Python bindings for CPL recipes

Release 0.7.4

Ole Streicher

Nov 05, 2018

Contents

This is a non-official python module to access CPL recipes. It is not meant as part of the CPL or the MUSE pipeline software, but may be useful for testing and analysis.

See also:

http://www.eso.org/sci/software/cpl

"The Common Pipeline Library (CPL) consists of a set of C libraries, which have been developed to standardise the way VLT instrument pipelines are built, to shorten their development cycle and to ease their maintenance. The Common Pipeline Library was not designed as a general purpose image processing library, but rather to address two primary requirements. The first of these was to provide an interface to the VLT pipeline runtime- environment. The second was to provide a software kit of medium-level tools, which allows astronomical data-reduction tasks to be built rapidly." [ESO]

CHAPTER 1

Installation

1.1 Prequisites

- Python 2.6 or higher,
- Astropy

1.2 Binary packages

On Debian and debian-based systems (Ubuntu, Mint), python-cpl can be installed with the command

```
apt-get install python-cpl
```

Python CPL comes with the Ubuntu distribution since 12.04. Debian packages are in Wheezy (Debian 7), Squeeze (Debian 8), and Testing

1.3 Source code

- Python Package Index
- Git repository. To access, do a:

  ```
  git clone git://github.com/olebole/python-cpl.git
  ```

  This gives you the current version in the subdirectory `python-cpl`. To update to the current version of an existing repository, do a `git pull` in the `python-cpl` directory.

  For more detailed information, check the manual page of *`git(1)`* and the github page of the project.

1.4 Compilation

For compilation, a C compiler is needed additionally to the software mentioned above.

The installation follows the standard procedure used in python. On default, the installation path `/usr/local`. If using a non-standard installation path, add the directory *`PREFIX`*`/lib/python2.7/site-packages/` (`lib64/python2.7/site-packages/` on 64 bit systems) to your environment variable `PYTHONPATH` where where *`PREFIX`* is the installation path for the package.

In the source directory of python-cpl, run

```
python setup.py install --prefix=PREFIX
```

There are other options available as well; use the `--help` option to list them.

1.5 Test suite

There are a number of tests defined in `test/TestRecipe.py`:

```
python TestRecipe.py
```

The test recipe needs an installed CPL development environment. The tests may print a memory corruption detection by glibc. This is normal, since the tests also check this behaviour in the recipe.

Tests are also automatically buils by Travis CI.

CHAPTER 2

Tutorial

2.1 Simple example

The following code takes BIAS input file names from the command line and writes the MASTER BIAS to the file name provided with the `-o` option:

```
from optparse import OptionParser
import sys
import cpl

parser = OptionParser(usage='%prog files')
parser.add_option('-o', '--output', help='Output file', default='master_bias.fits')
parser.add_option('-b', '--badpix-table', help='Bad pixel table')

(opt, filenames) = parser.parse_args()
if not filenames:
    parser.print_help()
    sys.exit()

cpl.esorex.init()

muse_bias = cpl.Recipe('muse_bias')
muse_bias.param.nifu = 1
muse_bias.calib.BADPIX_TABLE = opt.badpix_table

res = muse_bias(filenames)
res.MASTER_BIAS.writeto(opt.output)
```

2.2 Quick guide

Input lines are indicated with ">>>" (the python prompt). The package can be imported with

```
>>> import cpl
```

If you migrate from Esorex, you may just init the search path for CPL recipes from the esorex startup:

```
>>> cpl.esorex.init()
```

Otherwise, you will need to explicitely set the recipe search path:

```
>>> cpl.Recipe.path = '/store/01/MUSE/recipes'
```

List available recipes:

```
>>> cpl.Recipe.list()
[('muse_quick_image', ['0.2.0', '0.3.0']),
 ('muse_scipost', ['0.2.0', '0.3.0']),
 ('muse_scibasic', ['0.2.0', '0.3.0']),
 ('muse_flat', ['0.2.0', '0.3.0']),
 ('muse_subtract_sky', ['0.2.0', '0.3.0']),
 ('muse_bias', ['0.2.0', '0.3.0']),
 ('muse_ronbias', ['0.2.0', '0.3.0']),
 ('muse_fluxcal', ['0.2.0', '0.3.0']),
 ('muse_focus', ['0.2.0', '0.3.0']),
 ('muse_lingain', ['0.2.0', '0.3.0']),
 ('muse_dark', ['0.2.0', '0.3.0']),
 ('muse_combine_pixtables', ['0.2.0', '0.3.0']),
 ('muse_astrometry', ['0.2.0', '0.3.0']),
 ('muse_wavecal', ['0.2.0', '0.3.0']),
 ('muse_exp_combine', ['0.2.0', '0.3.0']),
 ('muse_dar_correct', ['0.2.0', '0.3.0']),
 ('muse_standard', ['0.2.0', '0.3.0']),
 ('muse_create_sky', ['0.2.0', '0.3.0']),
 ('muse_apply_astrometry', ['0.2.0', '0.3.0']),
 ('muse_rebin', ['0.2.0', '0.3.0'])]
```

Create a recipe specified by name:

```
>>> muse_scibasic = cpl.Recipe('muse_scibasic')
```

By default, it loads the recipe with the highest version number. You may also explicitely specify the version number:

```
>>> muse_scibasic = cpl.Recipe('muse_scibasic', version = '0.2.0')
```

List all parameters:

```
>>> print muse_scibasic.param
{'ybox': 40, 'passes': 2, 'resample': False, 'xbox': 15, 'dlambda': 1.25,
 'cr': 'none', 'thres': 5.8, 'nifu': 0, 'saveimage': True}
```

Set a parameter:

```
>>> muse_scibasic.param.nifu = 1
```

Print the value of a parameter (`None` if the parameter is set to default)

```
>>> print muse_scibasic.param.nifu.value
1
```

List all calibration frames:

```
>>> print muse_scibasic.calib
{'TRACE_TABLE': None, 'MASTER_SKYFLAT': None, 'WAVECAL_TABLE': None,
 'MASTER_BIAS': None, 'MASTER_DARK': None, 'GEOMETRY_TABLE': None,
 'BADPIX_TABLE': None, 'MASTER_FLAT': None, 'GAINRON_STAT': None}
```

Set calibration frames with files:

```
>>> muse_scibasic.calib.MASTER_BIAS    = 'MASTER_BIAS-01.fits'
>>> muse_scibasic.calib.MASTER_FLAT    = 'MASTER_FLAT-01.fits'
>>> muse_scibasic.calib.TRACE_TABLE    = 'TRACE_TABLE-01.fits'
>>> muse_scibasic.calib.GEOMETRY_TABLE = 'geometry_table.fits'
```

You may also set calibration frames with `astropy.io.fits.HDUList` objects. This is especially useful if you want to change the file on the fly:

```
>>> import astropy.io.fits
>>> wavecal = astropy.io.fits.open('WAVECAL_TABLE-01_flat.fits')
>>> wavecal[1].data.field('wlcc00')[:] *= 1.01
>>> muse_scibasic.calib.WAVECAL_TABLE = wavecal
```

To set more than one file for a tag, put the file names and/or `astropy.io.fits.HDUList` objects into a list:

```
>>> muse_scibasic.calib.MASTER_BIAS    = [ 'MASTER_BIAS-%02i.fits' % (i+1)
...                                         for i in range(24) ]
```

To run the recipe, call it with the input file names as arguments. The product frames are returned in the return value of the call. If you don't specify an input frame tag, the default (first) one of the recipe is used.

```
>>> res = muse_scibasic('Scene_fusion_1.fits')
```

Run the recipe with a nondefault tag (use raw data tag as argument name):

```
>>> res = muse_scibasic(raw = {'SKY':'sky_newmoon_no_noise_1.fits'})
```

Parameters and calibration frames may be changed for a specific call by specifying them as arguments:

```
>>> res =  muse_scibasic('Scene_fusion_1.fits', param = {'nifu': 2},
...                      calib = {'MASTER_FLAT': None,
...                               'WAVECAL_TABLE': 'WAVECAL_TABLE_noflat.fits'})
```

The results of a calibration run are `astropy.io.fits.HDUList` objects. To save them (use output tags as attributes):

```
>>> res.PIXTABLE_OBJECT.writeto('Scene_fusion_pixtable.fits')
```

They can also be used directly as input of other recipes.

```
>>> muse_sky = cpl.Recipe('muse_sky')
...
>>> res_sky = muse_sky(res.PIXTABLE_OBJECT)
```

If not saved, the output is usually lost! During recipe run, a temporary directory is created where the `astropy.io.fits.HDUList` input objects and the output files are put into. This directory is cleaned up afterwards.

To control message verbosity on terminal (use `'debug'`, `'info'`, `'warn'`, `'error'` or `'off'`):

```
>>> cpl.msg.esorex.level = 'debug'
```

CHAPTER 3

The Recipe interface

class cpl.**Recipe**(*name*, *filename=None*, *version=None*, *threaded=False*)
Pluggable Data Reduction Module (PDRM) from a ESO pipeline.

Recipes are loaded from shared libraries that are provided with the pipeline library of the instrument. The module does not need to be linked to the same library version as the one used for the compilation of python-cpl. Currently, recipes compiled with CPL versions from 4.0 are supported. The list of supported versions is stored as `cpl.cpl_versions`.

The libraries are searched in the directories specified by the class attribute *Recipe.path* or its subdirectories. The search path is automatically set to the esorex path when *cpl.esorex.init()* is called.

3.1 Static members

Recipe.**path = ['.']**
Search path for the recipes. It may be set to either a string, or to a list of strings. All shared libraries in the search path and their subdirectories are searched for CPL recipes. On default, the path is set to the current directory.

The search path is automatically set to the esorex path when *cpl.esorex.init()* is called.

Recipe.**memory_mode = 0**
CPL memory management mode. The valid values are

0 Use the default system functions for memory handling

1 Exit if a memory-allocation fails, provide checking for memory leaks, limited reporting of memory allocation and limited protection on deallocation of invalid pointers.

2 Exit if a memory-allocation fails, provide checking for memory leaks, extended reporting of memory allocation and protection on deallocation of invalid pointers.

Note: This variable is only effective before the CPL library was initialized. Even *cpl.Recipe.list()* initializes the library. Therefore it is highly recommended to set this as the first action after importing *cpl*.

---

static Recipe.**list**()
Return a list of recipes.

Searches for all recipes in in the directory specified by the class attribute *Recipe.path* or its subdirectories.

static Recipe.**set_maxthreads**(*n*)
Set the maximal number of threads to be executed in parallel.

Note: This affects only threads that are started afterwards with the `threaded = True` flag.

See also:

Parallel execution

3.2 Constructor

Recipe.**__init__**(*name*, *filename=None*, *version=None*, *threaded=False*)
Try to load a recipe with the specified name in the directory specified by the class attribute *Recipe.path* or its subdirectories.

Parameters

- **name** (`str`) – Name of the recipe. Required. Use *cpl.Recipe.list()* to get a list of available recipes.
- **filename** (`str`) – Name of the shared library. Optional. If not set, *Recipe.path* is searched for the library file.
- **version** (`int` or `str`) – Version number. Optional. If not set, the newest version is loaded.
- **threaded** (`bool`) – Run the recipe in the background, returning immediately after calling it. Default is `False`. This may be also set as an attribute or specified as a parameter when calling the recipe.

3.3 Common attributes and methods

These attributes and methods are available for all recipes.

Recipe.**__name__**
Recipe name.

Recipe.**__file__ = None**
Shared library file name.

Recipe.**__author__**
Author name

Recipe.**__email__**
Author email

Recipe.**__copyright__**
Copyright string of the recipe

Recipe.**description**
Pair (synopsis, description) of two strings.

Recipe.**version**
Pair (versionnumber, versionstring) of an integer and a string. The integer will be increased on development progress.

Recipe.**cpl_version**
Version of the CPL library that is linked to the recipe, as a string

Recipe.**cpl_description**
Version numbers of CPL and its libraries that were linked to the recipe, as a string.

Recipe.**output_dir**
Output directory if specified, or None. The recipe will write the output files into this directory and return their file names. If the directory does not exist, it will be created before the recipe is executed. Output files within the output directory will be silently overwritten. If no output directory is set, the recipe call will result in astropy.io.fits.HDUList result objects. The output directory may be also set as parameter in the recipe call.

Recipe.**temp_dir**
Base directory for temporary directories where the recipe is executed. The working dir is created as a subdir with a random file name. If set to None, the system temp dir is used. Defaults to '.'.

Recipe.**threaded**
Specify whether the recipe should be executed synchroniously or as an extra process in the background.

See also:

Parallel execution

Recipe.**tag**
Default tag when the recipe is called. This is set automatically only if the recipe provided the information about input tags. Otherwise this tag has to be set manually.

Recipe.**tags**
Possible tags for the raw input frames, or 'None if this information is not provided by the recipe.

Recipe.**output**
Return a dictionary of output frame tags.

Keys are the tag names, values are the corresponding list of output tags. If the recipe does not provide this information, an exception is raised.

Recipe.**memory_dump**
If set to 1, a memory dump is issued to stdout if the memory was not totally freed after the execution. If set to 2, the dump is always issued. Standard is 0: nothing dumped.

3.4 Recipe parameters

Recipe parameters may be set either via the *Recipe.param* attribute or as named keywords on the run execution. A value set in the recipe call will overwrite any value that was set previously in the *Recipe.param* attribute for that specific call.

Recipe.**param**
This attribute contains all recipe parameters. It is iteratable and then returns all individual parameters:

```
>>> for p in muse_scibasic.param:
...     print p.name, p.value, p.default
...
nifu None 99
cr None dcr
```

(continues on next page)

(continued from previous page)

```
xbox None 15
ybox None 40
passes None 2
thres None 4.5
sample None False
dlambda None 1.2
```

On interactive sessions, all parameter settings can be easily printed by printing the *param* attribute of the recipe:

```
>>> print muse_scibasic.param
 [Parameter('nifu', default=99), Parameter('cr', default=dcr),
  Parameter('xbox', default=15), Parameter('ybox', default=40),
  Parameter('passes', default=2), Parameter('thres', default=4.5),
  Parameter('sample', default=False), Parameter('dlambda', default=1.2)]
```

To set the value of a recipe parameter, the value can be assigned to the according attribute:

```
>>> muse_scibasic.param.nifu = 1
```

The new value is checked against parameter type, and possible value limitations provided by the recipe. Hyphens in parameter names are converted to underscores. In a recipe call, the same parameter can be specified as `dict`:

```
>>> res = muse_scibasic( ..., param = {'nifu':1})
```

To reset a value to its default, it is either deleted, or set to `None`. The following two lines:

```
>>> muse_scibasic.param.nifu = None
>>> del muse_scibasic.param.nifu
```

will both reset the parameter to its default value.

All parameters can be set in one step by assigning a `dict` to the parameters. In this case, all values that are not in the map are reset to default, and unknown parameter names are ignored. The keys of the map may contain contain the name or the fullname with context:

```
>>> muse_scibasic.param = { 'nifu':1, 'xbox':11, 'resample':True }
```

See also:

cpl.Parameter

3.5 Recipe frames

There are three groups of frames: calibration ("calib") frames, input ("raw") frames, and result ("product") frames. Calibration frames may be set either via the *Recipe.calib* attribute or as named keywords on the run execution. A value set in the recipe call will overwrite any value that was set previously in the *Recipe.calib* attribute for that specific call. Input frames are always set in the recipe call. If their tag name was not given, the tag name from *Recipe.tag* is used if the recipe provides it.

`Recipe.`**`calib`**

This attribute contains the calibration frames for the recipe. It is iterable and then returns all calibration frames:

```
>>> for f in muse_scibasic.calib:
...     print f.tag, f.min, f.max, f.frames
```

(continues on next page)

(continued from previous page)

```
TRACE_TABLE 1 1 None
WAVECAL_TABLE 1 1 None
MASTER_BIAS 1 1 master_bias_0.fits
MASTER_DARK None 1 None
GEOMETRY_TABLE 1 1 None
BADPIX_TABLE None None ['badpix_1.fits', 'badpix_2.fits']
MASTER_FLAT None 1 None
```

Note: Only MUSE recipes are able to provide the full list of calibration frames and the minimal/maximal number of calibration frames. For other recipes, only frames that were set by the users are returned here. Their minimum and maximum value will be set to `None`.

In order to assing a FITS file to a tag, the file name or the `astropy.io.fits.HDUList` is assigned to the calibration attribute:

```
>>> muse_scibasic.calib.MASTER_BIAS = 'MASTER_BIAS_0.fits'
```

Using `astropy.io.fits.HDUList` is useful when it needs to be patched before fed into the recipe.

```
>>> master_bias = astropy.io.fits.open('MASTER_BIAS_0.fits')
>>> master_bias[0].header['HIERARCH ESO DET CHIP1 OUT1 GAIN'] = 2.5
>>> muse_scibasic.calib.MASTER_BIAS = master_bias
```

Note that `astropy.io.fits.HDUList` objects are stored in temporary files before the recipe is called which may produce some overhead. Also, the CPL then assigns the random temporary file names to the FITS keywords `HIERARCH ESO PRO RECm RAWn NAME` which should be corrected afterwards if needed.

To assign more than one frame, put them into a list:

```
>>> muse_scibasic.calib.BADPIX_TABLE = [ 'badpix1.fits', 'badpix2.fits' ]
```

All calibration frames can be set in one step by assigning a `dict` to the parameters. In this case, frame that are not in the map are set are removed from the list, and unknown frame tags are silently ignored. The key of the map is the tag name; the values are either a string, or a list of strings, containing the file name(s) or the `astropy.io.fits.HDUList` objects.

```
>>> muse_scibasic.calib = { 'MASTER_BIAS':'master_bias_0.fits',
...                 'BADPIX_TABLE':[ 'badpix_1.fits', 'badpix_2.fits' ] }
```

In a recipe call, the calibration frame lists may be overwritten by specifying them in a `dict`:

```
>>> res = muse_scibasic( ..., calib = {'MASTER_BIAS':'master_bias_1.fits'})
```

See also:

cpl.FrameConfig

3.6 Runtime environment

For debugging purposes, the runtime environment of the recipe may be changed. The change may be either done by specifying the *Recipe.env* attribute of as a parameter on the recipe invocation. The change will have no influence on the environment of the framework itself.

Note: Some variables are only read on startup (like MALLOC_CHECK_), changing or deleting them will have no effect.

Recipe.**env = None**
Environment changes for the recipe. This is a dict with the name of the environment variable as the key and the content as the value. It is possible to overwrite a specific environment variable. Specifying None as value will remove the variable:

```
>>> muse_flat.env['MUSE_RESAMPLE_LAMBDA_LOG'] = '1'
>>> muse_flat.env['MUSE_TIMA_FILENAME'] = 'tima.fits'
```

In a recipe call, the runtime environment may be overwritten as well:

```
>>> res = muse_flat( ..., env = {'MUSE_PLOT_TRACE':'true'})
```

3.7 Recipe invocation

Recipe.**__call__**(**data*, ***ndata*)
Call the recipes execution with a certain input frame.

Parameters

- **raw** (astropy.io.fits.HDUlist or str or a *list* of them, or dict) – Data input frames.
- **tag** (str) – Overwrite the *tag* attribute (optional).
- **threaded** (bool) – overwrite the *threaded* attribute (optional).
- **loglevel** (int) – set the log level for python logging (optional).
- **logname** (str) – set the log name for the python logging.Logger (optional, default is 'cpl.' + recipename).
- **output_dir** (str) – Set or overwrite the *output_dir* attribute. (optional)
- **param** (dict) – overwrite the CPL parameters of the recipe specified as keys with their dictionary values (optional).
- **calib** (dict) – Overwrite the calibration frame lists for the tags specified as keys with their dictionary values (optional).
- **env** (dict) – overwrite environment variables for the recipe call (optional).

Returns The object with the return frames as astropy.io.fits.HDUList objects

Return type *cpl.Result*

Raise exceptions.ValueError If the invocation parameters are incorrect.

Raise exceptions.IOError If the temporary directory could not be built, the recipe could not start or the files could not be read/written.

Raise *cpl.CplError* If the recipe returns an error.

Raise *cpl.RecipeCrash* If the CPL recipe crashes with a SIGSEV or a SIGBUS

Note: If the recipe is executed in the background (`threaded = True`) and an exception occurs, this exception is raised whenever result fields are accessed.

See also:

Parallel execution

CHAPTER 4

Parallel execution

The library allows a simple parallelization of recipe processing. The parallelization is done using independent processes and thus does not depend on parallelization features in the CPL or the recipe implementation.

To specify that a recipe should be executed in the background, the `threaded` attribute needs to be set to `True`. This may be done either in the recipe constructor, as a recipe attribute or as a parameter of the execution call. Each of the following three recipes will start a background process for the BIAS calculation:

```
# Create a threaded recipe
r1 = cpl.Recipe('muse_bias', threaded = True)
result1 = r1([ 'bias1.fits', 'bias2.fits', 'bias3.fits'])

# Prepare a recipe for background execution
r2 = cpl.Recipe('muse_bias')
r2.threaded = True
result2 = r2([ 'bias1.fits', 'bias2.fits', 'bias3.fits'])

# Execute a recipe in background
r3 = cpl.Recipe('muse_bias')
result3 = r3([ 'bias1.fits', 'bias2.fits', 'bias3.fits'], threaded = True)
```

If the `threaded` attribute is set to `True`, the execution call of the recipe immediately returns while the recipe is executed in the background. The current thread is stopped only if any of the results of the recipe is accessed and the recipe is still not finished.

The result frame of a background recipe is a subclass of `threading.Thread`. This interface may be used to control the thread execution.

The simples way to use parallel processing is to create a list where the members are created by the execution of the recipe. The following example shows the parallel execution of the 'muse_focus' recipe:

```
muse_focus = cpl.Recipe('muse_focus', threaded = True)
muse_focus.calib.MASTER_BIAS = 'master_bias.fits'

# Create a list of input files
files = [ 'MUSE_CUNGC%02i.fits' % i for i in range(20, 30) ]
```

(continues on next page)

(continued from previous page)

```
# Create a list of recipe results. Note that for each entry, a background
# process is started.
results = [ muse_focus(f) for f in files ]

# Save the results. The current thread is stopped until the according
# recipe is finished.
for i, res in enumerate(results):
    res.FOCUS_TABLE.writeto('FOCUS_TABLE_%02i.fits' % (i+1))
```

When using parallel processing note that the number of parallel processes is not limited by default, so this feature may produce a high load when called with a large number of processes. Parallelization in the recipe itself or in the CPL may also result in additional load.

To limit the maximal number of parallel processes, the function *cpl.Recipe.set_maxthreads()* can be called with the maximal number of parallel processes. Note that this function controls only the threads that are started afterwards.

If the recipe execution fails, the according exception will be raised whenever one of the results is accessed.

Note: Recipes may contain an internal parallelization using the openMP interface. Although it is recommended to leave them untouched, they may be changed via environment variable settungs in the *cpl.Recipe.env* attribute. See http://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html for a list of environment variables.

CHAPTER 5

The `cpl.Parameter` class

class `cpl.`**Parameter**(*name*)

Runtime configuration parameter of a recipe. Parameters are designed to handle monitor/control data and they provide a standard way to pass information to the recipe.

The CPL implementation supports three classes of parameters: a plain value, a value within a given range, or a value as part of an enumeration. When a parameter is created it is created for a particular value type. In the latter two cases, validation is performed whenever the value is set.

Attributes:

value
: The value of the parameter, or `None` if set to default

default
: The default value of the parameter (readonly).

name
: The parameter name (readonly). Parameter names are unique. They define the identity of a given parameter.

context
: The parameter context (readonly). The context usually consists of the instrument name and the recipe name, separated by a dot. The context is used to associate parameters together.

range
: The numeric range of a parameter, or `None` if the parameter range is unlimited (readonly).

sequence
: A `list` of possible values for the parameter if the parameter are limited to an enumeration of possible values (readonly).

The following example prints the attributes of one parameter:

```
>>> print 'name:    ', muse_scibasic.param.cr.name
name:     cr
>>> print 'fullname:', muse_scibasic.param.cr.fullname
```

(continues on next page)

(continued from previous page)

```
fullname: muse.muse_scibasic.cr
>>> print 'context: ', muse_scibasic.param.cr.context
context:  muse.muse_scibasic
>>> print 'sequence:', muse_scibasic.param.cr.sequence
sequence: ['dcr', 'none']
>>> print 'range:   ', muse_scibasic.param.cr.range
range:    None
>>> print 'default: ', muse_scibasic.param.cr.default
default:  dcr
>>> print 'value:   ', muse_scibasic.param.cr.value
value:    None
```

See also:

Recipe.param

CHAPTER 6

The `cpl.FrameConfig` class

class `cpl.`**FrameConfig**(*tag*, *min_frames=0*, *max_frames=0*, *frames=None*)

Frame configuration.

Each *`FrameConfig`* object stores information about one the data type a recipe can process. They are used for defining the calibration files. However, since this information is not generally provided by CPL recipes, it contains only dummy information, except for the MUSE recipes.

The objects stores a frame tag, a unique identifier for a certain kind of frame, the minimum and maximum number of frames needed.

Attributes:

tag
: Category tag name. The tag name is used to distinguish between different types of files. An examples of tag names is 'MASTER_BIAS' which specifies the master bias calibration file(s).

min
: Minimal number of frames, or `None` if not specified. A frame is required if the *`min`* is set to a value greater than 0.

max
: Maximal number of frames, or `None` if not specified

frames
: List of frames (file names or `astropy.io.fits.HDUList` objects) that are assigned to this frame type.

See also:

`Recipe.calib`

CHAPTER 7

Execution results

7.1 Result frames

class cpl.**Result**

Calling *cpl.Recipe.__call__()* returns an object that contains all result ('production') frames in attributes. All results for one tag are summarized in one attribute of the same name. So, the `muse_bias` recipe returns a frame with the tag MASTER_BIAS in the according attribute:

```
res = muse_bias(...)
res.MASTER_BIAS.writeto('master_bias')
```

The attribute content is either a `astropy.io.fits.HDUList` or a `list()` of HDU lists, depending on the recipe and the call: If the recipe produces one out put frame of a tag per input file, the attribute contains a list if the recipe was called with a list, and if the recipe was called with a single input frame, the result attribute will also contain a single input frame. If the recipe combines all input frames to one output frame, a single `astropy.io.fits.HDUList` es returned, independent of the input parameters. The following examples will illustrate this:

```
muse_scibasic = cpl.Recipe('muse_scibasic')
...
# Only single input frame, so we get one output frame
res = muse_scibasic('raw.fits')
res.PIXTABLE_OBJ.writeto('pixtable.fits')

# List of input frames results in a list of output frames
res = muse_scibasic([ 'raw1.fits', 'raw2.fits', 'raw3.fits' ])
for i, h in res.PIXTABLE_OBJ:
    h.writeto('pixtable%i.fits' % (i+1))

# If we call the recipe with a list containing a single frame, we get a list
# with a single frame back
res = muse_scibasic([ 'raw1.fits' ])
res.PIXTABLE_OBJ[0].writeto('pixtable1.fits')
```

(continues on next page)

(continued from previous page)

```
# The bias recipe always returns one MASTER BIAS, regardless of number of
# input frames. So we always get a single frame back.
muse_bias = cpl.Recipe('muse_bias')
...
res = muse_bias([ 'bias1.fits', 'bias2.fits', 'bias3.fits' ])
res.MASTER_BIAS.writeto('master_bias.fits')
```

Note: This works well only for MUSE recipes. Other recipes dont provide the necessary information about the recipe.

7.2 Run statistics

In Addition to the result frames the *cpl.Result* object provides the attribute `cpl.Result.stat` which contains several statistics of the recipe execution:

`cpl.Result.`**`return_code`**
: The return code of the recipe. Since an exception is thrown if the return code indicates an error, this attribute is always set to 0.

`cpl.Result.stat.`**`user_time`**
: CPU time in user mode, in seconds.

`cpl.Result.stat.`**`sys_time`**
: CPU time in system mode, in seconds.

`cpl.Result.stat.`**`memory_is_empty`**
: Flag whether the recipe terminated with freeing all available Memory. This information is only available if the CPL internal memory allocation functions are used. If this information is not available, this flag ist set to `None`.

See also:

Recipe.memory_mode

7.3 Execution log

`cpl.Result.`**`log`**
: List of log messages for the recipe.

See also:

cpl.logger.LogList

`cpl.Result.`**`error`**
: If one or more error was set during the recipe run, the first error is stored in this attribute. The following errors are chained and can be accessed with the `cpl.CplError.next` attribute.

Note: An error here does not indicate a failed recipe execution, since a failed execution would result in a non-zero return code, and an exception would be thrown.

See also:

cpl.CplError

7.4 Thread control

If the recipe was called in the background (see *Parallel execution*), the result object is returned immediately and is dervived from `threading.Thread`. Its interface can be used to control the thread execution:

`cpl.Result.`**`isAlive`**`()`
Returns whether the recipe is still running

`cpl.Result.`**`join`**`(`*timeout* = *None*`)`
Wait until the recipe terminates. This blocks the calling thread until the recipe terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As *join()* always returns `None`, you must call *isAlive()* after *join()* to decide whether a timeout happened – if the recipe is still running, the *join()* call timed out.

When the timeout argument is not present or `None`, the operation will block until the recipe terminates.

A thread can be *cpl.Result.join()* ed many times.

Like in the foreground execution, the output frames may be retrieved as attributes of the *cpl.Result* frame. If any of the attributes is accessed, the calling thread will block until the recipe is terminated. If the recipe execution raised an exception, this exception will be raised whenever an attribute is accessed.

7.5 CPL Exceptions

`exception` `cpl.`**`CplError`**`(`*retval*, *res*, *logger=None*`)`
Error message from the recipe.

If the CPL recipe invocation returns an error, it is converted into a *cpl.CplError* exception and no frames are returned. Also, the error is notified in the log file.

The exception is raised on recipe invocation, or when accessing the result frames if the recipe was started in background (*cpl.Recipe.threaded* set to `True`).

Attributes:

`code`
The CPL error code returned from the recipe.

`msg`
The supplied error message.

`filename`
The source file name where the error occurred.

`line`
The line number where the error occurred.

`log`
Log lines of the recipe that lead to this exception.

See also:

cpl.logger.LogList

next_error
Next error, or None.

exception cpl.**RecipeCrash**(*bt_file*)
Recipe crash exception

If the CPL recipe crashes with a SIGSEV or a SIGBUS, the C stack trace is tried to conserved in this exception. The stack trace is obtained with the GNU debugger gdb. If the debugger is not available, or if the debugger cannot be attached to the crashed recipe, the Exception remains empty.

When converted to a string, the Exception will return a stack trace similar to the Python stack trace.

The exception is raised on recipe invocation, or when accessing the result frames if the recipe was started in background (*cpl.Recipe.threaded* set to True).

Attributes:

elements
List of stack elements, with the most recent element (the one that caused the crash) at the end. Each stack element is a collections.namedtuple() with the following attributes:

filename
Source file name, including full path, if available.

line
Line number, if available

func
Function name, if available

params
Dictionary parameters the function was called with. The key here is the parameter name, the value is a string describing the value set.

localvars
Dictionary of local variables of the function, if available. The key here is the parameter name, the value is a string describing the value set.

signal
Signal that caused the crash.

CHAPTER 8

Log messages

We provide CPL log messages in two different ways: via Python logging and as a list of messages in the `cpl.Result` object.

For convienience, simple terminal messages and predefined log file output in a style similar to the original CPL messages.

8.1 Python style logging

The preferred and most flexible way to do logging is the use of the `logging` module of Python. A basic setup (similar to the style used in esorex) is:

```
import logging

log = logging.getLogger()
log.setLevel(logging.INFO)
ch = logging.FileHandler('cpl_recipe.log')
ch.setLevel(logging.INFO)
fr = logging.Formatter('%(created)s [%(levelname)s] %(name)s: %(message)s',
                       '%H:%M:%S')
ch.setFormatter(fr)
log.addHandler(ch)
```

The default basic log name for CPL log messages in the recipes is `cpl.`*recipename*. The log name can be changed with the `logname` parameter of *cpl.Recipe.__call__()* to follow own naming rules, or to separate the output of recipes that are executed in parallel:

```
res = [ muse_focus(f, logname = 'cpl.muse_focus%02i' % (i+1), threading = True)
        for i, f in enumerate(inputfiles) ]
```

To the basic log name the function name is appended to allow selective logging of a certain function. The following sample line:

```
logging.getLogger('cpl.muse_sky.muse_sky_create_skymask').setLevel(logging.DEBUG)
```

will log the debug messages from `muse_sky_create_skymask()` additionally to the other messages.

Note: Since the log messages are cached in CPL, they may occur with some delay in the python log module. Also, log messages from different recipes running in parallel may be mixed in their chronological order. The resolution of the log time stamp is one second. The fields `logging.LogRecord.args`, `logging.LogRecord.exc_info` and `logging.LogRecord.lineno` are not set. Also, due to limitations in the CPL logging module, level filtering is done only after the creation of the log entries. This may cause performance problems if extensive debug logging is done and filtered out by `logging.Logger.setLevel()`. In this case the *cpl.Recipe.__call__()* parameter `loglevel` may be used.

See also:

cpl.esorex.msg and *cpl.esorex.log*

EsoRex like convienience logging.

8.2 Log message lists

The `cpl.Result` object as well as a *cpl.CplError* have an attribute `cpl.Result.log` resp. *cpl.CplError.log* that contains the `list` of all log messages.

class cpl.logger.**LogList**

List of log messages.

Accessing this `list` directly will return the `logging.LogRecord` instances.

Example:

```
res = muse_bias(bias_frames)
for logrecord in res.log:
    print '%s: %s' % (entry.funcname, entry.msg)
```

To get them formatted as string, use the *error*, *warning*, *info* or *debug* attributes:

```
res = muse_bias(bias_frames)
for line in res.log.info:
    print line
```

error

Error messages as list of `str`

warning

Warnings and error messages as list of `str`

info

Info, warning and error messages as list of `str`

debug

Debug, info, warning, and error messages as list of `str`

CHAPTER 9

cpl.esorex **EsoRex** legacy support

EsoRex is a standard execution environment for CPL recipes provided by ESO.

9.1 Support for configuration and SOF files

cpl.esorex.**init**(*source=None*)
Set up the logging and the recipe search path from the esorex.rc file.

Parameters **source** (str or file) – Configuration file object, or string with file content. If not set, the esorex config file ~/.esorex/esorex.rc is used.

cpl.esorex.**load_rc**(*source=None*)
Read an **EsoRex** configuration file.

Parameters **source** (str or file) – Configuration file object, or string with file content. If not set, the **EsoRex** config file ~/.esorex/esorex.rc is used.

These files contain configuration parameters for **EsoRex** or recipes. The content of the file is returned as a map with the (full) parameter name as key and its setting as string value.

The result of this function may directly set as *cpl.Recipe.param* attribute:

```
import cpl
myrecipe = cpl.Recipe('muse_bias')
myrecipe.param = cpl.esorex.load_rc('muse_bias.rc')
```

cpl.esorex.**load_sof**(*source*)
Read an **EsoRex** SOF file.

Parameters **source** (str or file) – SOF ("Set Of Files") file object or string with SOF file content.

These files contain the raw and calibration files for a recipe. The content of the file is returned as a map with the tag as key and the list of file names as value.

The result of this function may directly set as *cpl.Recipe.calib* attribute:

```
import cpl
myrecipe = cpl.Recipe('muse_bias')
myrecipe.calib = cpl.esorex.read_sof(open('muse_bias.sof'))
```

Note: The raw data frame is silently ignored wenn setting *cpl.Recipe.calib* for MUSE recipes. Other recipes ignore the raw data frame only if it was set manually as *cpl.Recipe.tag* or in *cpl.Recipe.tags* since there is no way to automatically distinguish between them.

9.2 Convienence logging control

cpl.esorex.**msg = <cpl.esorex.CplLogger object>**

This variable is a CplLogger instance that provides a convienience stream handler similar to the terminal logging functionality of the CPL. It basically does the same as:

```
import logging

log = logging.getLogger()
log.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.OFF)
ch.setFormatter(logging.Formatter('[%(levelname)7s] %(message)s'))
log.addHandler(ch)
```

The following attributes control the format of the terminal messages:

CplLogger.**level**

Log level for output to the terminal. Any of [DEBUG, INFO, WARN, ERROR, OFF].

CplLogger.**format**

Output format.

See also:

logging.LogRecord attributes

Key mappings in the logging output.

CplLogger.**time**

If True, attach a time tag to output messages.

CplLogger.**component**

If True, attach the component name to output messages.

CplLogger.**threadid**

If True, attach a thread tag to output messages.

cpl.esorex.**log = <cpl.esorex.CplFileLogger object>**

This variable is a CplFileLogger instance that provides a convienience file handler similar to the file logging functionality of the CPL. It basically does the same as:

```
import logging

log = logging.getLogger()
log.setLevel(logging.INFO)
ch = logging.FileHandler(filename)
```

(continues on next page)

(continued from previous page)

```
ch.setLevel(logging.INFO)
ch.setFormatter(logging.Formatter('%(asctime)s [%(levelname)7s] %(funcName)s:
↪%(message)s'))
log.addHandler(ch)
```

The following attributes control the format of the log file messages:

`CplLogger.`**`dir`**
: Directory name that is prepended to the log file name.

`CplLogger.`**`level`**
: Log level for output to the terminal. Any of [DEBUG, INFO, WARN, ERROR, OFF].

`CplLogger.`**`format`**
: Output format.

  See also:

  logging.LogRecord attributes

  Key mappings in the logging output.

`CplLogger.`**`time`**
: If `True`, attach a time tag to output messages.

`CplLogger.`**`component`**
: If `True`, attach the component name to output messages.

`CplLogger.`**`threadid`**
: If `True`, attach a thread tag to output messages.

CHAPTER 10

cpl.dfs DFS header parsing

class cpl.dfs.**ProcessingInfo**(*source*, *recno=-1*)

Support for reading input files and parameters from the FITS header of a CPL processed file.

This is done through the FITS headers that were written by the DFS function called within the processing recipe.

name
Recipe name

version
Recipe version string

pipeline
Pipeline name

cpl_version
CPL version string

tag
Tag name

calib
Calibration frames from a FITS file processed with CPL. The result of this function may directly set as *cpl.Recipe.calib* attribute:

```
import cpl
myrecipe = cpl.Recipe('muse_bias')
myrecipe.calib = cpl.dfs.ProcessingInfo('MASTER_BIAS_0.fits').calib
```

Note: This will not work properly for files that had astropy.io.fits.HDUList inputs since they have assigned a temporary file name only.

raw
Raw (input) frames

Note: This will not work properly for files that had `astropy.io.fits.HDUList` inputs since they have assigned a temporary file name only.

param
Processing parameters. The result of this function may directly set as *cpl.Recipe.param* attribute:

```
import cpl
myrecipe = cpl.Recipe('muse_bias')
myrecipe.param = cpl.dfs.ProcessingInfo('MASTER_BIAS_0.fits').param
```

md5sum
MD5 sum of the data portions of the output file (header keyword 'DATAMD5').

md5sums
MD5 sums of the input and calibration files. `dict` with the file name as key and the corresponding MD5 sum as value.

Note: Due to a design decision in CPL, the raw input files are not accompanied with the MD5 sum.

`ProcessingInfo.__init__`(*source*, *recno=-1*)

Parameters

- **source** (`str` or `astropy.io.fits.HDUList` or `astropy.io.fits.PrimaryHDU` or `astropy.io.fits.Header`) – Object pointing to the result file header
- **recno** (`int`) – Record number. Optional. If not given, the last record (with the highest record number) is used.

CHAPTER 11

Restrictions for CPL recipes

Not every information can be retrieved from recipes with the standard CPL functions. Only MUSE recipes provide additional interfaces that allow the definition of input, calibration and output frames.

All other interfaces will have the following restrictions:

1. The *Recipe.calib* attribute is not filled with templates for calibration frames. After recipe creation, this attribute is empty. Also, no check on the required calibration frames may be done before calling the recipe. Anything that is set here will be forwarded to the recipe.
2. In the *cpl.esorex* support, directly assigning the recipe calibration files from the SOF file with `recipe.calib = cpl.esorex.read_sof('file')` will also put the raw input file into *Recipe.calib* unless *Recipe.tags* and/or *Recipe.tag* are set manually. The standard recipe interface does not provide a way to distinguish between raw input and calibration files.
3. The *Recipe.tags* attribute is set to `None`.
4. The *Recipe.tag* attribute is not initially set. If this attribute is not set manually, the tag is required when executing the attribute.
5. Accessing the attribute *Recipe.output()* raises an exception.

11.1 Technical Background

CPL recipes register all their parameter definitions with the CPL function `cpl_parameterlist_append()`. All registered parameters may be retrieved from the recipe structure as a structure which contains all defined parameters.

For frames, such a mechanism does not exist, although components of the infrastructure are implemented. The CPL modules `cpl_recipeconfig` allows the definition of input, raw, and output frames for a recipe. However, this module is only half-way done, has no connection to the recipe definition and is not mandantory for CPL recipes. The MUSE pipeline recipes (with the exception of those contributed by ESO) implement a central frameconfig registry which allows to access this meta information from the Python interface.

CHAPTER 12

Feedback

Bug reports should be made on the developer web page. Send python specific questions to python-cpl@liska.ath.cx. Questions regading CPL should be mailed to cpl-help@eso.org.

Python Module Index

C

Index