Numina Documentation

Release 0.15.dev0

Sergio Pascual, Nicolás Cardiel, Pablo Picazo-Sánchez

April 01, 2016

Contents

Welcome. This is the Documentation for Numina (version 0.15, date April 01, 2016),

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.

1.1 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.

1.1.1 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

1.1.2 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
```

1.1.3 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
```

1.1.4 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*

1.2 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.

1.2.1 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.

1.2.2 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
```

1.2.3 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
$
```

1.2.4 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.

1.3 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.

1.3.1 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
```

1.3.2 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
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`).

1.3.3 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.

1.3.4 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.

1.3.5 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.

1.3.6 Numina Installation

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

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

1.4 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.

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

1.4.1 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.

1.4.2 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.

1.4.3 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.

1.4.4 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.

2.1 Numina Pipeline Concepts

2.1.1 Instrument

2.1.2 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:

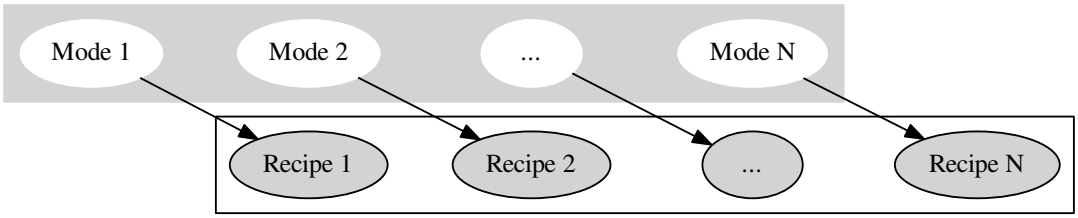
- The configuration of the Telescope
- The configuration of the Instrument
- The type of processing required by the images obtained durinf the observation

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.

2.1.3 Recipes

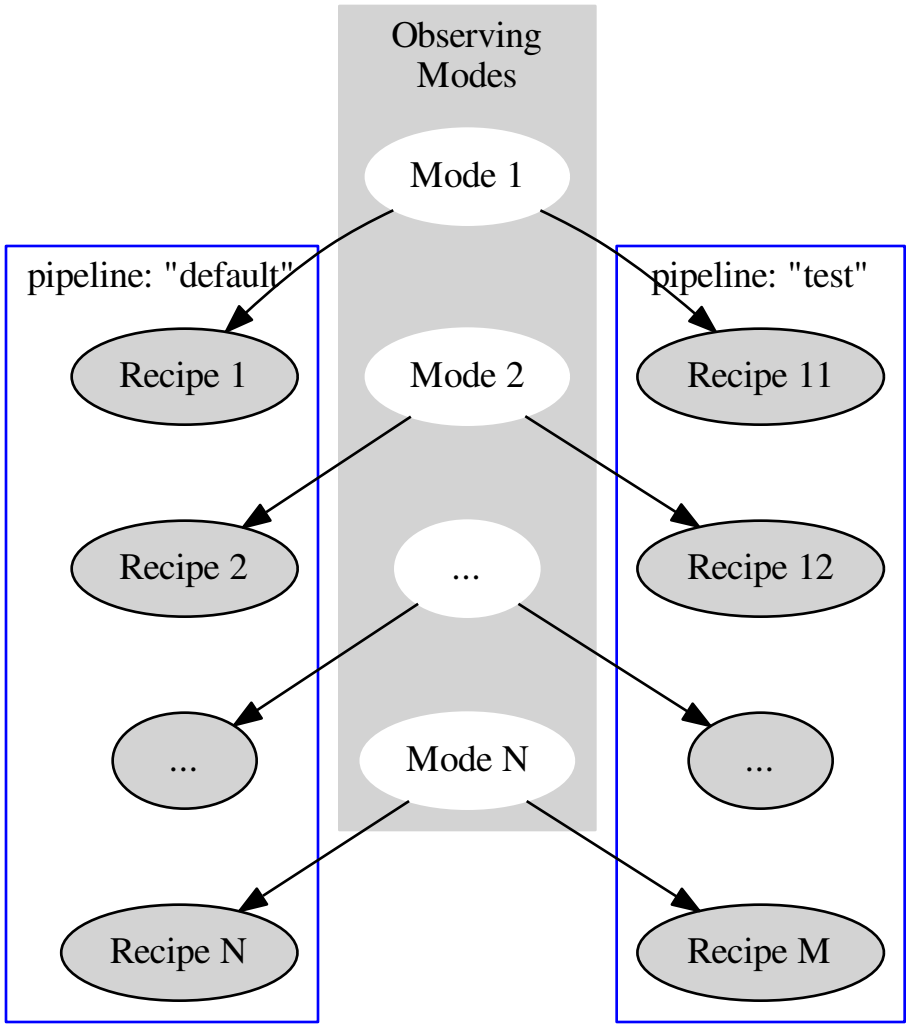
A recipe is a method to process the images obtained in a particular observing mode. Recipes in general require (as inputs) the list of raw images obtained during the observation. Recipes can require other inputs (calibrations), and those inputs can be the outputs of other recipes.

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



2.1.4 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.



2.1.5 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.

2.1.6 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: 12
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
```

2.1.7 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.3 DRP Data Types

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

2.3.1 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)
```

2.3.2 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.

2.3.3 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 April 01, 2016

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.

3.1 Numina modules

3.1.1 `numina.array` — Array manipulation

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

3.1.2 numina.array.background — Background estimation

3.1.3 numina.array.blocks — Generation of blocks

3.1.4 numina.array.combine — Array combination

3.1.5 Combination methods in numina.array.combine

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

mean_method()
Mean method

median_method()
Median method

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

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

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

3.1.6 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)
```

3.1.7 numina.array.cosmetics — Array cosmetics

3.1.8 numina.array.imsurfit — Image surface fitting

3.1.9 numina.array.nirproc — nIR preprocessing

3.1.10 numina.core — Core classes for Pipelines

3.1.11 numina.core.dataframe —

3.1.12 numina.core.oresult — Observation Result

3.1.13 numina.core.pipeline —

3.1.14 numina.core.pipelineload —

3.1.15 numina.core.products —

3.1.16 numina.core.recipeinout —

3.1.17 numina.core.recipes —

3.1.18 numina.core.dataholders —

3.1.19 numina.core.requirements —

3.1.20 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.

3.1.21 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.

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

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

3.1.22 numina.frame — Frame manipulation

3.1.23 numina.logger —

3.1.24 numina.core.qc — Quality Control for Numina

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

3.1.25 numina.treedict —

An implementation of hierarchical dictionary.

3.1.26 `numina.user` — CLI interface

User command line interface of Numina.

3.1.27 `numina.util` —

3.1.28 `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

3.2 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

e

f

n

t

u

Symbols

A

C

D

E

F

I

M

N

O

P

Q

R

S

V