High level Data Reduction Library
HDRL

Armin Gabasch on behalf of the HDRL team

ESO Pipeline Systems Group
Goals

- Identify common state of the art data-reduction algorithms and homogenize/share them across the various pipelines.
- Propagate the pixel by pixel error in the various data reduction steps and associate an error-bar to the various results.
- Supplement the algorithms with extensive unit and regression tests in order to reduce the maintenance cost.
- Benefit from multi-core architecture by parallelizing dedicated data-reduction steps (OpenMP).
- Help ESO and the consortia to accelerate the development and validation of new pipelines.
Modules

What we provide with each release:

• The **HDRL library** module.

• A demo pipeline named **hdrldemo** where we show pipeline code examples of the hdrl features.

• A **developer manual** where all relevant hdrl information and algorithms are described. The document can be found in the hdrl subdirectory **doc/**

• Doxygen information for all public functions.

• The tagged versions of the hdrl library can be found in:
  • [http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/](http://svnhq2.hq.eso.org/p2/tags/Pipelines/common/hdrl/)

• The tagged versions of the hdrldemo pipeline can be found in:
  • [http://svnhq2.hq.eso.org/p2/tags/Pipelines(hdrldemo](http://svnhq2.hq.eso.org/p2/tags/Pipelines(hdrldemo}
Release

- The release of a stable version is triggered by a new functionality/algorithm and not bound to a fixed release-cycle.
- Whenever a new stable hdrl release is available it will be announced to the pipeline developer and can then be included in the pipeline with the svn “external” mechanism, i.e. during the svn checkout of a pipeline.
  - bug-fixes and new features can be immediately included in a new pipeline release independent of the installation environment (VERY important for a Paranal delivery).
- Pipeline developer can decide when he wants to upgrade to a new stable release.
Code Validation

- Building and executing unit tests on each commit on many platforms.
- Executing regression tests regularly, e.g. once a day.
- Testing the pipeline kit creation and installation by using our standard tools.
- Perform static code analysis:
  - cppcheck
- Perform dynamic code analysis:
  - Valgrind
  - Thread/address/undefined-behaviour sanitizer (gcc >= 4.8)
- Analysing the Coverage of the unit and regression tests.
# Code Validation

## Code Coverage

### Cobertura Coverage Report

#### Trend

![Graph showing code coverage trend](image)

#### Project Coverage summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Packages</th>
<th>Files</th>
<th>Classes</th>
<th>Lines</th>
<th>Conditionals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cobertura Coverage Report</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>88%</td>
<td>75%</td>
</tr>
</tbody>
</table>

#### Coverage Breakdown by Package

<table>
<thead>
<tr>
<th>Name</th>
<th>Packages</th>
<th>Files</th>
<th>Classes</th>
<th>Lines</th>
<th>Conditionals</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdir</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>hdir.lds</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>79%</td>
<td>72%</td>
</tr>
<tr>
<td>hdir tests</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99%</td>
<td>99%</td>
</tr>
</tbody>
</table>

### Configuration Matrix

- **SL5.2**: production, debug
- **SL5.5**: production, debug
- **SL6.3**: production, debug
- **Fedora19**: production, debug
- **Fedora20**: production, debug
- **CentOS6.4**: production, debug
- **Ubuntu10.04**: production, debug
- **Ubuntu12.04**: production, debug
- **Ubuntu14.04**: production, debug
- **Debian6.0**: production, debug
- **Debian7**: production, debug
- **OSX10.7**: production, debug
- **OSX10.8**: production, debug
- **OSX10.9**: production, debug
- **OpenSUSE12.3**: production, debug
- **OpenSUSE13.1**: production, debug
Verify the correctness of the implementation by using simulated AND real data from many different instruments covering a variety of typical use-cases.

The scientific validation is done in collaboration with scientists from the Science Data Products Group.
Library usage in a recipe

4.5.2  Bad-pixel detection on a single image

4.5.2.1  Functions

The bad pixels are computed using the following function call

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

4.5.2.2  Inputs

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

```c
hdrl_parameter * params = hdrl_bpm_2d_parameter_create_filtersmooth(
    double    kappa_low,
    double    kappa_high,
    int       maxiter,
    int       filter,
    int       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

```c
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);
```

4.5.2.3  Outputs

The result is a mask of type chpl_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.
Library usage in a recipe

But there are also a lot of cpl-like function including automatic error propagation like

<table>
<thead>
<tr>
<th>Library Function</th>
<th>Library Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdrl_image_add_image()</td>
<td>hdrl_imagelist_add_image()</td>
</tr>
<tr>
<td>hdrl_image_add_scalar()</td>
<td>hdrl_imagelist_add_imagelist()</td>
</tr>
<tr>
<td>hdrl_image_div_image()</td>
<td>hdrl_imagelist_add_scalar()</td>
</tr>
<tr>
<td>hdrl_image_div_scalar()</td>
<td>hdrl_imagelist_div_image()</td>
</tr>
<tr>
<td>hdrl_image_mul_image()</td>
<td>hdrl_imagelist_div_imagelist()</td>
</tr>
<tr>
<td>hdrl_image_mul_scalar()</td>
<td>hdrl_imagelist_div_scalar()</td>
</tr>
<tr>
<td>hdrl_image_sub_image()</td>
<td>hdrl_imagelist_sub_image()</td>
</tr>
<tr>
<td>hdrl_image_sub_scalar()</td>
<td>hdrl_imagelist_sub_imagelist()</td>
</tr>
<tr>
<td>hdrl_image_get_mean</td>
<td>hdrl_imagelist_sub_hat()</td>
</tr>
<tr>
<td>hdrl_image_get_median</td>
<td>hdrl_imagelist_sub_imagelist()</td>
</tr>
<tr>
<td>hdrl_image_get_minmax_mean</td>
<td>hdrl_imagelist_sub_hat()</td>
</tr>
<tr>
<td>hdrl_image_get_sigclip_mean</td>
<td>hdrl_imagelist_sub_hat()</td>
</tr>
<tr>
<td>hdrl_image_get_sum</td>
<td>hdrl_imagelist_sub_hat()</td>
</tr>
<tr>
<td>hdrl_image_get_stdev</td>
<td>hdrl_imagelist_sub_hat()</td>
</tr>
</tbody>
</table>

...
Implemented Algorithms

- Overscan computation and subtraction.
- Master frame combination.
- Cosmic ray detection on a single image.
- Bad pixel determination on
  - Single images.
  - Stack of identical images, e.g. bias/dark frames.
  - Sequence of images, e.g. domeflats with different exposure time.
- Computation of the Strehl ratio (mostly finished).
Overscan computation/subtraction

Computed in a running box of a user-defined size. Implemented methods: mean, median, weighted mean, min-max rejection, Kappa-Sigma clipping.
Masterframe Combination

The master-bias and master-dark frames can be created with the general collapsing functions available in hdrl. Currently the following collapsing algorithms are provided:

- Mean
- Median
- Weighted mean
- Min-max rejection
- Kappa-Sigma clipping based on interquartile range

All of those take bad pixels into account and propagate the pixel-to-pixel error.

For the master-flat a more sophisticated process is currently developed - work in progress.
Cosmic Ray Detection

Modified LA-Cosmic described in van Dokkum, 2001 based on laplacian edge detection.

Modifications:

- We use the error image passed to the function and not an error model as described in the paper.
- We do several iterations and in each iteration we replace the detected cosmic ray hits by the median of the surroundings pixels taking into account the pixel quality information.
Cosmic Ray Detection
Cosmic Ray Detection
Cosmic Ray Detection
Bad Pixel Detection

• On single images
  • Edge detection using LaCosmic
  • Smooth image – subtract smoothed image – detect outliers on residual image with different methods.

• On Stack of identical images, e.g. bias/dark
  • Collapse the stack of images – subtract the collapsed image - detect outliers on residual images with different methods.

• On a sequence of images, e.g. dome-flats with different exposure time
  • Fit a polynomial along each pixel sequence of the images and convert the information of the fit into a bad pixel map, e.g. using the chi/chi2 of the fit, comparing the fit-coefficients, ...
The most commonly used metrics for evaluating the AO correction.

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.
Strehl ratio

Calibrated PSFs + Noise = 2 x Nyquist sampling

- Teramo_4 (SR=0.0979)
- Catania_4 (SR=0.2802)
- Padova_4 (SR=0.7359)
- Firenze_4 (SR=1.0000)

Theoretical Strehl Error vs Measured Strehl Error

Peak/BKG RMS

1x10^1
1x10^4
Future Algorithms

- Optimal Extraction
- Fringe Correction
- Wavelength calibration
- Satellite trail detection
- Source detection/extraction
- ...

ESO Pipeline Workshop, Oct 24 2014