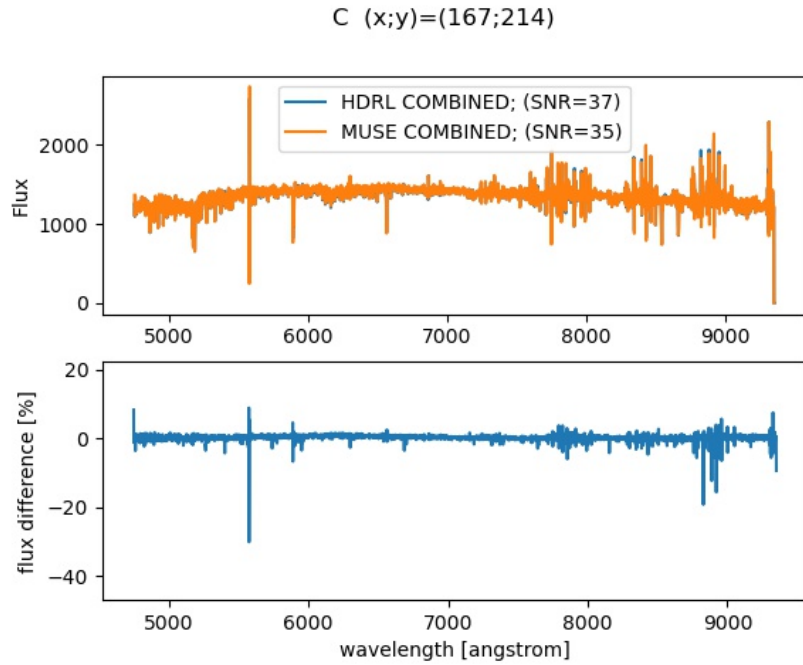


Figure J.23: As in figure J.21, but for source C in figure J.19.

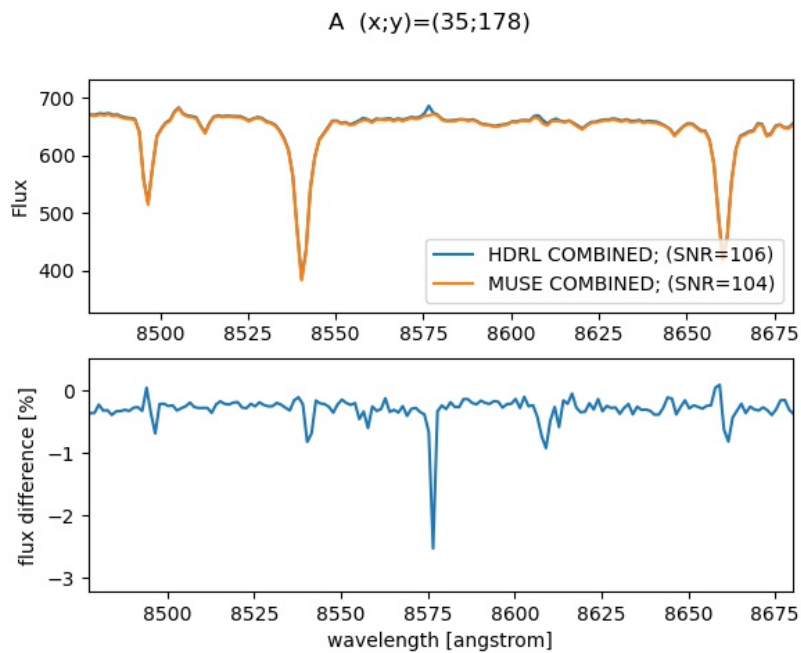


Figure J.24: As in figure J.21, but for the wavelength region around CaT absorption features.

### J.7.3 Difference in derived properties

In this section we measure the differences in physical properties derived from the analysis of the spectra extracted from the MUSE and HDRL cubes. As example, we study the two-dimensional velocity field of the ionized gas in FCC 184. The galaxy is known to have a steep spatial gradient of the rotational velocity. We aim at quantifying how much the velocity gradient is affected by the smearing effect.

The analysis is done for the cube of FCC 184.

In each spaxel we fit  $H\alpha$  (6562.82 Å) and  $[NII](6548.03, 6583.41 \text{ Å})$  emission lines; the 3 lines share the same radial velocity and velocity dispersion. In the fit, we include a broad  $H\alpha$  in absorption (with independent velocity and velocity dispersion) and a 5-th order additive polynomial to mimic the effect of the underlying stellar continuum.

Figure J.25 compares the central two-dimensional velocity fields obtained from the MUSE cube and HDRL cube. The smearing smooths the velocity field, which is less noisy in the HDRL cube. The impact of the central velocity gradient is within 1 sigma, as shown in Figure J.26. In the velocity field extracted from the MUSE cube, the velocity gradient is  $\sim 9.84 \pm 0.40 \text{ km/sec/arcsec}$  (measured along  $PA \sim 55^\circ$ ). In the velocity field extracted from the HDRL cube, the velocity gradient is a bit shallower, and it amounts to  $9.36 \pm 0.24 \text{ km/sec/arcsec}$ .

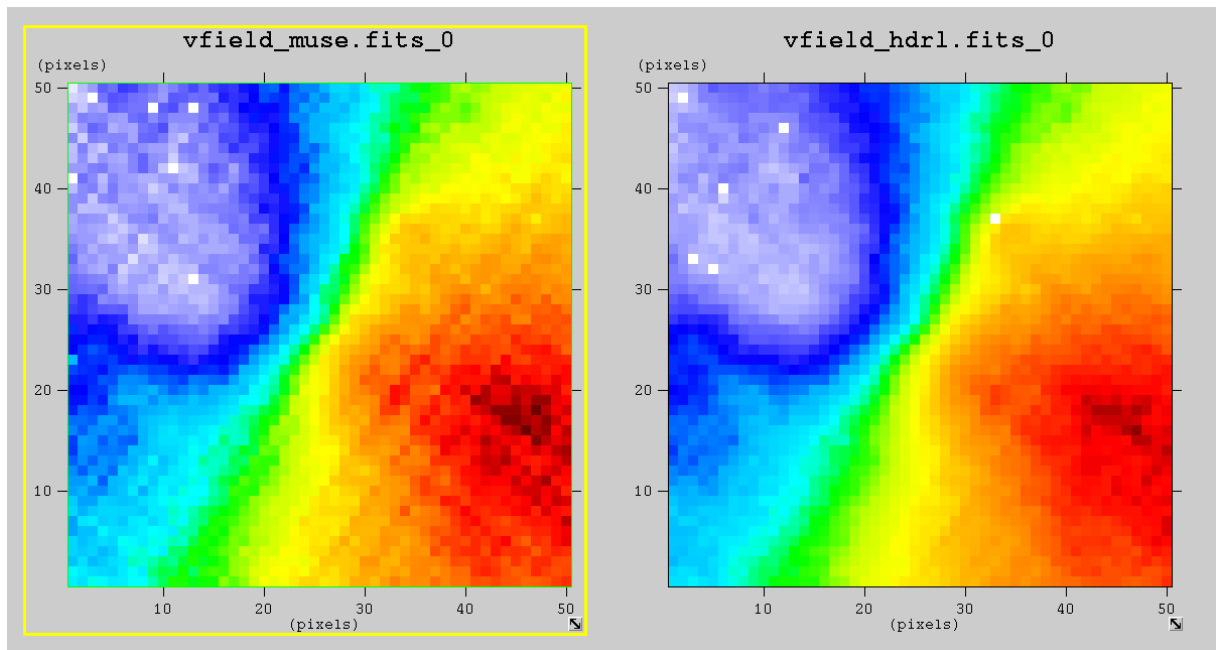


Figure J.25: Comparison between the two-dimensional velocity fields of FCC 184 as measured in the MUSE cube (left) and right cube (HDRL). The velocity range is from 1220 to 1360 km/sec

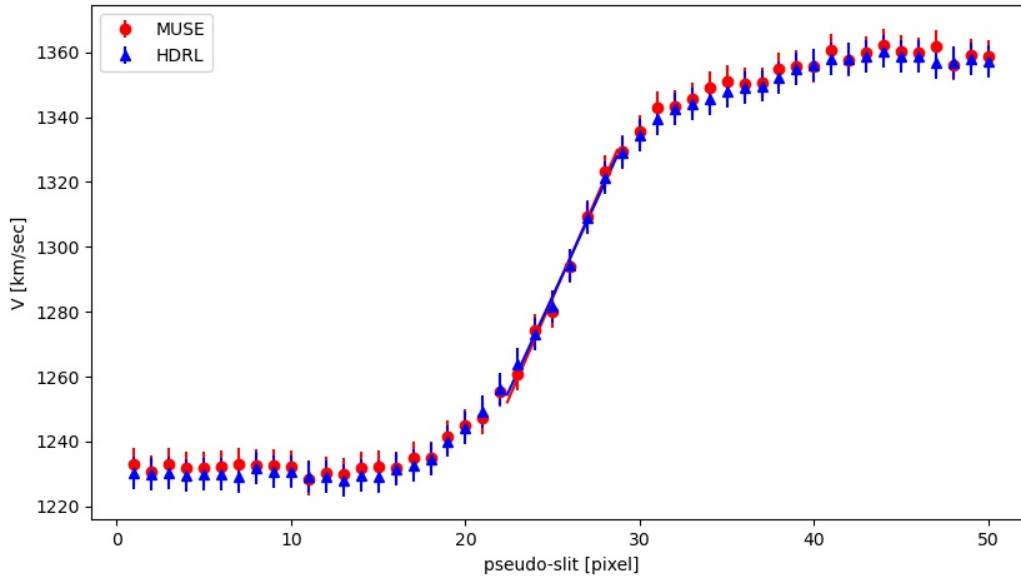


Figure J.26: Comparison between the rotation curves extracted from the velocity fields ( $PA \sim 55^\circ$ ). The central velocity gradient is measured from the central 6 pixels (1 pixel = 1.25 arcsec), and it amounts to  $12.3 \pm 0.5$  km/sec/pixel ( $\sim 9.84 \pm 0.40$  km/sec/arcsec) for the MUSE velocity field and  $11.7 \pm 0.3$  km/sec/pixel ( $9.36 \pm 0.24$  km/sec/arcsec) for the HDRL velocity field.

## J.8 Error Handling in *hdrldemo\_resample*

The routines within *hdrldemo\_resample* allow for the propagation of errors. To test whether or not the errors are propagated correctly, a number of synthetic error data sets have been created. The first set consists of a single and 10 dithered synthetic images with Gaussian-like errors. The second set consists of a single and 30 dithered synthetic images with a constant error of  $10 e^-$ .

### J.8.1 Gaussian Errors

The background of each synthetic image was created using the `iraf.mknoise` function with a background level of  $500 e^-$ . with Poisson noise, a gain of  $2.5 e^-/ADU$ , and a Gaussian readnoise of  $5.0 e^-$ . This resulted in synthetic images with a background of  $500 \pm 19.3 e^-$ . A faux error plane was then created by simply taking the square root of the data signal. This resulted in a synthetic error plane with an average level of  $11.2 \pm 0.3 e^-$ .

Based on this, we would expect the median error plane of the resampled images with a single synthetic image as input to be close to the  $11.2 \pm 0.3 e^-$  value. As can be seen in table J.8.0, this is the case for all interpolation methods except for the `LINEAR` method. Clearly, the `LINEAR` method is significantly underestimating the error.

For the resampled combination of 10 synthetic images, the error propagation should deliver an average error value of approximately  $\frac{11.2}{\sqrt{10}} = 3.5 e^-$ . It would appear from table J.8.0 that the `NEAREST` method significantly over-estimates the error. However, since the `NEAREST` method takes the nearest value of each pixel in both the area around it, and in the stack of combined frames. Thus, the `NEAREST` method does not compute a strict average of combined frames, this error value is correct.



Table J.8.0: Error Statistics of the Synthetic Data Images (Normal Errors) and the Resultant Interpolation Images

Image[ext] <sup>1</sup>	median	trim_mean	trim_std_dev
synthetic_stars_gal_bkgd_1...10.fits[2]	11.199	11.202	0.305
hdrldemo_resample_DRIZZLE_single_1.fits[3]	11.165	11.151	0.464
hdrldemo_resample_DRIZZLE_single_3.fits[3]	11.165	11.151	0.464
hdrldemo_resample_LANCZOS_single_1.fits[3]	11.134	11.121	3.278
hdrldemo_resample_LANCZOS_single_3.fits[3]	11.146	11.135	3.180
hdrldemo_resample_LINEAR_single_1.fits[3]	8.969	8.958	0.395
hdrldemo_resample_LINEAR_single_3.fits[3]	6.381	6.374	0.401
hdrldemo_resample_NEAREST_single_1.fits[3]	11.165	11.151	0.464
hdrldemo_resample_NEAREST_single_3.fits[3]	11.165	11.151	0.464
hdrldemo_resample_QUADRATIC_single_1.fits[3]	11.076	11.062	0.514
hdrldemo_resample_QUADRATIC_single_3.fits[3]	11.000	10.986	0.818
hdrldemo_resample_RENKA_single_1.fits[3]	11.143	11.129	0.488
hdrldemo_resample_RENKA_single_3.fits[3]	11.144	11.131	0.487
hdrldemo_resample_DRIZZLE_coadd_1.fits[3]	2.503	2.544	0.900
hdrldemo_resample_DRIZZLE_coadd_3.fits[3]	2.503	2.544	0.900
hdrldemo_resample_LANCZOS_coadd_1.fits[3]	2.653	2.699	0.961
hdrldemo_resample_LANCZOS_coadd_3.fits[3]	2.926	2.983	1.110
hdrldemo_resample_LINEAR_coadd_1.fits[3]	1.323	1.346	0.502
hdrldemo_resample_LINEAR_coadd_3.fits[3]	0.632	0.645	0.240
hdrldemo_resample_NEAREST_coadd_1.fits[3]	11.162	11.148	0.450
hdrldemo_resample_NEAREST_coadd_3.fits[3]	11.162	11.148	0.450
hdrldemo_resample_QUADRATIC_coadd_1.fits[3]	1.703	1.733	0.624
hdrldemo_resample_QUADRATIC_coadd_3.fits[3]	1.199	1.222	0.430
hdrldemo_resample_RENKA_coadd_1.fits[3]	2.149	2.186	0.778
hdrldemo_resample_RENKA_coadd_3.fits[3]	2.126	2.165	0.776

<sup>1</sup> The image and the [extension] of the error frame.



## J.8.2 Constant Errors

For this data set, a faux error plane was then created and set to a constant value of  $10 e^-$ . Based on this, we would expect the error plane of the resampled images with a single synthetic image as input to be close to the  $10.0 \pm 0.0 e^-$  value. As can be seen in table J.8.0, this is the case for all interpolation methods except for the LINEAR method. Clearly, the LINEAR method is again underestimating the error when a single image is being resampled.

For the resampled combination of 30 synthetic images, the error propagation should deliver an average error value of approximately  $\frac{10.0}{\sqrt{30}} = 1.83 e^-$ . As can be seen in table J.8.0, the LINEAR method slightly under-estimates the error. The same may be true of the QUADRATIC method. It would also appear that the NEAREST method significantly over-estimates the error. However, since the NEAREST method takes the nearest value of each pixel in both the area around it, and in the stack of combined frames, this value is correct.

Table J.8.0: Error Statistics of the Synthetic Data Images (Fixed Errors) and the Resultant Interpolation Images

Image[ext] <sup>1</sup>	median	trim_mean	trim_std_dev
synthetic_stars_gal_bkgd_1..30.fits[2]	10.00	10.00	0.00
hdrldemo_resample_DRIZZLE_single_1.fits[3]	10.000	10.000	0.000
hdrldemo_resample_DRIZZLE_single_3.fits[3]	10.000	10.000	0.000
hdrldemo_resample_LANCZOS_single_1.fits[3]	9.971	9.971	2.869
hdrldemo_resample_LANCZOS_single_3.fits[3]	9.979	9.979	2.795
hdrldemo_resample_LINEAR_single_1.fits[3]	8.031	8.031	0.122
hdrldemo_resample_LINEAR_single_3.fits[3]	5.713	5.713	0.271
hdrldemo_resample_NEAREST_single_1.fits[3]	10.000	10.000	0.000
hdrldemo_resample_NEAREST_single_3.fits[3]	10.000	10.000	0.000
hdrldemo_resample_QUADRATIC_single_1.fits[3]	9.920	9.920	0.205
hdrldemo_resample_QUADRATIC_single_3.fits[3]	9.852	9.852	0.605
hdrldemo_resample_RENKA_single_1.fits[3]	9.981	9.981	0.141
hdrldemo_resample_RENKA_single_3.fits[3]	9.980	9.980	0.143
hdrldemo_resample_DRIZZLE_coadd_1.fits[3]	1.291	1.320	0.533
hdrldemo_resample_DRIZZLE_coadd_3.fits[3]	1.291	1.320	0.533
hdrldemo_resample_LANCZOS_coadd_1.fits[3]	1.368	1.399	0.564
hdrldemo_resample_LANCZOS_coadd_3.fits[3]	1.509	1.547	0.619
hdrldemo_resample_LINEAR_coadd_1.fits[3]	0.683	0.698	0.281
hdrldemo_resample_LINEAR_coadd_3.fits[3]	0.326	0.334	0.137
hdrldemo_resample_NEAREST_coadd_1.fits[3]	10.000	10.000	0.000
hdrldemo_resample_NEAREST_coadd_3.fits[3]	10.000	10.000	0.000
hdrldemo_resample_QUADRATIC_coadd_1.fits[3]	0.878	0.898	0.363
hdrldemo_resample_QUADRATIC_coadd_3.fits[3]	0.618	0.634	0.252
hdrldemo_resample_RENKA_coadd_1.fits[3]	1.108	1.133	0.457
hdrldemo_resample_RENKA_coadd_3.fits[3]	1.096	1.122	0.452

<sup>1</sup> The image and the [extension] of the error frame.



## K Definition of Limiting Magnitude

The Equation 13 of Naylor 1998, MNRAS,296, 339 gives the signal to noise  $Q$  of a source of total flux  $T$  that can be reached with optimal extraction:

$$Q = \frac{T}{FWHM \cdot \Sigma_1} \sqrt{\frac{2 \ln(2)}{\pi}} \quad (35)$$

$FWHM$  is the full with half maximum of the seeing.  $\Sigma_1$  is the standard deviation of the background noise “before” convolution with the Gaussian.

$$T = \frac{Q \cdot FWHM \cdot \Sigma_1 \pi^{1/2}}{\sqrt{2 \ln(2)}} \quad (36)$$

Replacing  $\sigma = FWHM/\sqrt{4 \ln 4}$ :

$$\begin{aligned} T &= \frac{Q \sigma \sqrt{4 \ln 4} \cdot \Sigma_1 \pi^{1/2}}{\sqrt{2 \ln(2)}} \\ &= \frac{Q 2 \sigma \sqrt{2 \ln 2} \cdot \Sigma_1 \pi^{1/2}}{\sqrt{2 \ln(2)}} \\ &= Q 2 \sigma \cdot \Sigma_1 \pi^{1/2} \end{aligned} \quad (37)$$

If I convolve the background by a Gaussian function  $g$  with  $FWHM$  equal to the seeing, then I obtain another image, with its standard deviation  $\Sigma_2$ . If the original background has a Gaussian distribution with standard deviation  $\Sigma_1$ , then  $\Sigma_1$  is related to  $\Sigma_2$  by:

$$\Sigma_1 = 2 \Sigma_2 \cdot \pi^{1/2} \cdot \sigma \quad (38)$$

Hence:

$$\begin{aligned} T &= Q 2 \sigma \cdot \left( 2 \Sigma_2 \cdot \pi^{1/2} \cdot \sigma \right) \pi^{1/2} \\ &= 4 Q \sigma^2 \cdot \Sigma_2 \cdot \pi \end{aligned} \quad (39)$$

The flux  $T$  of a source that is  $Q = 5$  brighter than the noise  $N$  defines the limiting magnitude ( $Q = T/N$  is the signal to noise):

$$\begin{aligned} AB_{\text{MAGLIM}} &= -2.5 \log(5 \cdot N) + Z_{\text{PT}} \\ &= -2.5 \log(5 \cdot T \cdot N/T) + Z_{\text{PT}} \\ &= -2.5 \log(5 \cdot T/Q) + Z_{\text{PT}} \\ &= -2.5 \log(5 Q (4 \Sigma_2 \cdot \pi \sigma^2)/Q) + Z_{\text{PT}} \\ &= -2.5 \log(2 \cdot 2 \cdot 5 \Sigma_2 \cdot \pi \sigma^2) + Z_{\text{PT}} \\ &= -2.5 \log(5 \Sigma_2 \cdot 2 \pi \sigma^2) + Z_{\text{PT}} - 2.5 \log 2 \end{aligned} \quad (40)$$

Which is equivalent to (in the case of Gaussian noise):

$$\begin{aligned} AB_{\text{MAGLIM}} &= -2.5 \log \left( 5 \frac{\Sigma_1}{2 \pi^{1/2} \sigma} \cdot 2 \pi \sigma^2 \right) + Z_{\text{PT}} - 2.5 \log 2 \\ &= -2.5 \log \left( 5 \Sigma_1 \cdot \pi^{1/2} \sigma \right) + Z_{\text{PT}} - 2.5 \log 2 \end{aligned} \quad (41)$$



Note. Equation 38 is the solution of the integral.

$$\begin{aligned}\Sigma_2(r) &= \Sigma_1(r) \sqrt{\int_0^\infty \frac{2\pi r}{(2\sigma^2\pi)^2} \left( e^{-\frac{r^2}{2\sigma^2}} \right)^2 dr} \\ &= \Sigma_1(r) \sqrt{\int_0^\infty 2\pi r \left( \frac{e^{-\frac{r^2}{2\sigma^2}}}{2\sigma^2\pi} \right)^2 dr} \\ &= \Sigma_1(r) \sqrt{\int_0^\infty 2\pi r g^2(r; \sigma) dr}\end{aligned}\tag{42}$$

where  $r = \sqrt{x^2 + y^2}$  represent the radial coordinates and  $g(r; \sigma)$  is the Gaussian function used for the convolution.

**HDRL implements equation 40**, see 4.15.1. It implements image convolution; this step is indeed needed to include all non-gaussian noise and features in an image that might affect the computation of the magnitude.



# HDRL Pipeline Developer Manual

Doc. Number: ESO-299492  
Doc. Version: 7  
Sof. Version: 1.6.0  
Released on: 2026-06-02  
Page: 173 of 173

---

— End of document —