Numina Documentation

Release 0.15.dev3

Sergio Pascual, Nicolás Cardiel, Pablo Picazo-Sánchez

Jul 11, 2017

Contents

Welcome. This is the Documentation for Numina (version 0.15, date Jul 11, 2017),

Numina user guide: *Numina User Guide*

Numina pipeline creation guide: *Numina Pipeline Creation Guide*

Numina reference guide: *Numina Reference*.

CHAPTER 1

Numina User Guide

This guide is intended as an introductory overview of Numina and explains how to install and make use of the most important features. For detailed reference documentation of the functions and classes contained in the package, see the *Numina Reference*.

Warning: This "User Guide" is still a work in progress; some of the material is not organized, and several aspects of Numina are not yet covered sufficient detail.

Numina Installation

This is Numina, the data reduction package used by the following GTC instruments: EMIR, FRIDA, MEGARA and MIRADAS.

Numina is distributed under GNU GPL, either version 3 of the License, or (at your option) any later version. See the file LICENSE.txt for details.

Requirements

Python >= 2.7 is required. Additionally the following packages are required in order to work properly:

- setuptools
- six
- numpy
- scipy
- astropy
- PyYaml

- singledispatch

(only if Python < 3.4)

Additional packages are optionally required:

- sphinx to build the documentation
- pytest for testing

Webpage: https://guaix.fis.ucm.es/projects/numina

Maintainer: sergiopr@fis.ucm.es

Stable version

The latest stable version of Numina can be downloaded from https://pypi.python.org/pypi/numina/

To install Numina, use the standard installation procedure:

```
$ tar zxvf numina-X.Y.Z.tar.gz
$ cd numina-X.Y.Z
$ python setup.py install
```

The *install* command provides options to change the target directory. By default installation requires administrative privileges. The different installation options can be checked with:

```
$ python setup.py install --help
```

Development version

The development version can be checked out with:

```
$ git clone https://github.com/guaix-ucm/numina.git
```

And then installed following the standard procedure:

```
$ cd numina
$ python setup.py install
```

Building the documentation

The Numina documentation is base on sphinx. With the package installed, the html documentation can be built from the *doc* directory:

```
$ cd doc
$ make html
```

The documentation will be copied to a directory under *build/sphinx*.

The documentation can be built in different formats. The complete list will appear if you type *make*

Numina Deployment with Virtualenv

Virtualenv is a tool to build isolated Python environments.

It's a great way to quickly test new libraries without cluttering your global site-packages or run multiple projects on the same machine which depend on a particular library but not the same version of the library.

Install Virtualenv

To install globally with pip (if you have pip 1.3 or greater installed globally):

```
$ sudo yum install python-virtualenv
```

For other ways of installing the package, check virtualenv_install webpage.

Create Virtual Environment

We urge reader to read the virtualenv_usage webpage to use and create new virtual environments.

As an example, a new virtual environment named numina is created where no packages but pip and setuptools are installed:

```
$ virtualenv numina
```

Activate the Environment

Once the environment is created, you need to activate it. Just go to *bin/* folder created under numina and load with your command line interpreter the script bin/activate:

```
$ cd numina/bin
$ source activate
(numina) $
```

Notice that the prompt changes once you are activate the environment. To deactivate it just type *deactivate*:

```
(numina) $ deactivate
$
```

Numina Installation

Numina is registered in the Python Package Index. That means (among other things) that can be installed inside the environment with one command:

```
(numina) $ pip install numina
```

The requirements of numina will be downloaded and installed inside the virtual environment automatically.

Numina Deployment in Solaris 10

Solaris 10 is the Operative System (OS) under a substantial part of the GTC Control System runs. The installation of the Python stack in this OS is not trivial, so in the following a description of the required steps is provided.

Basic Tools Installation

Firstly a GNU compiler collection should be installed (compilers for C, C++ and Fortran). The opencsw project provides precompiled binaries of these programs. Refer to the project's documentation to setup opencsw in the system and then install with:

```
/opt/csw/bin/pkgutil -i CSWgcc4core
/opt/csw/bin/pkgutil -i CSWgcc4g++
/opt/csw/bin/pkgutil -i CSWgcc4gfortran
```

Additionally, both the Pyhton program and the developer tools can also be installed from opencsw

```
/opt/csw/bin/pkgutil -i CSWpython27
/opt/csw/bin/pkgutil -i CSWpython27-dev
```

ATLAS and LAPACK Installation

ATLAS is a linear algebra library. Numpy can be installed without any linear algebra library, but scipy can't.

LAPACK provides Fortran routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.

ATLAS needs to be built with LAPACK support, so both libraries can be found at source code of ATLAS and source code of LAPACK.

Once the source code of ATLAS and LAPACK are downloaded, the instructions to build them can be found at build documentation which basically requires to setup a different directory to run the `configure` command in it and then `make install`.

As an example, these `configure` and `make` lines are used in our development machine:

```
../configure --cc=/opt/csw/bin/gcc --shared --with-netlib-lapack-tarfile=/path/to/
↪lapack-3.5.0.tar.gz --prefix=/opt/atlas
make
make install
```

The install step may require root privileges. The libraries and headers will be installed under some prefix (in our case, `/opt/atlas/include` and `/opt/atlas/lib`).

Numpy Installation

Download the latest numpy source code from numpy's webpage.

Numpy source distribution contains a file called `site.cfg` which describes the different types of linear algebra libraries present in the system. Copy `site.cfg.example` to `site.cfg` and edit the section containing the ATLAS libraries. Everything in the file should be commented except the following

```
[atlas]
library_dirs = /opt/atlas/lib
include_dirs = /opt/atlas/include
```

The paths should point to the version of ATLAS installed in the system.

Other packages (such as scipy) will also use a `site.cfg` file. To avoid editing the same file again, copy `site.cfg` to `.numpy-site.cfg` in the `$HOME` directory.

```
cp site.cfg $HOME/.numpy-site.cfg
```

After this configuration step, numpy should be built.

```
python setup.py build
python setup.py install --prefix /path/to/my/python/packages
```

The last step may require root privileges. Notice that you can use `--user` instead of `--prefix` for local packages.

Scipy Installation

As of this writing, the last released version of scipy is 0.15.1 and it doesn't work in Solaris 10 due to a bug[1].

This bug may be fixed in next stable release (check the release notes of scipy), but meanwhile a patch can be used.

Download the scipy 0.15.1 source code from scipy's webpage. Then download the patch: scipy151-solaris10.patch.

Extract the source code and apply the patch with the command:

```
patch -p1 -u -d scipy-0.15.1 < scipy151-solaris10.patch
```

After this step, build and install scipy normally.

```
python setup.py build
python setup.py install --prefix /path/to/my/python/packages
```

During the build step, local `.numpy-site.cfg` will be read so the path to the ATLAS libraries will be used.

The prefix used to install scipy must be the same than the used with numpy. In general all python packages must be installed under the same prefix.

Pip Installation

To install pip, download get-pip.py.

Then run the following:

```
python get-pip.py
```

Refer to https://pip.pypa.io/en/latest/installing.html#install-pip to more detailed documentation.

Numina Installation

Finally, numina can be installed directly using `pip`. Remember to set the same prefix used previously with numpy and scipy.

[1] https://github.com/scipy/scipy/issues/4704

```
pip install numina --prefix /path/to/my/python/packages
```

Command Line Interface

The **numina** script is the interface with the pipelines It is called like this:

```
$ numina [global-options] comands [comand-options]
```

The **numina** script has several options:

-d, --debug
Debug enabled, increases verbosity.

-l filename
A file con configuration options for logging.

Options for run

The run subcommand processes the observing result with the appropriated reduction recipe.

It is called like this:

```
$ numina [global-options] run [comand-options] observation-result.yaml
```

--instrument 'name'
Name of one of the predefined instrument configurations.

--pipeline 'name'
Name of one of the predefined pipelines.

--requirements filename
File with the description of the parameters of the recipe.

--basedir path
File path used to resolve relative paths in the following options.

--datadir path
File path to the folder containing the pristine data to be processed.

--resultsdir path
File path to the directory where results are stored.

--workdir path
File path to the a directory where the recipe can write. Files in datadir are copied here.

--cleanup
Remove intermediate and temporal files created by the recipe.

observing_result filename
Filename contaning the description of the observation result.

Options for show-instruments

The show-instruments subcommand outputs information about the instruments with available pipelines.

It is called like this:

```
$ numina [global-options] show-instruments [options]
```

-o, --observing-modes
Show names and keys of Observing Modes in addition of instrument information.

name
Name of the instruments to show. If empty show all instruments.

Options for show-modes

The show-modes subcommand outputs information about the observing modes of the available instruments.

It is called like this:

```
$ numina [global-options] show-modes [options]
```

-i, --instrument name
Filter modes by instrument name.

name
Name of the observing mode to show. If empty show all observing modes.

Options for show-recipes

The show-recipes subcommand outputs information about the recipes of the available instruments.

It is called like this:

```
$ numina [global-options] show-recipes [options]
```

-i, --instrument name
Filter recipes by instrument name.

-t, --template
Generate a template file to be used a requirement file by **numina run**.

name
Name of the recipe to show. If empty show all recipes.

CHAPTER 2

Numina Pipeline Creation Guide

This guide is intended as an introductory overview of pipeline creation with Numina. For detailed reference documentation of the functions and classes contained in the package, see the *Numina Reference*.

Warning: This "Pipeline Creation Guide" is still a work in progress; some of the material is not organized, and several aspects of Numina are not covered in sufficient detail yet.

Numina Pipeline Concepts

Instrument

Observing Modes

Each Instrument has a list of predefined types of observations that can be carried out with it. Each Observing Mode is defined by:

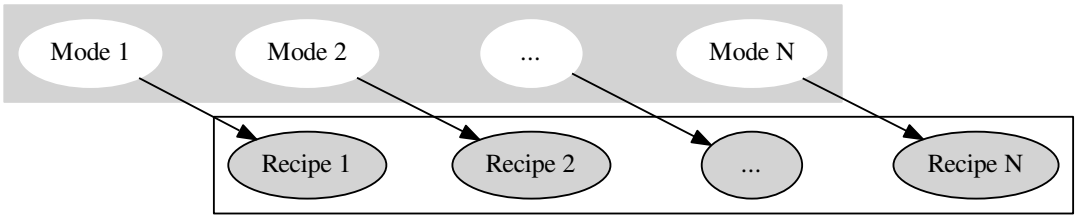
- The configuration of the Telescope
- The configuration of the Instrument
- The type of processing required by the images obtained durinf the observation

Some of the observing modes of a Instrument are **Scientific**, that is, modes devoted to obtain data to perform scientific analysis. Other modes are devoted to **Calibration**; these modes produce data required to correct the scientific images from the effects of the Instrument, the Telescope and the atmosphere.

Recipes

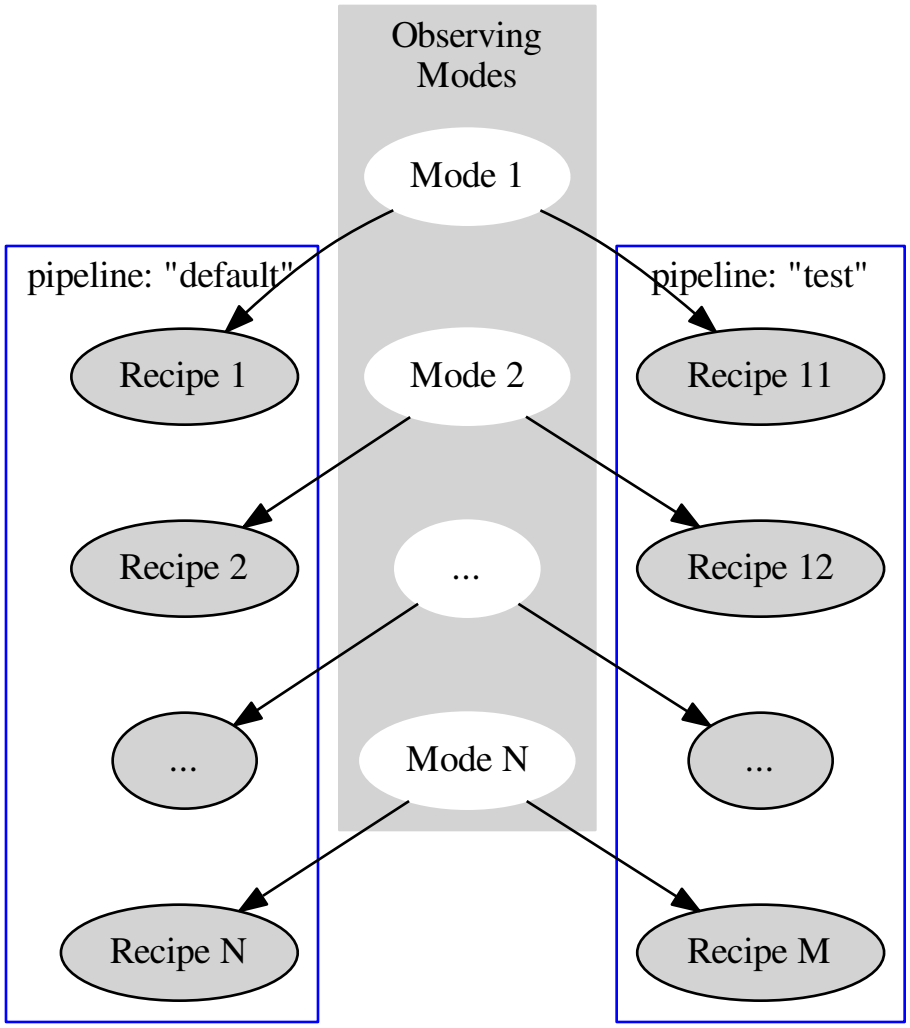
A recipe is a method to process the images obtained in a particular observing mode. Recipes in general require (as inputs) the list of raw images obtained during the observation. Recipes can require other inputs (calibrations), and those inputs can be the outputs of other recipes.

Images obtained in a particular mode are processed by one recipe.



Pipelines

A pipeline represents a particular mapping between the observing modes and the reduction algorithms that process each mode. Each instrument has at least one pipeline called *default*. It may have other pipelines for specific purposes.



Products, Requirements and Data Types

A recipe announces its required inputs as `Requirement` and its outputs as `Product`.

Both Products and Requirements have a name and a type. Types can be plain Python types or defined by the developer.

Format of the input files

The default format of the input and output files is YAML, a data serialization language.

Format of the Observation Result file

This file contains the result of an observation. It represents an *ObservationResult* object.

The contents of the object are serialized as a dictionary with the following keys:

id: not required, integer, defaults to 1 Unique identifier of the observing block

instrument: required, string Name of the instrument, as it appears in the instrument file (see below)

mode: required, string Name of the observing mode

children: not required, list of integers, defaults to empty list Identifications of nested observing blocks

frames: required, list of file names List of raw images

```
id: 21
instrument: EMIR
mode: nb_image
children: []
frames:
- r0121.fits
- r0122.fits
- r0123.fits
- r0124.fits
- r0125.fits
- r0126.fits
- r0127.fits
- r0128.fits
- r0129.fits
- r0130.fits
- r0131.fits
- r0132.fits
```

Format of the requirement file (version 1)

```
version: 1
products:
  EMIR:
   - {id: 1, content: 'file1.fits', type: 'MasterFlat', tags: {'filter': 'J'}, ob:␣
↪200}
   - {id: 4, content: 'file4.fits', type: 'MasterBias', tags: {'readmode': 'cds'},␣
↪ob: 400}
  MEGARA:
   - {id: 1, content: 'file1.fits', type: 'MasterFlat', tags: {'vph': 'LR1'}, ob:␣
↪1200}
   - {id: 2, content: 'file2.yml', type: 'TraceMap', tags: {'vph': 'LR2', 'readmode':
↪'fast'}, ob: 1203}
requirements:
  EMIR:
    default:
       TEST6:
          pinhole_nominal_positions: [ [0, 1], [0 , 1]]
          box_half_size: 5
       TEST9:
          median_filter_size: 5
MEGARA:
    default:
       mos_image: {}
```

Format of the requirement file

Warning: This section documents a deprecated format

Deprecated since version 0.14.0.

This file contains configuration parameters for the recipes that are not related to the particular instrument used.

The contents of the file are serialized as a dictionary with the following keys:

requirements: required, dictionary A dictionary of parameter names and values.

logger: optional, dictionary A dictionary used to configure the custom file logger

```
requirements:
  master_bias: master_bias-1.fits
  master_bpm: bpm.fits
  master_dark: master_dark-1.fits
  master_intensity_ff: master_flat.fits
  nonlinearity: [1.0, 0.0]
  subpixelization: 4
  window:
  - [800, 1500]
  - [800, 1500]
logger:
  logfile: processing.log
  format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
  enabled: true
```

Generating template requirement files

Template requirement files can be generated by **`numina show-recipes`** The flag generates templates for the named recipe or for all the available recipes if no name is passed.

For example:

```
$ numina show-recipes -t emir.recipes.DitheredImageRecipe
# This is a numina 0.9.0 template file
# for recipe 'emir.recipes.DitheredImageRecipe'
#
# The following requirements are optional:
#  sources: None
#  master_bias: master_bias.fits
#  offsets: None
# end of optional requirements
requirements:
  check_photometry_actions: [warn, warn, default]
  check_photometry_levels: [0.5, 0.8]
  extinction: 0.0
  iterations: 4
  master_bpm: master_bpm.fits
  master_dark: master_dark.fits
  master_intensity_ff: master_intensity_ff.fits
  nonlinearity: [1.0, 0.0]
  sky_images: 5
  sky_images_sep_time: 10
```

```
#products:
# catalog: None
# frame: frame.fits
#logger:
# logfile: processing.log
# format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
# enabled: true
---
```

The # character is a comment, so every line starting with it can safely removed. The names of FITS files in requirements must be edited to point to existing files.

Numina Pipeline Example

This guide is intended as an introductory overview of the creation of instrument reduction pipelines with Numina. For detailed reference documentation of the functions and classes contained in the package, see the *Numina Reference*.

Warning: This "Pipeline Creation Guide" is still a work in progress; some of the material is not organized, and several aspects of Numina are not yet covered sufficient detail.

Execution environment of the Recipes

Recipes have different execution environments. Some recipes are designed to process observing modes required for the observation. These modes are related to visualization, acquisition and focusing. The Recipes are integrated in the GTC environment. We call these recipes the **Data Factory Pipeline**, (*DFP*).

Other group of recipes are devoted to scientific observing modes: imaging, spectroscopy and auxiliary calibrations. These Recipes constitute the **Data Reduction Pipeline**, (*DRP*). The software is meant to be standalone, users shall download the software and run it in their own computers, with reduction parameters and calibrations provided by the instrument team.

Users of the DRP will use the simple Numina CLI. Users of the DFP shall interact with the software through the GTC Inspector.

Instrument Reduction Pipeline Example

In the following sections we create an Instrument Reduction Pipeline for an instrument name *CLODIA*.

In order to make a new Instrument Reduction Pipeline visible to Numina and the GTC Control System you have to create a full Python package that will contain the reduction recipes, data types and other processing code.

The creation of Python packages is described in detail (for example) in the Python Packaging User Guide.

Then, we create a Python package called *clodiadrp* with the following structure (we ignore files such as README or LICENSE as they are not relevant here):

```
clodiadrp
|-- clodiadrp
|   |-- __init__.py
|-- setup.py
```

From here the steps are:

1. Create a configuration yaml file.
2. Create a loader file.
3. Link the *entry_point* option in the *setup.py* with the loader file.
4. Create the Pipeline's Recipes.

In the following we will continue with the same example as previously.

Configuration File

The configuration file contains basic information such as:

- the list of modes of the instrument
- the list of recipes of the instrument
- the mapping between recipes and modes.

In this example, we assume that CLODIA has three modes: **Bias**, **Flat** and **Image**. The first two modes are used for pedestal and flat-field illumination correction. The third is the main scientific mode of the instrument.

Create a new yaml file in the root folder named *drp.yaml*.

```
name: CLODIA
configurations:
  default: {}
modes:
 -key: bias
  name: Bias
  summary: Bias mode
  description: >
    Full description of the Bias mode
 -key: flat
  name: Flat
  summary: Flat mode
  description: >
    Full description of the Flat mode
 -key: image
  name: Image
  summary: Image mode
  description: >
    Full description of the Image mode
pipelines:
  default:
    version: 1
    recipes:
      bias: clodiadrp.recipes.recipe
      flat: clodiadrp.recipes.recipe
      image: clodiadrp.recipes.recipe
```

The entry *modes* contains a list of the observing modes of the instrument. There are three: Bias, Flat and Image. Each entry contains information about the mode. A *name*, a short *summary* and a multi-line *description*. The field *key* is used to map the observing modes and the *recipes*, so *key* has to be unique and equal to only one value in each *recipes* block under *pipelines*.

The entry *pipelines* contains only one pipeline, called *default* by convention. The *pipeline* contains recipes, each related to one observing mode by means of the filed *key*. For the moment we haven't developed any recipe, so the value of each key (*clodiadrp.recipes.recipe*) doesn't exist yet.

Note: This file has to be included in *package_data* inside *setup.py* to be distributed with the package, see Installing Package Data for details.

Loader File

Create a new loader file in the root folder named *loader.py* with the following information:

```
import numina.core

def drp_load():
    """Entry point to load CLODIA DRP."""
    return numina.core.drp_load('clodiadrp', 'drp.yaml')
```

Create entry point

Once we have created the *loader.py* file, the only thing we have to do is to make CLODIA visible to Numina/GCS. To do so, just modify the *setup.py* file to add an entry point.

```
from setuptools import setup

setup(name='clodiadrp',
      entry_points = {
        'numina.pipeline.1': ['CLODIA = clodiadrp.loader:drp_load'],
        },
)
```

Both the Numina CLI tool and GCS check this particular entry point. They call the function provided by the entry point. The function *drp_load()* reads and parses the YAML file and creates an object of class *InstrumentDRP* for each recipes it founds. These objects are used by Numina CLI and GCS to discover the available Instrument Reduction Pipelines.

At this stage, the file layout is as follows:

```
clodiadrp
|-- clodiadrp
|   |-- __init__.py
|   |-- loader.py
|   |-- drp.yaml
|-- setup.py
```

Note: In fact, it is not necessary to use a YAML file to contain the Instrument information. The only strict requirement is that the function in the entry point 'numina.pipeline.1' must return a valid *InstrumentDRP* object. The use of a YAML file and the *drp_load()* function is only a matter of convenience.

Recipes Creation

We haven't created any reduction recipe yet. As a matter of organization, we suggest to create a dedicated subpackage for recipes *clodiadrp.recipes* and a module for each recipe. The file layout is:

```
clodiadrp
|-- clodiadrp
|   |-- __init__.py
|   |-- loader.py
|   |-- drp.yaml
|   |-- recipes
|   |   |-- __init__.py
|   |   |-- bias.py
|   |   |-- flat.py
|   |   |-- image.py
|-- setup.py
```

Recipes must provide three things: 1) a description of the inputs of the recipe; 2) a description of the products of the recipe and 3) a *run* method which is in charge of executing the proccessing. Additionally, all Recipes must inherit from *BaseRecipe*.

We start with a simple *Bias* recipe. Its purpose is to process images previously taken in *Bias* mode, that is, a series of pedestal images. The recipe will receive the result of the observation and return a master bias image.

```
from numina.core import Product, Requirement
from numina.core import DataFrameType
from numina.core.products import ObservationResultType
from numina.core.recipes import BaseRecipe

class Bias(BaseRecipe):                                (1)

    obresult = Requirement(ObservationResultType)      (2)
    master_bias = Product(DataFrameType)               (3)

    def run(self, rinput):                             (4)

        # Here the raw images are processed
        # and a final image myframe is created

        result = self.create_result(master_bias=myframe)   (5)
        return result
```

1. Each recipe must be a class derived from *BaseRecipe*
2. This recipe only requires the result of the observation. Each requirement is an object of the *Requirement* class or any subclass of it. The type of the requirement is *ObservationResultType*, representing the result of the observation.
3. This recipe only produces one result. Each product is an object of *Product* class. The type of the product is given by *DataFrameType*, representing an image.
4. Each recipe must provide a *run* method. The method has only one argument that collects the values of all inputs declared by the recipe. In this case, *rinput* has a member named *obresult* and can be accessed through *rinput.obresult* which belongs to *ObservationResult* class.
5. The recipe must return an object that collects all the declared products of the recipe, of *RecipeResult* class. This is accomplished internally by the *create_result* method. It will raise a run time exception if any of the declared products are not provided.

We can now create the *Flat* recipe (inside *flat.py*). This recipe has two requirements, the observation result and a master bias image (flat-field images require bias subtraction).

```
from numina.core import Product, Requirement
from numina.core import DataFrameType
```

```
from numina.core.products import ObservationResultType
from numina.core.recipes import BaseRecipe

class Flat(BaseRecipe):

    obresult = Requirement(ObservationResultType)   (1)
    master_bias = Requirement(DataFrameType)        (2)
    master_flat = Product(DataFrameType)

    def run(self, rinput):                           (3)

        # Here the raw images are processed
        # and a final image myframe is created

        result = self.create_result(master_flat=myframe)   (4)
        return result
```

1. This recipe only requires the result of the observation. Each requirement is an object of the *Requirement* class or any subclass of it. The type of the requirement is *ObservationResultType*, representing the result of the observation.
2. It also requires a master bias image which belongs to *DataFrameType* class (represents an image).
3. In this case, *rinput* has two members: 1) *rinput.obresult* of *ObservationResult* class and 2) a *rinput.master_bias* of *DataFrame* class
4. The arguments of *create_result* must be the same names used in the product definition.

Finally, the recipe for *Image* mode reduction (inside *image.py*) has three requirements, the observation result, a master bias and a master flat images

```
from numina.core import Product, Requirement
from numina.core import DataFrameType
from numina.core.products import ObservationResultType
from numina.core.recipes import BaseRecipe

class Image(BaseRecipe):

    obresult = Requirement(ObservationResultType)
    master_bias = Requirement(DataFrameType)
    master_flat = Requirement(DataFrameType)
    final = Product(DataFrameType)

    def run(self, rinput):                          (1)

        # Here the raw images are processed
        # and a final image myframe is created

        result = self.create_result(final=myframe)
        return result
```

1. In this case, *rinput* will have three members *rinput.obresult* of *ObservationResult* class, *rinput.master_bias* of *DataFrame* class and *rinput.master_flat* of *DataFrame* class.

Note: It is not strictly required that the requirements and products names are consistent between recipes, although it is highly recommended.

Now we must update *drp.yaml* to insert the full name of the recipes (package and class), as follows

```
name: CLODIA
configurations:
  default: {}
modes:
 -key: bias
  name: Bias
  summary: Bias mode
  description: >
    Full description of the Bias mode
 -key: flat
  name: Flat
  summary: Flat mode
  description: >
    Full description of the Flat mode
 -key: image
  name: Image
  summary: Image mode
  description: >
    Full description of the Image mode
pipelines:
  default:
    version: 1
    recipes:
      bias: clodiadrp.recipes.bias.Bias
      flat: clodiadrp.recipes.flat.Flat
      image: clodiadrp.recipes.image.Image
```

Specialized data products

There is some information that is missing of our current setup. The products of some recipes are the inputs of others. The master bias created by *Bias* is the input that *Flat* and *Image* require. To represent this situation we use specialized data products. We start by adding a new module *products*:

```
clodiadrp
|-- clodiadrp
|   |-- __init__.py
|   |-- loader.py
|   |-- products.py
|   |-- drp.yaml
|   |-- recipes
|   |   |-- __init__.py
|   |   |-- bias.py
|   |   |-- flat.py
|   |   |-- image.py
|-- setup.py
```

We have two types of images that are products of recipes that can be required by other recipes: **master bias** and **master flat**. We represent this by creating two new types derived from *DataFrameType* (becasue the new types are images) and *DataProductTag* (because the new types are products that must be handled by both Numina CLI and GTC Control system) classes.

```
from numina.core.products import DataFrameType, DataProductTag

class MasterBias(DataFrameType, DataProductTag):
    pass
```

```
class MasterFlat(DataFrameType, DataProductTag):
    pass
```

Now we must modify our recipes as follows. First *Bias*

```
from numina.core import Product, Requirement
from numina.core.products import ObservationResultType
from numina.core.recipes import BaseRecipe
from clodiadrp.products import MasterBias   (1)

class Bias(BaseRecipe):

    obresult = Requirement(ObservationResultType)
    master_bias = Product(MasterBias)         (2)

    ...                                        (3)
```

1. Import the new type *MasterBias*.
2. Declare that our recipe produces *MasterBias* images.
3. *run* method remains unchanged.

Then *Flat*:

```
from numina.core import Product, Requirement
from numina.core.products import ObservationResultType
from numina.core.recipes import BaseRecipe
from clodiadrp.products import MasterBias, MasterFlat

class Flat(BaseRecipe):

    obresult = Requirement(ObservationResultType)
    master_bias = Requirement(MasterBias)          (1)
    master_flat = Product(MasterFlat)              (2)

    ...                                        (3)
```

1. *MasterBias* is used as a requirement. This guaranties that the images provided here are those created by *Bias* and no other.
2. Declare that our recipe produces *MasterFlat* images.
3. *run* method remains unchanged.

And finally *Image*:

```
from numina.core import Product, Requirement
from numina.core import DataFrameType
from numina.core.products import ObservationResultType
from numina.core.recipes import BaseRecipe
from clodiadrp.products import MasterBias, MasterFlat

class Image(BaseRecipe):

    obresult = Requirement(ObservationResultType)
    master_bias = Requirement(MasterBias)        (1)
    master_flat = Requirement(MasterFlat)        (2)
```

```
    final = Product(DataFrameType)                (3)

    ...                                          (4)
```

1. *MasterBias* is used as a requirement. This guaranties that the images provided here are those created by *Bias* and no other.
2. *MasterFlat* is used as a requirement. This guaranties that the images provided here are those created by *Flat* and no other.
3. Declare that our recipe produces *Image* images.
4. *run* method remains unchanged.

DRP Data Types

Custom data types can be used as Requirements and Products by Recipes. New data types can be derived as follows.

Create a new DataType

New Data Types must derive from *numina.core.DataType* or one of its subclasses. In the constructor, we must declare the base type of the objects ot this Data Product.

For example, a *MasterBias* Data Product is an image, so its base type is a *DataFrame*. A table of 2D coordinates will have a *numpy.ndarray* base type.

In general, we are interested in defining new DataTypes for objects that will contain information that will be used as inputs in different recipes. In this case, we must derive from *numina.core.DataProductType*.

As an example, we create a DataType that will store information about the trace of a spectrum. The information will be stored in Python *dict*.

```
class TraceMap(DataProductType):
    def __init__(self, default=None):
        super(TraceMap, self).__init__(dict, default)
```

Construction of objects

The input of a recipe is created by inspecting the Recipe Requirements. The Recipe Loader is in charge of finding an appropriated value for each requirement. The value is passed to *Requirement.convert*, that in turn calls *DataType.convert*. The default implementation just returns in input object unchanged.

Loading and Storage with the command line Recipe Loader

Each Recipe Loader can implement its own mechanism to store and load Data Products. The Command Line Recipe Loader uses text files in YAML format.

To define how a particular DataProduct is stored under the default Recipe Loader, two functions must be defined, a store function and a load function. Then thse two functions must be registered with the global functions *numina.store.dump* and *numina.store.load*.

```
from numina.store import dump, load

from .products import TraceMap

@dump.register(TraceMap)
def dump_tracemap(tag, obj, where):

    filename = where.destination + '.yaml'

    with open(filename, 'w') as fd:
        yaml.dump(obj, fd)

    return filename

@load.register(TraceMap)
def load_tracemap(tag, obj):

    with open(obj, 'r') as fd:
        traces = yaml.load(fd)

    return traces
```

In this example, *tag* is an argument of type *TraceMap* and *obj* is of type *dict*.

CHAPTER 3

Numina Reference

Release 0.15

Date Jul 11, 2017

Warning: This "Reference" is still a work in progress; some of the material is not organized, and several aspects of Numina are not yet covered sufficient detail.

Numina modules

`numina.array` — Array manipulation

`numina.array.`**`fixpix`**(*data*, *mask*, *kind='linear'*)
Interpolate 2D array data in rows

`numina.array.`**`fixpix2`**(*data*, *mask*, *iterations=3*, *out=None*)
Substitute pixels in mask by a bilinear least square fitting.

`numina.array.`**`numberarray`**(*x*, *shape*)
Return x if it is an array or create an array and fill it with x.

`numina.array.`**`rebin`**(*a*, *newshape*)
Rebin an array to a new shape.

`numina.array.`**`rebin_scale`**(*a*, *scale=1*)
Scale an array to a new shape.

`numina.array.`**`subarray_match`**(*shape*, *ref*, *sshape*, *sref=None*)
Compute the slice representation of intersection of two arrays.

Given the shapes of two arrays and a reference point ref, compute the intersection of the two arrays. It returns a tuple of slices, that can be passed to the two images as indexes

Parameters

- **shape** – the shape of the reference array
- **ref** – coordinates of the reference point in the first array system
- **sshape** – the shape of the second array

Param sref: coordinates of the reference point in the second array system, the origin by default

Returns two matching slices, corresponding to both arrays or a tuple of Nones if they don't match

Return type a tuple

Example

```
>>> import numpy
>>> im = numpy.zeros((1000, 1000))
>>> sim = numpy.ones((40, 40))
>>> i,j = subarray_match(im.shape, [20, 23], sim.shape)
>>> im[i] = 2 * sim[j]
```

numina.array.**process_ramp**(*inp*[, *out=None*, *axis=2*, *ron=0.0*, *gain=1.0*, *nsig=4.0*, *dt=1.0*, *saturation=65631*])

New in version 0.8.2.

Compute the result 2d array computing slopes in a 3d array or ramp.

Parameters

- **inp** – input array
- **out** – output array
- **axis** – unused
- **ron** – readout noise of the detector
- **gain** – gain of the detector
- **nsig** – rejection level to detect glitched and cosmic rays
- **dt** – time interval between exposures
- **saturation** – saturation level

Returns a 2d array

numina.array.background — Background estimation

Background estimation

Background estimation following Costa 1992, Bertin & Arnouts 1996

numina.array.background.**background_estimator**(*bdata*)

Estimate the background in a 2D array

numina.array.background.**create_background_map**(*data*, *bsx*, *bsy*)

Create a background map with a given mesh size

numina.array.blocks — Generation of blocks

numina.array.blocks.**blk_1d**(*blk*, *shape*)
Iterate through the slices that recover a line.

This function is used by *blk_nd()* as a base 1d case.

The last slice is returned even if is lesser than blk.

Parameters

- **blk** – the size of the block
- **shape** – the size of the array

Returns a generator that yields the slices

numina.array.blocks.**blk_1d_short**(*blk*, *shape*)
Iterate through the slices that recover a line.

This function is used by *blk_nd_short()* as a base 1d case.

The function stops yielding slices when the size of the remaining slice is lesser than *blk*.

Parameters

- **blk** – the size of the block
- **shape** – the size of the array

Returns a generator that yields the slices

numina.array.blocks.**blk_coverage_1d**(*blk*, *size*)
Return the part of a 1d array covered by a block.

Parameters

- **blk** – size of the 1d block
- **size** – size of the 1d a image

Returns a tuple of size covered and remaining size

Example

```
>>> blk_coverage_1d(7, 100)
(98, 2)
```

numina.array.blocks.**blk_nd**(*blk*, *shape*)
Iterate through the blocks that cover an array.

This function first iterates trough the blocks that recover the part of the array given by max_blk_coverage and then iterates with smaller blocks for the rest of the array.

Parameters

- **blk** – the N-dimensional shape of the block
- **shape** – the N-dimensional shape of the array

Returns a generator that yields the blocks

Example

```
>>> result = list(blk_nd(blk=(5,3), shape=(11, 11)))
>>> result[0]
(slice(0, 5, None), slice(0, 3, None))
>>> result[1]
(slice(0, 5, None), slice(3, 6, None))
>>> result[-1]
(slice(10, 11, None), slice(9, 11, None))
```

The generator yields blocks of size blk until it covers the part of the array given by *max_blk_coverage()* and then yields smaller blocks until it covers the full array.

See also:

blk_nd_short() Yields blocks of fixed size

numina.array.blocks.**blk_nd_short**(*blk*, *shape*)
Iterate trough the blocks that strictly cover an array.

Iterate trough the blocks that recover the part of the array given by max_blk_coverage.

Parameters

- **blk** – the N-dimensional shape of the block
- **shape** – the N-dimensional shape of the array

Returns a generator that yields the blocks

Example

```
>>> result = list(blk_nd_short(blk=(5,3), shape=(11, 11)))
>>> result[0]
(slice(0, 5, None), slice(0, 3, None))
>>> result[1]
(slice(0, 5, None), slice(3, 6, None))
>>> result[-1]
(slice(5, 10, None), slice(6, 9, None))
```

In this case, the output of max_blk_coverage is (10, 9), so only this part of the array is covered

See also:

blk_nd() Yields blocks of blk size until the remaining part is smaller than *blk* and the yields smaller blocks.

numina.array.blocks.**block_view**(*arr*, *block=(3, 3)*)
Provide a 2D block view to 2D array.

No error checking made. Therefore meaningful (as implemented) only for blocks strictly compatible with the shape of A.

numina.array.blocks.**blockgen**(*blocks*, *shape*)
Generate a list of slice tuples to be used by combine.

The tuples represent regions in an N-dimensional image.

Parameters

- **blocks** – a tuple of block sizes
- **shape** – the shape of the n-dimensional array

Returns an iterator to the list of tuples of slices

Example

```
>>> blocks = (500, 512)
>>> shape = (1040, 1024)
>>> for i in blockgen(blocks, shape):
...     print i
(slice(0, 260, None), slice(0, 512, None))
(slice(0, 260, None), slice(512, 1024, None))
(slice(260, 520, None), slice(0, 512, None))
(slice(260, 520, None), slice(512, 1024, None))
(slice(520, 780, None), slice(0, 512, None))
(slice(520, 780, None), slice(512, 1024, None))
(slice(780, 1040, None), slice(0, 512, None))
(slice(780, 1040, None), slice(512, 1024, None))
```

numina.array.blocks.**blockgen1d**(*block*, *size*)

Compute 1d block intervals to be used by combine.

blockgen1d computes the slices by recursively halving the initial interval (0, size) by 2 until its size is lesser or equal than block

Parameters

- **block** – an integer maximum block size
- **size** – original size of the interval, it corresponds to a 0:size slice

Returns a list of slices

Example

```
>>> blockgen1d(512, 1024)
[slice(0, 512, None), slice(512, 1024, None)]
```

numina.array.blocks.**max_blk_coverage**(*blk*, *shape*)

Return the maximum shape of an array covered by a block.

Parameters

- **blk** – the N-dimensional shape of the block
- **shape** – the N-dimensional shape of the array

Returns the shape of the covered region

Example

```
>>> max_blk_coverage(blk=(7, 6), shape=(100, 43))
(98, 42)
```

numina.array.bpm — Bad Pixel Mask interpolation

Fix points in an image given by a bad pixel mask

numina.array.combine — Array combination

Different methods for combining lists of arrays.

numina.array.combine.**flatcombine**(*arrays*, *masks=None*, *dtype=None*, *scales=None*, *low=3.0*, *high=3.0*, *blank=1.0*)

Combine flat arrays.

Parameters

- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **out** – optional output, with one more axis than the input arrays
- **blank** – non-positive values are substituted by this on output

Returns mean, variance of the mean and number of points stored

numina.array.combine.**generic_combine**(*method*, *arrays*, *masks=None*, *dtype=None*, *out=None*, *zeros=None*, *scales=None*, *weights=None*)

Stack arrays using different methods.

Parameters

- **method** (*PyCObject*) – the combination method
- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **zeros** –
- **scales** –
- **weights** –

Returns median, variance of the median and number of points stored

numina.array.combine.**mean**(*arrays*, *masks=None*, *dtype=None*, *out=None*, *zeros=None*, *scales=None*, *weights=None*)

Combine arrays using the mean, with masks and offsets.

Arrays and masks are a list of array objects. All input arrays have the same shape. If present, the masks have the same shape also.

The function returns an array with one more dimension than the inputs and with size (3, shape). out[0] contains the mean, out[1] the variance and out[2] the number of points used.

Parameters

- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **out** – optional output, with one more axis than the input arrays

Returns mean, variance of the mean and number of points stored

Example

```
>>> import numpy
>>> image = numpy.array([[1., 3.], [1., -1.4]])
>>> inputs = [image, image + 1]
>>> mean(inputs)
array([[[ 1.5,  3.5],
        [ 1.5, -0.9]],

       [[ 0.5,  0.5],
        [ 0.5,  0.5]],

       [[ 2. ,  2. ],
        [ 2. ,  2. ]]])
```

numina.array.combine.**median**(*arrays*, *masks=None*, *dtype=None*, *out=None*, *zeros=None*, *scales=None*, *weights=None*)

Combine arrays using the median, with masks.

Arrays and masks are a list of array objects. All input arrays have the same shape. If present, the masks have the same shape also.

The function returns an array with one more dimension than the inputs and with size (3, shape). out[0] contains the mean, out[1] the variance and out[2] the number of points used.

Parameters

- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **out** – optional output, with one more axis than the input arrays

Returns median, variance of the median and number of points stored

numina.array.combine.**minmax**(*arrays*, *masks=None*, *dtype=None*, *out=None*, *zeros=None*, *scales=None*, *weights=None*, *nmin=1*, *nmax=1*)

Combine arrays using mix max rejection, with masks.

Inputs and masks are a list of array objects. All input arrays have the same shape. If present, the masks have the same shape also.

The function returns an array with one more dimension than the inputs and with size (3, shape). out[0] contains the mean, out[1] the variance and out[2] the number of points used.

Parameters

- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **out** – optional output, with one more axis than the input arrays
- **nmin** –
- **nmax** –

Returns mean, variance of the mean and number of points stored

numina.array.combine.**quantileclip**(*arrays*, *masks=None*, *dtype=None*, *out=None*, *zeros=None*, *scales=None*, *weights=None*, *fclip=0.1*)

Combine arrays using the sigma-clipping, with masks.

Inputs and masks are a list of array objects. All input arrays have the same shape. If present, the masks have the same shape also.

The function returns an array with one more dimension than the inputs and with size (3, shape). out[0] contains the mean, out[1] the variance and out[2] the number of points used.

Parameters

- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **out** – optional output, with one more axis than the input arrays
- **fclip** – fraction of points removed on both ends. Maximum is 0.4 (80% of points rejected)

Returns mean, variance of the mean and number of points stored

numina.array.combine.**sigmaclip**(*arrays*, *masks=None*, *dtype=None*, *out=None*, *zeros=None*, *scales=None*, *weights=None*, *low=3.0*, *high=3.0*)

Combine arrays using the sigma-clipping, with masks.

Inputs and masks are a list of array objects. All input arrays have the same shape. If present, the masks have the same shape also.

The function returns an array with one more dimension than the inputs and with size (3, shape). out[0] contains the mean, out[1] the variance and out[2] the number of points used.

Parameters

- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **out** – optional output, with one more axis than the input arrays
- **low** –
- **high** –

Returns mean, variance of the mean and number of points stored

numina.array.combine.**zerocombine**(*arrays*, *masks*, *dtype=None*, *scales=None*)

Combine zero arrays.

Parameters

- **arrays** – a list of arrays
- **masks** – a list of mask arrays, True values are masked
- **dtype** – data type of the output
- **scales** –

Returns median, variance of the median and number of points stored

Combination methods in `numina.array.combine`

All these functions return a `PyCapsule`, that can be passed to *`generic_combine()`*

`numina.array.combine.`**`mean_method`**`()`
Mean method

`numina.array.combine.`**`median_method`**`()`
Median method

`numina.array.combine.`**`sigmaclip_method`**`(`[*low=0.0*[, *high=0.0*]]`)`
Sigmaclip method

Parameters

- **`low`** – Number of sigmas to reject under the mean
- **`high`** – Number of sigmas to reject over the mean

Raises `ValueError` if **low** or **high** are negative

`numina.array.combine.`**`quantileclip_method`**`(`[*fclip=0.0*]`)`
Quantile clip method

Parameters **`fclip`** – Fraction of points to reject on both ends

Raises `ValueError` if **fclip** is negative or greater than 0.4

`numina.array.combine.`**`minmax_method`**`(`[*nmin=0*[, *nmax=0*]]`)`
Min-max method

Parameters

- **`nmin`** – Number of minimum points to reject
- **`nmax`** – Number of maximum points to reject

Raises `ValueError` if **nmin** or **nmax** are negative

Extending `generic_combine()`

New combination methods can be implemented and used by *`generic_combine()`* The combine function expects a `PyCapsule` object containing a pointer to a C function implementing the combination method.

int **`combine`**(double **data*, double **weights*, size_t *size*, double **out[3]*, void **func_data*)
Operate on two arrays, containing **data** and **weights**. The result, its variance and the number of points used in the calculation (useful when there is some kind of rejection) are stored in **out[0]**, **out[1]** and **out[2]**.

Parameters

- **`data`** – a pointer to an array containing the data
- **`weights`** – a pointer to an array containing weights
- **`size`** – the size of data and weights
- **`out`** – an array of pointers to the pixels in the result arrays
- **`func_data`** – additional parameters of the function encoded as a void pointer

Returns 1 if operation succeeded, 0 in case of error.

If the function uses dynamically allocated data stored in *func_data*, we must also implement a function that deallocates the data once it is used.

void **destructor_function** (PyObject* *cobject*)

Parameters

- **cobject** – the object owning dynamically allocated data

Simple combine method

As an example, I'm going to implement a combination method that returns the minimum of the input arrays. Let's call the method *min_method*

First, we implement the C function. I'm going to use some C++ here (it makes the code very simple).

```
int min_combine(double *data, double *weights, size_t size, double *out[3],
          void *func_data) {

    double* res = std::min_element(data, data + size);

    *out[0] = *res;
    // I'm not going to compute the variance for the minimum
    // but it should go here
    *out[1] = 0.0;
    *out[2] = size;

    return 1;
}
```

A destructor function is not needed in this case as we are not using *func_data*.

The next step is to build a Python extension. First we need to create a function returning the `PyCapsule` in C code like this:

```
static PyObject *
py_method_min(PyObject *obj, PyObject *args) {
  if (not PyArg_ParseTuple(args, "")) {
    PyErr_SetString(PyExc_RuntimeError, "invalid parameters");
    return NULL;
  }
  return PyCapsule_New((void*)min_function, "numina.cmethod", NULL);
}
```

The string `"numina.cmethod"` is the name of the `PyCapsule`. It cannot be loadded unless it is the name expected by the C code.

The code to load it in a module is like this:

```
static PyMethodDef mymod_methods[] = {
 {"min_combine", (PyCFunction) py_method_min, METH_VARARGS, "Minimum method."},
 ...,
 { NULL, NULL, 0, NULL } /* sentinel */
};

PyMODINIT_FUNC
init_mymodule(void)
{
  PyObject *m;
  m = Py_InitModule("_mymodule", mymod_methods);
}
```

When compiled, this code created a file *_mymodule.so* that can be loaded by the Python interpreter. This module will contain, among others, a *min_combine* function.

```
>>> from _mymodule import min_combine
>>> method = min_combine()
...
>>> o = generic_combine(method, arrays)
```

A combine method with parameters

A combine method with parameters follow a similar approach. Let's say we want to implement a sigma-clipping method. We need to pass the function a *low* and a *high* rejection limits. Both numbers are real numbers greater than zero.

First, the Python function. I'm skipping error checking code hre.

```
static PyObject *
py_method_sigmaclip(PyObject *obj, PyObject *args) {
   double low = 0.0;
   double high = 0.0;
   PyObject *cap = NULL;

   if (!PyArg_ParseTuple(args, "dd", &low, &high)) {
      PyErr_SetString(PyExc_RuntimeError, "invalid parameters");
      return NULL;
   }

   cap = PyCapsule_New((void*) my_sigmaclip_function, "numina.cmethod", my_
↪destructor);

   /* Allocating space for the two parameters */
   /* We use Python memory allocator */
   double *funcdata = (double*)PyMem_Malloc(2 * sizeof(double));

   funcdata[0] = low;
   funcdata[1] = high;
   PyCapsule_SetContext(cap, funcdata);
   return cap;
}
```

Notice that in this case we construct the `PyCObject` using the same function than in the previouis case. The aditional data is stored as *Context*.

The deallocator is simply:

```
void my_destructor_function(PyObject* cap) {
   void* cdata = PyCapsule_GetContext(cap);
   PyMem_Free(cdata);
}
```

and the combine function is:

```
int my_sigmaclip_function(double *data, double *weights, size_t size, double *out[3],
         void *func_data) {

    double* fdata = (double*) func_data;
    double slow = *fdata;
```

```
    double shigh = *(fdata + 1);

    /* Operations go here */

    return 1;
}
```

Once the module is created and loaded, a sample session would be:

```
>>> from _mymodule import min_combine
>>> method = sigmaclip_combine(3.0, 3.0)
...
>>> o = generic_combine(method, arrays)
```

numina.array.cosmetics — Array cosmetics

numina.array.cosmetics.**ccdmask**(*flat1*, *flat2=None*, *mask=None*, *lowercut=6.0*, *uppercut=6.0*, *siglev=1.0*, *mode='region'*, *nmed=(7, 7)*, *nsig=(15, 15)*)

Find cosmetic defects in a detector using two flat field images.

Two arrays representing flat fields of different exposure times are required. Cosmetic defects are selected as points that deviate significantly of the expected normal distribution of pixels in the ratio between *flat2* and *flat1*. The median of the ratio is computed and subtracted. Then, the standard deviation is estimated computing the percentiles nearest to the pixel values corresponding to'siglev' in the normal CDF. The standard deviation is then the distance between the pixel values divided by two times *siglev*. The ratio image is then normalized with this standard deviation.

The behavior of the function depends on the value of the parameter *mode*. If the value is 'region' (the default), both the median and the sigma are computed in boxes. If the value is 'full', these values are computed using the full array.

The size of the boxes in 'region' mode is given by *nmed* for the median computation and *nsig* for the standard deviation.

The values in the normalized ratio array above *uppercut* are flagged as hot pixels, and those below '-lowercut' are flagged as dead pixels in the output mask.

Parameters

- **flat1** – an array representing a flat illuminated exposure.
- **flat2** – an array representing a flat illuminated exposure.
- **mask** – an integer array representing initial mask.
- **lowercut** – values below this sigma level are flagged as dead pixels.
- **uppercut** – values above this sigma level are flagged as hot pixels.
- **siglev** – level to estimate the standard deviation.
- **mode** – either 'full' or 'region'
- **nmed** – region used to compute the median
- **nsig** – region used to estimate the standard deviation

Returns the normalized ratio of the flats, the updated mask and standard deviation

Note: This function is based on the description of the task ccdmask of IRAF

See also:

cosmetics() Operates much like this function but computes median and sigma in the whole image instead of in boxes

numina.array.cosmetics.**cosmetics**(*flat1*, *flat2=None*, *mask=None*, *lowercut=6.0*, *uppercut=6.0*, *siglev=2.0*)

Find cosmetic defects in a detector using two flat field images.

Two arrays representing flat fields of different exposure times are required. Cosmetic defects are selected as points that deviate significantly of the expected normal distribution of pixels in the ratio between *flat2* and *flat1*.

The median of the ratio array is computed and subtracted to it.

The standard deviation of the distribution of pixels is computed obtaining the percentiles nearest the pixel values corresponding to *nsig* in the normal CDF. The standar deviation is then the distance between the pixel values divided by two times *nsig*. The ratio image is then normalized with this standard deviation.

The values in the ratio above *uppercut* are flagged as hot pixels, and those below '-lowercut' are flagged as dead pixels in the output mask.

Parameters

- **flat1** – an array representing a flat illuminated exposure.
- **flat2** – an array representing a flat illuminated exposure.
- **mask** – an integer array representing initial mask.
- **lowercut** – values bellow this sigma level are flagged as dead pixels.
- **uppercut** – values above this sigma level are flagged as hot pixels.
- **siglev** – level to estimate the standard deviation.

:returns:the updated mask

numina.array.fwhm — FWHM

FWHM calculation

numina.array.fwhm.**compute_fw_at_frac_max_1d_simple**(*Y*, *xc*, *X=None*, *f=0.5*)

Compute the full width at fraction f of the maximum

numina.array.fwhm.**compute_fwhm_1d_simple**(*Y*, *xc*, *X=None*)

Compute the FWHM.

numina.array.imsurfit — Image surface fitting

Least squares 2D image fitting to a polynomial.

numina.array.imsurfit.**imsurfit**(*data*, *order*, *output_fit=False*)

Fit a bidimensional polynomial to an image.

Parameters

- **data** – a bidimensional array

- **order** (*integer*) – order of the polynomial
- **output_fit** (*bool*) – return the fitted image

Returns a tuple with an array with the coefficients of the polynomial terms

```
>>> import numpy
>>> xx, yy = numpy.mgrid[-1:1:100j,-1:1:100j]
>>> z = 456.0 + 0.3 * xx - 0.9* yy
>>> imsurfit(z, order=1)
(array([  4.56000000e+02,   3.00000000e-01,  -9.00000000e-01]),)
```

numina.array.interpolation — Interpolation

A monotonic piecewise cubic interpolator.

class numina.array.interpolation.**SteffenInterpolator**(*x*, *y*, *yp_0=0.0*, *yp_N=0.0*, *extrapolate='raise'*, *fill_value=nan*)

A monotonic piecewise cubic 1-d interpolator.

A monotonic piecewise cubic interpolator based on Steffen, M., Astronomy & Astrophysics, 239, 443-450 (1990)

x and *y* are arrays of values used to approximate some function f: `y = f(x)`. This class returns an object whose call method uses monotonic cubic splines to find the value of new points.

Parameters

- **x** (*(N,), array_like*) – A 1-D array of real values, sorted monotonically increasing.
- **y** (*(N,), array_like*) – A 1-D array of real values.
- **yp_0** (*float. optional*) – The value of the derivative in the first sample.
- **yp_N** (*float, optional*) – The value of the derivative in the last sample.
- **extrapolate** (*str, optional*) – Specifies the kind of extrapolation as a string ('extrapolate', 'zeros', 'raise', 'const', 'border') If 'raise' is set, when interpolated values are requested outside of the domain of the input data (x,y), a ValueError is raised. If 'const' is set, 'fill_value' is returned. If 'zeros' is set, '0' is returned. If 'border' is set, 'y[0]' is returned for values below 'x[0]' and 'y[N-1]' is returned for values above 'x[N-1]' If 'extrapolate' is set, the extreme polynomial are extrapolated outside of their ranges. Default is 'raise'.
- **fill_value** (*float, optional*) – If provided, then this value will be used to fill in for requested points outside of the data range when 'extrapolation'is set to "const". If not provided, then the default is NaN.

numina.array.mode — Mode

Mode estimators.

numina.array.mode.**mode_half_sample**(*a*, *is_sorted=False*)

Estimate the mode using the Half Sample mode.

A method to estimate the mode, as described in D. R. Bickel and R. Frühwirth (contributed equally), "On a fast, robust estimator of the mode: Comparisons to other robust estimators with applications," Computational Statistics and Data Analysis 50, 3500-3530 (2006).

Example

>> import numpy as np >> np.random.seed(1392838) >> a = np.random.normal(1000, 200, size=1000) >> a[:100] = np.random.normal(2000, 300, size=100) >> b = np.sort(a) >> mode_half_sample(b, is_sorted=True) 1041.9327885039545

numina.array.mode.**mode_sex**(*a*)
Estimate the mode as sextractor

numina.array.nirproc — nIR preprocessing

numina.array.nirproc.**fowler_array**(*fowlerdata*, *ti=0.0*, *ts=0.0*, *gain=1.0*, *ron=1.0*, *badpixels=None*, *dtype='float64'*, *saturation=65631*, *blank=0*, *normalize=False*)
Loop over the first axis applying Fowler processing.

fowlerdata is assumed to be a 3D numpy.ndarray containing the result of a nIR observation in Fowler mode (Fowler and Gatley 1991). The shape of the array must be of the form 2N_p x M x N, with N_p being the number of pairs in Fowler mode.

The output signal is just the mean value of the differences between the last N_p values (S_i) and the first N_p values (R-i).

$$S_F = \frac{1}{N_p} \sum_{i=0}^{N_p-1} S_i - R_i$$

If the source has a radiance F, then the measured signal is equivalent to:

$$S_F = FT_I - FT_S(N_p - 1) = FT_E$$

being T_I the integration time (*ti*), the time since the first productive read to the last productive read for a given pixel and T_S the time between samples (*ts*). T_E is the time between correlated reads $T_E = T_I - T_S(N_p - 1)$.

The variance of the signnal is the sum of two terms, one for the readout noise:

$$\mathrm{var}(S_{F1}) = \frac{2\sigma_R^2}{N_p}$$

and other for the photon noise:

$$\mathrm{var}(S_{F2}) = FT_E - FT_S\frac{1}{3}(N_p - \frac{1}{N_p}) = FT_I - FT_S(\frac{4}{3}N_p - 1 - \frac{1}{3N_p})$$

Parameters

- **fowlerdata** – Convertible to a 3D numpy.ndarray with first axis even
- **ti** – Integration time.
- **ts** – Time between samples.
- **gain** – Detector gain.
- **ron** – Detector readout noise in counts.
- **badpixels** – An optional MxN mask of dtype 'uint8'.
- **dtype** – The dtype of the float outputs.
- **saturation** – The saturation level of the detector.

- **blank** – Invalid values in output are substituted by *blank*.

Returns A tuple of (signal, variance of the signal, numper of pixels used and badpixel mask.

Raises ValueError

numina.array.nirproc.**ramp_array**(*rampdata*, *ti*, *gain=1.0*, *ron=1.0*, *badpixels=None*, *dtype='float64'*, *saturation=65631*, *blank=0*, *nsig=None*, *normalize=False*)

Loop over the first axis applying ramp processing.

rampdata is assumed to be a 3D numpy.ndarray containing the result of a nIR observation in folow-up-the-ramp mode. The shape of the array must be of the form N_s x M x N, with N_s being the number of samples.

Parameters

- **fowlerdata** – Convertible to a 3D numpy.ndarray
- **ti** – Integration time.
- **gain** – Detector gain.
- **ron** – Detector readout noise in counts.
- **badpixels** – An optional MxN mask of dtype 'uint8'.
- **dtype** – The dtype of the float outputs.
- **saturation** – The saturation level of the detector.
- **blank** – Invalid values in output are substituted by *blank*.

Returns A tuple of signal, variance of the signal, numper of pixels used and badpixel mask.

Raises ValueError

numina.array.offrot — Offset and Rotation

Fit offset and rotation

numina.array.offrot.**fit_offset_and_rotation**(*coords0*, *coords1*)

Fit a rotation and a traslation between two sets points.

Fit a rotation matrix and a traslation bewtween two matched sets consisting of M N-dimensional points

Parameters

- **coords0** (*(M, N) array_like*) –
- **coords1** (*(M, N) array_lke*) –

Returns

- **offset** (*(N,) array_like*)
- **rotation** (*(N, N) array_like*)

Notes

Fit offset and rotation using Kabsch's algorithm[1]_[2]

[2] Also here: http://nghiaho.com/?page_id=671

numina.array.peaks — Peak finding

numina.array.recenter — Recenter

Recenter routines

numina.array.recenter.**centering_centroid**(*data*, *xi*, *yi*, *box*, *nloop=10*, *toldist=0.001*, *maxdist=10.0*)

returns x, y, background, status, message

status is:

- 0: not recentering
- 1: recentering successful
- 2: maximum distance reached
- 3: not converged

numina.array.robusfit — Robust fits

Robust fits

numina.array.robustfit.**fit_theil_sen**(*x*, *y*)

Compute a robust linear fit using the Theil-Sen method.

See http://en.wikipedia.org/wiki/Theil%E2%80%93Sen_estimator for details. This function "pairs up sample points by the rank of their x-coordinates (the point with the smallest coordinate being paired with the first point above the median coordinate, etc.) and computes the median of the slopes of the lines determined by these pairs of points".

Parameters

- **x** (*array_like, shape (M,)*) – X coordinate array.
- **y** (*array_like, shape (M,) or (M,K)*) – Y coordinate array. If the array is two dimensional, each column of the array is independently fitted sharing the same x-coordinates. In this last case, the returned intercepts and slopes are also 1d numpy arrays.

Returns **coef** – Intercept and slope of the linear fit. If y was 2-D, the coefficients in column k of coef represent the linear fit to the data in y's k-th column.

Return type ndarray, shape (2,) or (2, K)

Raises ValueError: – If the number of points to fit is < 5

numina.array.stats —

numina.array.stats.**robust_std**(*x*, *debug=False*)

Compute a robust estimator of the standard deviation

See Eq. 3.36 (page 84) in Statistics, Data Mining, and Machine in Astronomy, by Ivezic, Connolly, VanderPlas & Gray

Parameters

- **x** (*1d numpy array, float*) – Array of input values which standard deviation is requested.
- **debug** (*bool*) – If True prints computed values

Returns **sigmag** – Robust estimator of the standar deviation

Return type float

numina.array.stats.**summary**(*x*, *rm_nan=False*, *debug=False*)

Compute basic statistical parameters.

Parameters

- **x** (*1d numpy array, float*) – Input array with values which statistical properties are requested.
- **rm_nan** (*bool*) – If True, filter out NaN values before computing statistics.
- **debug** (*bool*) – If True prints computed values.

Returns **result** – Minimum, percentile 25, percentile 50 (median), mean, percentile 75, maximum, standard deviation, robust standard deviation, percentile 15.866 (equivalent to -1 sigma in a normal distribution) and percentile 84.134 (+1 sigma).

Return type dictionary of floats

numina.array.trace — Spectrum tracing

numina.array.wavecalib — Wavelength calibration

Automatic identification of lines and wavelength calibration

numina.array.wavecalib.arccalibration.**arccalibration**(*wv_master*, *xpos_arc*, *naxis1_arc*, *crpix1*, *wv_ini_search*, *wv_end_search*, *error_xpos_arc*, *times_sigma_r*, *frac_triplets_for_sum*, *times_sigma_theil_sen*, *poly_degree_wfit*, *times_sigma_polfilt*, *times_sigma_cook*, *times_sigma_inclusion*, *debugplot=0*)

Performs arc line identification for arc calibration.

This function is a wrapper of two functions, which are responsible of computing all the relevant information concerning the triplets generated from the master table and the actual identification procedure of the arc lines, respectively.

The separation of those computations in two different functions helps to avoid the repetition of calls to the first function when calibrating several arcs using the same master table.

Parameters

- **wv_master** (*1d numpy array, float*) – Array with wavelengths corresponding to the master table (Angstroms).
- **xpos_arc** (*1d numpy array, float*) – Location of arc lines (pixels).
- **naxis1_arc** (*int*) – NAXIS1 for arc spectrum.
- **crpix1** (*float*) – CRPIX1 value to be employed in the wavelength calibration.
- **wv_ini_search** (*float*) – Minimum valid wavelength.

- **wv_end_search** (*float*) – Maximum valid wavelength.
- **error_xpos_arc** (*float*) – Error in arc line position (pixels).
- **times_sigma_r** (*float*) – Times sigma to search for valid line position ratios.
- **frac_triplets_for_sum** (*float*) – Fraction of distances to different triplets to sum when computing the cost function.
- **times_sigma_theil_sen** (*float*) – Number of times the (robust) standard deviation around the linear fit (using the Theil-Sen method) to reject points.
- **poly_degree_wfit** (*int*) – Degree for polynomial fit to wavelength calibration.
- **times_sigma_polfilt** (*float*) – Number of times the (robust) standard deviation around the polynomial fit to reject points.
- **times_sigma_cook** (*float*) – Number of times the standard deviation of Cook's distances to detect outliers. If zero, this method of outlier detection is ignored.
- **times_sigma_inclusion** (*float*) – Number of times the (robust) standard deviation around the polynomial fit to include a new line in the set of identified lines.
- **debugplot** (*int*) – Determines whether intermediate computations and/or plots are displayed: 00 : no debug, no plots 01 : no debug, plots without pauses 02 : no debug, plots with pauses 10 : debug, no plots 11 : debug, plots without pauses 12 : debug, plots with pauses

Returns **list_of_wvfeatures** – A list of size equal to the number of identified lines, which elements are instances of the class WavecalFeature, containing all the relevant information concerning the line identification.

Return type list (of WavecalFeature instances)

numina.array.wavecalib.arccalibration.**arccalibration_direct**(*wv_master*, *ntriplets_master*, *ratios_master_sorted*, *triplets_master_sorted_list*, *xpos_arc*, *naxis1_arc*, *crpix1*, *wv_ini_search*, *wv_end_search*, *error_xpos_arc=1.0*, *times_sigma_r=3.0*, *frac_triplets_for_sum=0.5*, *times_sigma_theil_sen=10.0*, *poly_degree_wfit=3*, *times_sigma_polfilt=10.0*, *times_sigma_cook=10.0*, *times_sigma_inclusion=5.0*, *debugplot=0*)

Performs line identification for arc calibration using line triplets.

This function assumes that a previous call to the function responsible for the computation of information related to the triplets derived from the master table has been previously executed.

Parameters

- **wv_master** (*1d numpy array, float*) – Array with wavelengths corresponding to the master table (Angstroms).
- **ntriplets_master** (*int*) – Number of triplets built from master table.

- **ratios_master_sorted** (*1d numpy array, float*) – Array with values of the relative position of the central line of each triplet, sorted in ascending order.
- **triplets_master_sorted_list** (*list of tuples*) – List with tuples of three numbers, corresponding to the three line indices in the master table. The list is sorted to be in correspondence with *ratios_master_sorted*.
- **xpos_arc** (*1d numpy array, float*) – Location of arc lines (pixels).
- **naxis1_arc** (*int*) – NAXIS1 for arc spectrum.
- **crpix1** (*float*) – CRPIX1 value to be employed in the wavelength calibration.
- **wv_ini_search** (*float*) – Minimum valid wavelength.
- **wv_end_search** (*float*) – Maximum valid wavelength.
- **error_xpos_arc** (*float*) – Error in arc line position (pixels).
- **times_sigma_r** (*float*) – Times sigma to search for valid line position ratios.
- **frac_triplets_for_sum** (*float*) – Fraction of distances to different triplets to sum when computing the cost function.
- **times_sigma_theil_sen** (*float*) – Number of times the (robust) standard deviation around the linear fit (using the Theil-Sen method) to reject points.
- **poly_degree_wfit** (*int*) – Degree for polynomial fit to wavelength calibration.
- **times_sigma_polfilt** (*float*) – Number of times the (robust) standard deviation around the polynomial fit to reject points.
- **times_sigma_cook** (*float*) – Number of times the standard deviation of Cook's distances to detect outliers. If zero, this method of outlier detection is ignored.
- **times_sigma_inclusion** (*float*) – Number of times the (robust) standard deviation around the polynomial fit to include a new line in the set of identified lines.
- **debugplot** (*int*) – Determines whether intermediate computations and/or plots are displayed: 00 : no debug, no plots 01 : no debug, plots without pauses 02 : no debug, plots with pauses 10 : debug, no plots 11 : debug, plots without pauses 12 : debug, plots with pauses 21 : debug, many plots without pauses 22 : debug, many plots with pauses

Returns **list_of_wvfeatures** – A list of size equal to the number of identified lines, which elements are instances of the class WavecalFeature, containing all the relevant information concerning the line identification.

Return type list (of WavecalFeature instances)

numina.array.wavecalib.arccalibration.**fit_list_of_wvfeatures**(*list_of_wvfeatures*, *naxis1_arc*, *crpix1*, *poly_degree_wfit*, *weighted=False*, *debugplot=0*, *plot_title=None*)

Fit polynomial to arc calibration list_of_wvfeatures.

Parameters

- **list_of_wvfeatures** (*list (of WavecalFeature instances)*) – A list of size equal to the number of identified lines, which elements are instances of the class WavecalFeature, containing all the relevant information concerning the line identification.
- **naxis1_arc** (*int*) – NAXIS1 of arc spectrum.

- **crpix1** (*float*) – CRPIX1 value to be employed in the wavelength calibration.
- **poly_degree_wfit** (*int*) – Polynomial degree corresponding to the wavelength calibration function to be fitted.
- **weighted** (*bool*) – Determines whether the polynomial fit is weighted or not, using as weights the values of the cost function obtained in the line identification. Since the weights can be very different, typically weighted fits are not good because, in practice, they totally ignore the points with the smallest weights (which, in the other hand, are useful when handling the borders of the wavelength calibration range).
- **debugplot** (*int*) – Determines whether intermediate computations and/or plots are displayed: 00 : no debug, no plots 01 : no debug, plots without pauses 02 : no debug, plots with pauses 10 : debug, no plots 11 : debug, plots without pauses 12 : debug, plots with pauses
- **plot_title** (*string or None*) – Title for residuals plot.

Returns **solution_wv** – Instance of class SolutionArcCalibration, containing the information concerning the arc lines that have been properly identified. The information about all the lines (including those initially found but at the end discarded) is stored in the list of WavecalFeature instances 'list_of_wvfeatures'.

Return type SolutionArcCalibration instance

numina.array.wavecalib.arccalibration.**gen_triplets_master**(*wv_master*, *debugplot=0*)

Compute information associated to triplets in master table.

Determine all the possible triplets that can be generated from the array *wv_master*. In addition, the relative position of the central line of each triplet is also computed.

Parameters

- **wv_master** (*1d numpy array, float*) – Array with wavelengths corresponding to the master table (Angstroms).
- **debugplot** (*int*) – Determines whether intermediate computations and/or plots are displayed: 00 : no debug, no plots 01 : no debug, plots without pauses 02 : no debug, plots with pauses 10 : debug, no plots 11 : debug, plots without pauses 12 : debug, plots with pauses

Returns

- **ntriplets_master** (*int*) – Number of triplets built from master table.
- **ratios_master_sorted** (*1d numpy array, float*) – Array with values of the relative position of the central line of each triplet, sorted in ascending order.
- **triplets_master_sorted_list** (*list of tuples*) – List with tuples of three numbers, corresponding to the three line indices in the master table. The list is sorted to be in correspondence with *ratios_master_sorted*.

numina.array.wavecalib.arccalibration.**select_data_for_fit**(*list_of_wvfeatures*)

Select information from valid arc lines to facilitate posterior fits.

Parameters **list_of_wvfeatures** (*list (of WavecalFeature instances)*) – A list of size equal to the number of identified lines, which elements are instances of the class WavecalFeature, containing all the relevant information concerning the line identification.

Returns

- **nfit** (*int*) – Number of valid points for posterior fits.

- **ifit** (*list of int*) – List of indices corresponding to the arc lines which coordinates are going to be employed in the posterior fits.
- **xfit** (*1d numpy aray*) – X coordinate of points for posterior fits.
- **yfit** (*1d numpy array*) – Y coordinate of points for posterior fits.
- **wfit** (*1d numpy array*) – Cost function of points for posterior fits. The inverse of these values can be employed for weighted fits.

numina.array.wavecalib.peaks_spectrum.**find_peaks_spectrum**(*sx*, *nwinwidth*, *threshold=0*, *debugplot=0*)

Find peaks in array.

The algorithm imposes that the signal at both sides of the peak decreases monotonically.

Parameters

- **sx** (*1d numpy array, floats*) – Input array.
- **nwinwidth** (*int*) – Width of the window where each peak must be found.
- **threshold** (*float*) – Minimum signal in the peaks.
- **debugplot** (*int*) – Determines whether intermediate computations and/or plots are displayed: 00 : no debug, no plots 01 : no debug, plots without pauses 02 : no debug, plots with pauses 10 : debug, no plots 11 : debug, plots without pauses 12 : debug, plots with pauses

Returns **ixpeaks** – Peak locations, in array coordinates (integers).

Return type 1d numpy array, int

numina.array.wavecalib.peaks_spectrum.**refine_peaks_spectrum**(*sx*, *ixpeaks*, *nwinwidth*, *method=None*, *debugplot=0*)

Refine line peaks in spectrum.

Parameters

- **sx** (*1d numpy array, floats*) – Input array.
- **ixpeaks** (*1d numpy array, int*) – Initial peak locations, in array coordinates (integers). These values can be the output from the function find_peaks_spectrum().
- **nwinwidth** (*int*) – Width of the window where each peak must be refined.
- **method** (*string*) – "poly2" : fit to a 2nd order polynomial "gaussian" : fit to a Gaussian
- **debugplot** (*int*) – Determines whether intermediate computations and/or plots are displayed: 00 : no debug, no plots 01 : no debug, plots without pauses 02 : no debug, plots with pauses 10 : debug, no plots 11 : debug, plots without pauses 12 : debug, plots with pauses

Returns

- **fxpeaks** (*1d numpy array, float*) – Refined peak locations, in array coordinates.
- **sxpeaks** (*1d numpy array, float*) – When fitting Gaussians, this array stores the fitted line widths (sigma). Otherwise, this array returns zeros.

Store the solution of a wavelength calibration

class numina.array.wavecalib.solutionarc.**CrLinear**(*crpix*, *crval*, *crmin*, *crmax*, *cdelt*)

Store information concerning the linear wavelength calibration.

Parameters

- **crpix** (*float*) – CRPIX1 value employed in the linear wavelength calibration.
- **crval** (*float*) – CRVAL1 value corresponding tot he linear wavelength calibration.
- **crmin** (*float*) – CRVAL value at pixel number 1 corresponding to the linear wavelength calibration.
- **crmax** (*float*) – CRVAL value at pixel number NAXIS1 corresponding to the linear wavelength calibration.
- **cdelt** (*float*) – CDELT1 value corresponding to the linear wavelength calibration.

Identical to parameters.

class numina.array.wavecalib.solutionarc.**SolutionArcCalibration**(*features*, *coeff*, *residual_std*, *cr_linear*)

Auxiliary class to store the arc calibration solution.

Note that this class only stores the information concerning the arc lines that have been properly identified. The information about all the lines (including those initially found but at the end discarded) is stored in the list of WavecalFeature instances.

Parameters

- **features** (*list (of WavecalFeature instances)*) – A list of size equal to the number of identified lines, which elements are instances of the class WavecalFeature, containing all the relevant information concerning the line identification.
- **coeff** (*1d numpy array (float)*) – Coefficients of the wavelength calibration polynomial.
- **residual_std** (*float*) – Residual standard deviation of the fit.
- **cr_linear** (*instance of CrLinear*) – Object containing the linear approximation parameters crpix, crval, cdelt, crmin and crmax.

Identical to parameters.

update_features(*poly*)

Evaluate wavelength at xpos using the provided polynomial.

class numina.array.wavecalib.solutionarc.**WavecalFeature**(*line_ok*, *category*, *lineid*, *funcost*, *xpos*, *ypos=0.0*, *peak=0.0*, *fwhm=0.0*, *reference=0.0*, *wavelength=0.0*)

Store information concerning a particular line identification.

Parameters

- **line_ok** (*bool*) – True if the line has been properly identified.
- **category** (*char*) – Line identification type (A, B, C, D, E, R, T, P, K, I, X). See documentation embedded within the arccalibration_direct function for details.
- **lineid** (*int*) – Number of identified line within the master list.
- **xpos** (*float*) – Pixel x-coordinate of the peak of the line.
- **ypos** (*float*) – Pixel y-coordinate of the peak of the line.
- **peak** (*float*) – Flux of the peak of the line.
- **fwhm** (*float*) – FWHM of the line.

- **reference** (*float*) – Wavelength of the identified line in the master list.
- **wavelength** (*float*) – Wavelength of the identified line estimated from the wavelength calibration polynomial.
- **funcost** (*float*) – Cost function corresponding to each identified arc line.

Identical to parameters.

numina.array.utils —

Utility routines

numina.array.utils.**coor_to_pix_1d**(*w*)
Return the pixel where a coordinate is located.

numina.array.utils.**expand_region**(*tuple_of_s*, *a*, *b*, *start=0*, *stop=None*)
Apply expend_slice on a tuple of slices

numina.array.utils.**expand_slice**(*s*, *a*, *b*, *start=0*, *stop=None*)
Expand a slice on the start/stop limits

numina.array.utils.**image_box**(*center*, *shape*, *box*)
Create a region of size box, around a center in a image of shape.

numina.array.utils.**slice_create**(*center*, *block*, *start=0*, *stop=None*)
Return an slice with a symmetric region around center.

numina.core — Core classes for Pipelines

numina.core.dataframe —

Basic Data Products

class numina.core.dataframe.**DataFrame**(*frame=None*, *filename=None*)
A handle to a image in disk or in memory.

numina.core.dataholders —

Recipe requirements

class numina.core.dataholders.**Product**(*ptype*, *description=''*, *validation=True*, *destination=None*, *optional=False*, *default=None*, *choices=None*)
Product holder for RecipeResult.

numina.core.metaclass —

Base metaclasses

class numina.core.metaclass.**RecipeInputType**
Metaclass for RecipeInput.

class numina.core.metaclass.**RecipeResultType**
Metaclass for RecipeResult.

class numina.core.metaclass.**StoreType**
Metaclass for storing members.

numina.core.metarecipes —

Metaclasses for Recipes.

class numina.core.metarecipes.**RecipeType**
Metaclass for Recipe.

numina.core.metarecipes.**generate_docs**(*klass*)
Add documentation to generated classes

numina.core.objimport —

Import objects by name

numina.core.objimport.**import_object**(*path*)
Import an object given its fully qualified name.

numina.core.oresult — Observation Result

Results of the Observing Blocks

class numina.core.oresult.**ObservationResult**(*instrument='UNKNOWN'*, *mode='UNKNOWN'*)
The result of a observing block.

numina.core.oresult.**dataframe_from_list**(*values*)
Build a DataFrame object from a list.

numina.core.oresult.**obsres_from_dict**(*values*)
Build a ObservationResult object from a dictionary.

numina.core.pipeline —

DRP related classes

class numina.core.pipeline.**InstrumentDRP**(*name*, *configurations*, *modes*, *pipelines*, *products=None*)
Description of an Instrument Data Reduction Pipeline

iterate_mode_provides(*modes*, *pipeline*)
Return the mode that provides a given product

query_provides(*product*, *pipeline='default'*, *search=False*)
Return the mode that provides a given product

search_mode_provides(*product*, *pipeline='default'*)
Search the mode that provides a given product

class numina.core.pipeline.**ObservingMode**
Observing modes of an Instrument.

class numina.core.pipeline.**Pipeline**(*instrument*, *name*, *recipes*, *version=1*, *products=None*, *provides=None*)
Base class for pipelines.

depsolve()
Load all recipes to search for products

get_recipe_object(*mode*)
Load recipe object, according to observing mode

load_product_class(*mode*)
Load recipe object, according to observing mode

load_product_object(*name*)
Load product object, according to name

load_recipe_object(*mode*)
Load recipe object, according to observing mode

provides(*mode_label*)
Return the ProductEntry for some mode

query_recipe(*mode*)
Recursive query of all calibrations required by a mode

who_provides(*product_label*)
Return the ProductEntry for some requirement

numina.core.pipelineload —

Build a LoadableDRP from a yaml file

numina.core.pipelineload.**check_section**(*node*, *section*, *keys=None*)
Validate keys in a section

numina.core.pipelineload.**drp_load**(*package*, *resource*, *confclass=None*)
Load the DRPS from a resource file.

numina.core.pipelineload.**drp_load_data**(*package*, *data*, *confclass=None*)
Load the DRPS from data.

numina.core.pipelineload.**load_mode**(*node*)
Load one observing mdode

numina.core.pipelineload.**load_mode_validator**(*obs_mode*, *node*)
Load observing mode validator

numina.core.pipelineload.**load_modes**(*node*)
Load all observing modes

numina.core.products —

class numina.core.products.**ArrayType**(*default=None*)
A type of array.

class numina.core.products.**ConfigurationTag**
A type that is part of the instrument configuration.

class numina.core.products.**DataFrameType**
A type of DataFrame.

convert(*obj*)
Convert

validate(*value*)

class numina.core.products.**DataProductTag**
A type that is a data product.

name()
Unique name of the datatype

class numina.core.products.**InstrumentConfigurationType**
The type of InstrumentConfiguration.

class numina.core.products.**ObservationResultType**(*rawtype=None*)
The type of ObservationResult.

numina.core.recipeinout —

Recipe inpus and outputs

class numina.core.recipeinout.**ErrorRecipeResult**(*errortype*, *message*, *traceback*, *_error='‘*)
The error result of a Recipe.

class numina.core.recipeinout.**RecipeInput**(**args*, ***kwds*)
RecipeInput base class

class numina.core.recipeinout.**RecipeResult**(**args*, ***kwds*)
The result of a Recipe.

class numina.core.recipeinout.**define_input**(*inputClass*)
Recipe decorator.

numina.core.recipeinout.**define_requirements**
alias of *define_input*

class numina.core.recipeinout.**define_result**(*resultClass*)
Recipe decorator.

numina.core.recipes —

Basic tools and classes used to generate recipe modules.

A recipe is a class that complies with the *reduction recipe API*:

- The class must derive from numina.core.BaseRecipe.

class numina.core.recipes.**BaseRecipe**(**args*, ***kwargs*)
Base class for all instrument recipes

class RecipeInput(**args*, ***kwds*)
RecipeInput base class

class BaseRecipe.**RecipeResult**(**args*, ***kwds*)
The result of a Recipe.

BaseRecipe.**build_recipe_input**(*ob*, *dal*)
Build a RecipeInput object.

classmethod BaseRecipe.**create_input**(**args*, ***kwds*)
Pass the result arguments to the RecipeInput constructor

classmethod BaseRecipe.**create_result**(**args*, ***kwds*)
Pass the result arguments to the RecipeResult constructor

BaseRecipe.**run_qc**(*recipe_input*, *recipe_result*)
Run Quality Control checks.

BaseRecipe.**set_base_headers**(*hdr*)
Set metadata in FITS headers.

BaseRecipe.**validate_input**(*recipe_input*)
"Validate the input of the recipe

BaseRecipe.**validate_result**(*recipe_result*)
Validate the result of the recipe

numina.core.requirements —

Recipe requirement holders

class numina.core.requirements.**InstrumentConfigurationRequirement**
The Recipe requires the configuration of the instrument.

class numina.core.requirements.**ObservationResultRequirement**
The Recipe requires the result of an observation.

class numina.core.requirements.**Parameter**(*value*, *description*, *destination=None*, *optional=False*, *choices=None*, *validation=True*)
The Recipe requires a plain Python type.

class numina.core.requirements.**Requirement**(*rtype*, *description*, *destination=None*, *optional=False*, *default=None*, *choices=None*, *validation=True*)
Requirement holder holder for RecipeRequirement.

numina.core.taggers —

Function to retrieve tags from Observation results.

numina.core.taggers.**get_tags_from_full_ob**(*ob*, *reqtags=None*)

Parameters

- **(ObservationResult)** (*ob*) –
- **(dict)** (*reqtags*) –

Returns

Return type A dictionary

numina.core.types —

class numina.core.types.**AutoDataType**
Data type for types that are its own python type

class numina.core.types.**DataType**(*ptype*, *default=None*)
Base class for input/output types of recipes.

convert(*obj*)
Basic conversion to internal type

This method is intended to be redefined by subclasses

convert_in(*obj*)
Basic conversion to internal type of inputs.

This method is intended to be redefined by subclasses

convert_out(*obj*)
Basic conversion to internal type of outputs.

This method is intended to be redefined by subclasses

classmethod isproduct()
Check if the DataType is the product of a Recipe

name()
Unique name of the datatype

validate(*obj*)
Validate convertibility to internal representation

Returns True if 'obj' matches the data type

Return type bool

Raises ValidationError – If the validation fails

class numina.core.types.**ListOfType**(*ref*, *index=0*)
Data type for lists of other types.

class numina.core.types.**NullType**
Data type for None.

class numina.core.types.**PlainPythonType**(*ref=None*)
Data type for Python basic types.

numina.core.utils —

Recipes for system checks.

class numina.core.utils.**AlwaysFailRecipe**(**args*, ***kwargs*)
A Recipe that always fails.

class RecipeInput(**args*, ***kwds*)
RecipeInput base class

class AlwaysFailRecipe.**RecipeResult**(**args*, ***kwds*)
The result of a Recipe.

class numina.core.utils.**AlwaysSuccessRecipe**(**args*, ***kwargs*)
A Recipe that always successes.

class RecipeInput(**args*, ***kwds*)
RecipeInput base class

class AlwaysSuccessRecipe.**RecipeResult**(**args*, ***kwds*)
The result of a Recipe.

class numina.core.utils.**OBSuccessRecipe**(**args*, ***kwargs*)
A Recipe that always successes, it requires an OB

class OBSuccessRecipeInput(**args*, ***kwds*)
OBSuccessRecipeInput documentation.

obresult
ObservationResultType, requirement – Observation Result

OBSuccessRecipe.**RecipeInput**
alias of *OBSuccessRecipeInput*

class OBSuccessRecipe.**RecipeResult**(**args*, ***kwds*)
The result of a Recipe.

numina.core.validator —

Validator decorator

numina.core.validator.**validate**(*method*)
Decorate run method, inputs and outputs are validated

numina.flow —

exception numina.flow.**FlowError**
Error base class for flows.

class numina.flow.**ParallelFlow**(*nodeseq*)
A flow where Nodes are executed in parallel.

class numina.flow.**SerialFlow**(*nodeseq*)
A flow where Nodes are executed sequentially.

class numina.flow.node.**AdaptorNode**(*work*, *ninputs=1*, *noutputs=1*)
A *Node* that runs a function.

class numina.flow.node.**IdNode**
A Node that returns its inputs.

class numina.flow.node.**Node**(*ninputs=1*, *noutputs=1*)
An elemental operation in a Flow.

class numina.flow.node.**OutputSelector**(*ninputs*, *indexes*)
A Node that returns part of the results.

class numina.flow.processing.**BadPixelCorrector**(*badpixelmask*, *datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that corrects a frame from bad pixels.

class numina.flow.processing.**BiasCorrector**(*biasmap*, *biasvar=None*, *datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that corrects a frame from bias.

class numina.flow.processing.**Corrector**(*datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that corrects a frame from instrumental signatures.

class numina.flow.processing.**DarkCorrector**(*darkmap*, *darkvar=None*, *datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that corrects a frame from dark current.

class numina.flow.processing.**DivideByExposure**(*datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that divides its input by exposure time.

class numina.flow.processing.**FlatFieldCorrector**(*flatdata*, *datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that corrects a frame from flat-field.

class numina.flow.processing.**NonLinearityCorrector**(*polynomial*, *datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that corrects a frame from non-linearity.

class numina.flow.processing.**SimpleDataModel**
Model of the Data being processed

class numina.flow.processing.**SkyCorrector**(*skydata*, *datamodel=None*, *calibid='calibid-unknown'*, *dtype='float32'*)
A Node that corrects a frame from sky.

numina.flow.processing.**TagOptionalCorrector**
alias of *Corrector*

numina.exceptions — Numina exceptions

Exceptions for the numina package.

exception numina.exceptions.**DetectorElapseError**
Error in the clocking of a Detector.

exception numina.exceptions.**DetectorReadoutError**
Error in the readout of a Detector.

exception numina.exceptions.**Error**
Base class for exceptions in the numina package.

exception numina.exceptions.**NoResultFound**
No result found in a DAL query.

numina.exceptions.**NoResultFoundOrig**
alias of *NoResultFound*

exception numina.exceptions.**RecipeError**
A non recoverable problem during recipe execution.

exception numina.exceptions.**ValidationError**
Error during validation of Recipe inputs and outputs.

numina.frame — Frame manipulation

numina.logger —

Extra logging handlers for the numina logging system.

class numina.logger.**FITSHistoryHandler**(*header*)
Logging handler using HISTORY FITS cards

numina.logger.**log_to_history**(*logger*, *name*)
Decorate function, adding a logger handler stored in FITS.

numina.core.qc — Quality Control for Numina

Quality control for Numina-based applications.

This moudle defines functions and classes which implement quality asses for Numina-based applications.

QA Levels

The numeric values of the QC levels are given in this table.

Level	Numeric value
GOOD	100
FAIR	90
BAD	70

numina.treedict —

An implementation of hierarchical dictionary.

numina.user — CLI interface

User command line interface of Numina.

numina.util —

numina.user.xdgdirs —

Implementation of some of freedesktop.org Base Directories.

The directories are defined here:

http://standards.freedesktop.org/basedir-spec/

We only require xdg_data_dirs and xdg_config_home

Indices and tables

- genindex
- modindex
- search
- *Glossary*

CHAPTER 4

Glossary

DFP Data Factory Pipeline

DRP Data Reduction Pipeline

observing mode One of the prescribed ways of observing with an instrument

recipe A software object that processes the data obtained with a given observing mode of the instrument

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z