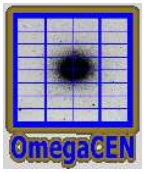

# **A**stro-**WISE** **E**nvironment

## User and Development Manual

October 16, 2015


Coordinator

# Contents

# Part I

# User's and Developer's

# Chapter 1

# Introduction

This document describes the Astro-WISE data reduction environment, `AWE` or the Astro-WISE Environment. It is aimed at both the beginning and the advanced user with reference for administrators (these more advanced topics will be indicated). Don't be fooled by the term *developer*. Any user who has ever looked into a source file and made even minor modifications is considered a *developer* here. There are things in this manual for all levels of user/developer, from beginner to advanced.

The first part of this document describes data processing in `AWE`, quality control in `AWE`, concepts for developing `AWE` code, database tasks, and `AWE` interface background. The second part consist of HOW-TOs: short, task-specific manuals showing how various aspects of the system work. The HOW-TOs will be referenced throughout the document.

New users should read this introduction chapter then move on to either the data reduction cookbook for a step-by-step walk-through, or go straight to the HOW-TOs for a task-by-task approach. There is an online version of the HOW-TOs for convenience at the Astro-WISE Portal:

`http://portal.astro-wise.org/`

## 1.1 Overview

### 1.1.1 History

The Astro-WISE Environment (`AWE`) was created by The Astro-WISE Consortium, a partnership of:

- OmegaCEN-NOVA at the Kapteyn Institute in Groningen, The Netherlands,
- Osservatorio Astronomico di Capodimonte in Napoli, Italy,
- Terapix at IAP in Paris, France,
- ESO in Garching bei München, Germany, and
- Universitäts-Sternwarte München , Germany,

and coordinated by OmegaCEN-NOVA. `AWE` was conceived as *the* solution to handle the vast amounts of astronomical data generated by all-sky surveys, particularly those to be observed with OmegaCAM on the VLT Survey Telescope (VST) on Cerro Paranal in Chile.

While waiting for OmegaCAM and VST completion, `AWE` has been expanded to include data from many different sources. Currently supported data sources include the Wide-Field Imager (WFI)

on the 2.2m MPG at La Silla, the Wide-Field Camera (WFC) on the INT at La Palma, and SuprimeCAM (SUP) on Subaru at Mauna Kea. The public portions of these datasets can be browsed at the Supported Data Sources section of the Astro-WISE Portal.

### 1.1.2 Basic Philosophy

- General
  - The Astro-WISE Environment (AWE) is an information system for the scientific analysis of extremely large datasets. It utilizes federated databases and dataservers, and parallel compute clusters to manage these vast amounts of data.
  - It was originally designed and developed specifically for astronomical wide-field imaging surveys, but has been used for the analysis of large datasets of handwritten archives and can be applied to any application involving very large datasets.
  - AWE is a federated system: data can be made any one-location in the federated system, but used everywhere in the federated system. This allows collaboration between diverse working groups.
    NOTE: AWE is currently operational in The Netherlands (Groningen and Leiden), Germany (Bonn and Munich), and Itlay (Naples)
  - Raw data is sacred in AWE. All data in the system is derived from raw data and can be traced back to the raw data within the system. This facilitates on-the-fly re-processing (OTFR) whenever improved methods or data is available.
  - All data in AWE is linked via backward chaining. Thus system is an all-in-one system: ALL input and output of processes are stored in the information system.
  - In AWE, the most recent product of a series of processes is considered the best: newer is better. Processes in AWE automatically use the latest versions of calibration files and software codes, which are both assumed to be the best.
  - The processing of data in AWE is split up such a way that it is embarassingly parallel. For astronomical data, this means all processes are per CCD.
  - AWE makes extensive use of the Python programming/scripting language in all its aspects.
- Object Model
  - Data are represented as Python objects with attributes corresponding to both pixel-data and meta-data, and methods corresponding to actions to perform on the object and its attributes.
  - Each object is considered a target that can be "made" with Tasks (e.g., found in the astro/recipes directory ) which set mandatory dependencies and run the target's make() method. Every make() fills in the newly instantiated (i.e., created) object. For example:

    ```
    awe> bias = BiasFrame()
    ```

    instantiates an EMPTY bias frame object.

    ```
    awe> bias.make()
    ```

    completes/fills in the object's attributes such as the observation date, pixel and header data, data statistics, etc.

  - With the Target Processor, the concept of a UNIX "make" is duplicated where all dependencies are checked for up-to-dateness (this includes existence) and will have their own make() method run if they are not up-to-date. This occurs recursively back to the raw data.

    NOTE: This recursive "making" of objects does not extend to the Tasks as the dependencies are only checked for existence.
  - Each object has verify(), inspect(), and compare() methods. These ensure optimum data quality.

- Code Access
  - Users have direct access to the Python code base via CVS checkout at `http://cvs.astro-wise.org/`
  - Users have the ability to add their own code or modify certain existing code.
  - The core parts of the system should not be modified by users.
  - Recipes can be modified for convenience in the users own checkout.
  - All the other parts of the system should only be modified if there is a bug, and preferably by the maintaniner of that part.
  - See §1.1.4 for more information.

- Documentation
  - The Astro-WISE portal (`http://portal.astro-wise.org/`) is the primary location of documentation for AWE.
  - The HOW-TOs (`http://www.astro-wise.org/portal/aw_howtos.shtml`) are the main source of documentation for operating the system. They describe all aspects of data processing in AWE, contain tutorials and troubleshooting guides, and all other things AWE.
  - For issues not addressed in the documentation, and general AWE news items, mailing lists exist:
    Issues mailing list
    News mailing list
  - Documentation for the code can be found at http://doc.astro-wise.org/ or from the AWE prompt via Python's help() function.

- Services
  - See §1.2 for the various web-based interfaces to the system.

### 1.1.3 Hardware

In the architecture of the Astro-WISE system, three main components are identified: a file-server, a database, and a high-performance compute cluster.

The file-server stores FITS-files, while the database keeps track of the relations between these files and their processing history. It is also through this database that decisions are made about which files to retrieve during the various processing steps. The compute cluster is used to process the data. During processing, requests are made to the database for the raw science data and for the necessary calibration files, which are then retrieved from the file-server.

> NOTE: The only files the user has direct access to locally (e.g., current directory) are data products retrieved and/or created during processing on the local machine. Normal processing using the compute cluster will leave no such data products, but only log files at most.

### 1.1.4 Software

The software consists of code written in Python, and includes an interactive command line environment (see §8.1), along with web services for **viewing** and **editing** the database (see §1.2). On a lower level a number of existing C programs are used, such as SExtractor, LDAC, SWarp, and Eclipse. (see §2.5).

#### Python and Object Oriented Programming

The code for the Astro-WISE system is written in Python, which is a language highly suitable for Object Oriented Programming (OOP). Within the OOP style in which the code is written, *classes* are associated with the various conventional calibration images, data images, and other derived data products. For example, in our system, bias exposures become *instances* of the `RawBiasFrame` class, and twilight flats become instances of the `RawTwilightFlatFrame` class. These instances of classes are the "objects" of OOP.

Classes may have incorporated *methods* and *attributes*. Methods perform a task on the object they belong to, while attributes are properties such as constants, flags, or links to other objects that may be needed by methods. In Astro-WISE various recipes have been coded that control the creation of instances of these classes. There may be different ways to create an instance of a class depending on which attributes are set to what values, and which methods are used. A `ColdPixelMap` object, for example, can be instantiated from the database (i.e. as the result of a query or search) or it can be created by using its "make()" method. In the latter case the ColdPixelMap can be derived either from a `DomeFlatFrame` or a `TwilightFlatFrame`, depending on which of those two objects are specified as the "flat" attribute of the ColdPixelMap object.

Within the Object Oriented Programming style, *inheritance* is an important and powerful concept. Classes can inherit attributes and methods from other classes. For example, both the `DomeFlatFrame` and `TwilightFlatFrame` classes are derived from the base class `BaseFlatFrame`; both are flat-fields afterall. Any method or attribute defined by the `BaseFlatFrame` class is inherited by both classes and both classes are free to redefine (this is called *polymorphism*) those methods or attributes and even add new ones as needed.

The bottom line is that Astro-WISE turns data into objects that are instances of Python classes with attributes and methods that can be inherited.

There is a significant amount of on-line documentation available for Python. Please see the Python web site `http://www.python.org` for further background on Python and for Python tutorials.

## 1.2 Web Services

### 1.2.1 Database Viewer

A web-service is available to view the database content and pixel data. It can be found at the following address:

`http://dbview.astro-wise.org/`

Help pages are provided by the webservice itself.

### 1.2.2 Database "Editor"

For a limited number of operations editing of database values is allowed. Specifically, it is possible to change valid ranges of calibration data (timestamps), and flags, to disqualify bad data. A special web-service tool to change these values can be found at the following web-site:

http://calts.astro-wise.org/

Help pages are provided by the webservice itself.

### 1.2.3 Processing Web Interface

The Target Processor is the culmination of all the benefits of the Astro-WISE system. It allows processing of a *target* (an end data product such as a `ReducedScienceFrame` or `SourceList`) and any of its dependencies that require it, on a parallel (e.g., compute cluster) or on a single (e.g., local machine) host. The dependency chain is followed back towards the raw data (**backward chaining**) to make sure only those objects requiring processing are actually processed.

http://process.astro-wise.org/

A web-based version of the `awe`-prompt command-line interface also exists for convenience (account required).

http://process.astro-wise.org/AWE

### 1.2.4 Image Handling Services

Services involving image sections (dependency cutouts, RGB generator, etc.) can be found in the IMGView service:

http://imageview.astro-wise.org/

## 1.3 Further Websites

The OmegaCAM web site:

http://www.astro.rug.nl/~omegacam/

The Astro-WISE web site:

http://www.astro-wise.org/

The Astro-WISE Web Services web site:

http://portal.astro-wise.org/

# Chapter 2

# Data Reduction Concepts and Walk-throughs

This chapter is intended as an introduction and simple reference document for data processing. It will provide examples of how to **use** the system, but it will not show how to **set up** the different necessary components. In particular it is assumed the user has a working system, and has access to the database, a parallel computing cluster and his/her own CVS checkout of the code. Once you have obtained the code, an environment variable called `$AWEPIPE` should point to the installation directory (awe). This variable is referenced repeatedly in the following text.

## 2.1 Processing steps

Several key points can be distinguished in the data reduction process:

- Ingesting raw data into database/data-server
- Producing calibration files
- Producing calibrated science data (applying calibration files)
- Coaddition of calibrated science data
- Source extraction
- User specific (much more so than previous steps at least)

This is also the order in which various pipelines (recipes) need to be run so that the necessary calibration files are present in the database. See §§2.3.4, 2.3.5, 2.3.6, and 2.3.7 for more specific information about the pipelines.

## 2.2 Ingesting raw data into the database

*See § 7.5.*

The first step in data reduction is the ingestion of the raw data into the database. This is handled by a *recipe* called `Ingest.py`, which can be found in `$AWEPIPE/astro/toolbox/ingest`. The *recipe* is invoked from the Unix command line with the following command :

```
awe $AWEPIPE/astro/toolbox/ingest/Ingest.py -i <raw data> -p <purpose> [-commit]
```

where `<raw data>` is a list of input data (filenames) to ingest, and `<purpose>` the purpose for which the input data was obtained. The input data should be be unsplit and uncompressed.

## 2.3 Data processing

This section will distinguish three general ways of using the system to do data reduction. The first, most laborious one, is to do this interactively, step-by-step at the Python prompt. While unsuitable to process a lot of data quickly, this gives a lot of insight into the inner workings of the code, and it may show the strengths of the system fairly evidently. It is also possible to use *recipes* to reduce data on a single machine. This is helpful while testing. Finally it is possible to use a parallel cluster, which is the most convenient way to process large volumes of data quickly.

Three characteristics of the Astro-WISE Environment (`AWE`) should be known prior to calibrating data with it.

1 The newest versions of calibration files present in `AWE` are considered best by default.

2 Each calibration file in the system has its individual period of validity associated with it, which is called its *timestamp.*

*3 Any calibration file in the system can be flagged as invalid to ensure that it will not be used in data calibration. The flag is called the* superflag.

The implication of these characteristics for automated calibration with `AWE` can be illustrated well with an example. Assume that a science exposure is to be automatically calibrated. `AWE` will search for calibration files (e.g., `BiasFrame`, `MasterFlatFrame`, etc.) that have a timestamp which encompasses the time at which the exposure was taken and have no superflag. If, for example, one `BiasFrame` exists it will be used. If more than one exists, the last created one will be used. (If none exist the system might try to construct one in some situations.)

An instrument specific note: to facilitate automated calibration for data from the ESO Wide Field Imager (WFI), calibration files have been created which have timestamps encompassing the times of all WFI images. The creation date of these "forever valid" data is set to a date before any of the other calibration data present in the system. The calibration files are `ReadNoise`, `BiasFrame`, `GainLinearity`, `HotPixelMap`, `ColdPixelMap`, `DomeFlatFrame`, `TwilightFlatFrame`, `MasterFlatFrame`, `FringeFrame`, photometric zeropoints and exinction curves, and bandpass transformations. The filter dependent calibration files have been derived for the U (#841, #877), B (#842, #878), V (#843) , R (#844) and I (#845, #879) broad-band filters. Thus for the calibration of WFI science image through such filters, the calibration files will always be available. For optimal reduction of a specific science dataset one should keep in mind that these defaults might not represent the best calibration files.

### 2.3.1 Interactive processing

An example of interactive processing:

```
awe> from astro.main.BiasFrame import BiasFrame
awe> from astro.main.RawFrame import RawBiasFrame
awe> from astro.main.ReadNoise import ReadNoise
awe> rn = ReadNoise.select(instrument='WFI', chip='ccd50', date='2001-02-01')
awe> r = RawBiasFrame.select(instrument='WFI', date='2001-02-01', chip='ccd50')
awe> for raw in r: raw.retrieve()
```

```
...
awe> b = BiasFrame()
awe> b.raw_bias_frames = list(r)
awe> b.read_noise = rn
awe> b.process_params.OVERSCAN_CORRECTION = 1
awe> b.set_filename()
awe> b.make()
awe> b.store()
awe> b.commit()
```

This will select a readnoise calibration file from the database, raw bias frames for the night of January 2nd, 2001, *retrieve* the FITS files from the data-server, create a masterbias frame, then upload the image to the data-server (*store*) and finally *commit* its dependencies and meta-data to the database. Note that a process parameter was tweaked. Whenever there is a possibility to adjust parameters, this is done by changing their values in the associated -Parameters class designated by the process_params attribute of BiasFrame. See §8.4 for a more exhaustive explanantion of how process parameters are set in the system.

Note that in interactive processing you may be tempted to use loops such as the following **wrong** code:

```
awe> sl = SourceList()
awe> query = RegriddedFrame.filename.like('*MYNAME*ccd52*.fits')
awe> for frame in query:
...       sl.frame = frame
...       sl.make()
...       sl.commit()
```

This is incorrect code in AWE because of the way objects are stored in the database (made persistent). A new instance of SourceList has to be created for every SourceList that you want to commit to the database. That is, the instantiation of the SourceList object should be done *within* the loop in this example:

```
awe> query = RegriddedFrame.filename.like('*MYNAME*ccd52*.fits')
awe> for frame in query:
...       sl = SourceList()
...       sl.frame = frame
...       sl.make()
...       sl.commit()
```

### 2.3.2 Non-parallel processing

Python classes are available to do any of the calibration steps; these classes act as recipes. They are located in **$AWEPIPE/astro/recipes/**, and must be imported into the Python interpreter, and can then be 'run':

```
awe> from astro.recipes.DomeFlat import DomeFlatTask
awe> task = DomeFlatTask(instrument='WFI', date='2000-04-28', chip='ccd50',
                         filter='#842', commit=1)
awe> task.execute()
```

It is possible to call the help file on these classes:

```
awe> help(DomeFlatTask)
```

A page containing docstrings, methods etc. defined in DomeFlatTask will be shown. Hit "q" to exit this page.

### 2.3.3 Parallel processing

*See §7.7.*

When the `awe`-prompt starts, an instance of the class "Processor" is automatically created and given the name "dpu" (Distributed Processing Unit). Using this class you can run tasks in parallel. Start a task as follows:

```
awe> dpu.run('ReadNoise', d='2000-04-28', i='WFI', c='ccd50', oc=6, C=1)
```

Here the first argument is the task name, the possible arguments can be found in table 10.1. The other arguments are query arguments, "d" is the "date" at the start of the observing night for which this ReadNoise object is derived, "i" is the "instrument" identifier, "c" is the "chip" (CCD) identfier (omit this argument to process the data for all the CCDs of "instrument" simultaneously), "oc" is the "overscan" correction method, and "C" is the "commit" switch.

Any invalid input to the processor is caught and a usage message is printed. Also note that if you have a local checkout of the code, this code and any changes to it are sent to the DPU and used there.

### 2.3.4 The bias pipeline

*See section 10.*

Recipes used:

- ReadNoise.py
- Bias.py
- HotPixels.py
- GainLinearity.py

### 2.3.5 The flat-field pipeline

*See section 10.*

Recipes used:

- DomeFlat.py
- ColdPixels.py
- TwilightFlat.py
- MasterFlat.py
- NightSkyFlat.py
- FringeFlat.py

### 2.3.6 The photometric pipeline

*See section 10.*

Recipes that can be used:

- PhotCalExtractResulttable.py
- PhotCalExtractZeropoint.py (this recipe represents the *OmegaCAM* pipeline)

These recipes and their underlying `Task` classes are described in detail in the chapters dedicated to the photometric pipeline.

### 2.3.7 The image pipeline

*See §21.1.*

The image pipeline is used to process raw science data and needs the outputs from the various calibration pipelines. The calibration steps performed are de-biasing, flatfielding, astrometric calibration and photometric calibration. The recipes that relate to the image pipeline are:

- Reduce.py
- Astrometry.py
- GAstrometricSourceList.py
- GAstrom.py
- Regrid.py
- Coadd.py

The `Reduce` recipe de-biases and flat-fields the raw science data. Astrometry can be done in two ways. First `Astrometry` derives a astrometric solution. After Astrometry has been run, it is possible to try to improve the astrometric solution by using overlap regions of all the images in a dither pattern. To this end `GAstrometricSourceList` creates a SourceList that may be used when running the `GAstrom` recipe. `Regrid` resamples a ReducedScienceFrame into a new grid of pixels, so that RegriddedFrames can be easily coadded into CoaddedRegriddedFrames.

Starting the image pipeline in single-CCD mode is quite easy. To reduce the data of a given raw science frame from the `awe`-prompt:

```
awe> r = ReduceTask(raw_filenames=['<input name>'], commit=1)
awe> r.execute()
```

where `<input name>` is the filename of the raw science frame to calibrate. Or:

```
awe> r = ReduceTask(instrument=<instrument>, date=<date>, filter=<filter>,
                    chip=<chip>, object_name=<object name>, commit=1)
awe> r.execute()
```

The recipe for running the image pipeline in parallel mode is the same one as used for running the calibration pipelines. In this case, however, the `-task` switch is set to either `Reduce` or `Science`. The command issued to run the pipeline is:

```
awe> dpu.run('Reduce', i=<instrument name>, d=<date>, f=<filter name>,\
...          o=<object name>, C=<commit: 0 or 1>)
```

Raw images that are ingested into the database have an attribute "OBJECT", which is matched to "object name" in the above statement. This OBJECT is the value of the header keyword OBJECT from the raw image. It is possible to use the wildcards "?" and "*" in the object name, which act similar to Unix command line wildcards.

**Photometric calibration in the image pipeline**

The photometric calibration in the image pipeline is achieved by writing the zeropoint and extinction information from calfile 563 into the header of the science frame. In order for this to work, these calfiles (obviously) have to be present in the database for every combination of chip and filter. The quick creation of these calfiles without having to run the photometric pipeline is decribed in the relevant chapters on the photometric pipeline.

## 2.4 Timestamps

For the smooth running of the image pipeline, some manual adjustments of the contents of the database are sometimes necessary. This is particularly true for the `timestamping` of the various calibration files, because the selection of the right calibration file depends on these timestamps.

Every calibration file has three timestamps, of which two determine the validity range of the file. These timestamps are `timestamp_start`, `timestamp_end`, and `creation_date`, respectively. The default timestamps that are created in the calibration pipeline are set to reflect the calibration plan of OmegaCAM. However, these timestamps are not really suited for 'random' sets of data, or for data which are not subjected to a rigorous calibration plan. It is therefore necessary to adjust the timestamps of the calfiles produced so that these fit the 'observing schedule' of the data at hand. This can be done using the database timestamp editor (see http://calts.astro-wise.org/).

## 2.5 Interfaces to other programs

### 2.5.1 SQL interface, interaction with the database

See §8.3 for information about this interface.

### 2.5.2 Eclipse interface

For image arithmetic the C library Eclipse is used. In order to use this library in `AWE` a Python wrapper/interface was written. There are three main classes used in the `AWE` in this interface: **image**, **cube**, and **pixelmap**, representing much used data structures. Here is an example of its use:

```
awe> import eclipse
awe> bias = eclipse.image.image('bias.fits')
awe> flat = eclipse.image.image('flat.fits')
awe> sci = eclipse.image.image('science.fits')
awe> result = (sci-bias) / flat
awe> result.save('sci_red.fits')
```

Note that in the above example "science.fits" is a trimmed image that has to be equal in shape and size to the bias and flat. Master bias and master flat files retrieved from the database are trimmed, while raw science data is not. Also note that a new header is created for "result" in the example above. You may want to keep the header of the science image though:

```
awe> hdr = eclipse.header.header('science.fits')
awe> result.save('sci_red.fits', hdr)
```

NOTE: Eclipse headers can be used at this low level, but for compatibility and advanced functionality like header verification, `AWE` uses DARMA headers based on the PyFITS interface (see §7.8 for more details).

Regions can be cut from images:

```
awe> region = result.extract_region(1,1,100,100)
awe> region.save('sci_red.region.fits')
```

If you specify the header for saving here it will adjust values such as "NAXIS1" and "NAXIS2" to reflect the real size of the image.

Statistics can be calculated in the following way (assume that "coldpixels.fits" is an 8-bit pixelmap FITS file locating cold pixels):

```
awe> coldpixels = eclipse.pixelmap.pixelmap('coldpixels.fits')
awe> mask = ~coldpixels
awe> stats = result.stat_opts(pixelmap=mask, zone=[1,1,100,100])
awe> stats.median
1412.8621
```

Note the bitwise negation operator (~) to switch between "masks" (bad pixels 0) and "flags" (bad pixels 1). A mask is optional for calculating the statistics.

Images (i.e. image objects) can be stacked in a cube:

```
awe> b1 = eclipse.image.image('bias1.fits')
awe> b2 = eclipse.image.image('bias2.fits')
awe> b3 = eclipse.image.image('bias3.fits')
awe> c = eclipse.cube.cube([b1,b2,b3])
awe> med_av = c.median()
we> med_av.save('med_av.fits')
```

Other functionalities such as Fourier transforms, image filtering, etc. are supported. For further information, import eclipse in Python and use the help functionality provided by Python. (See §7.2.)

### 2.5.3 SWarp interface

SWarp is an image coaddition program, that performs pixel remapping, projections etc. This interface is very straightforward, as it simply writes a configuration file such as used by this program (similar to SExtractor) and then calls the program itself.

```
awe> from astro.external import Swarp
awe> from astro.main.Config import create_config
awe> swarpconfig = create_config('swarp')
awe> swarpconfig.COMBINE = 'N'
awe> swarpconfig.RESAMPLE = 'Y'
awe> files = ['file1.fits', 'file2.fits', 'file3.fits']
awe> Swarp.swarp(files, config=swarpconfig)
```

The first argument of Swarp.Swarp is a list of files to be SWarped. The second is the optional Config object whose options can be set in multiple ways (see below), including the direct setting as shown above.

### 2.5.4 SExtractor interface

SExtractor is used to extract sources from images. While this is handled by the Catalog class, one can also call the SExtractor interface directly.

```
awe> from astro.external import Sextractor
awe> from astro.main.Config import create_config, create_params
awe> sexconf = create_config('sextractor')
awe> sexconf.set_from_keys(DETECT_THRESH=2.0, CATALOG_NAME='mycatalog.cat')
awe> sexparams = create_params('sextractor')
awe> sexparams.update_list(['FLUX_ISOCOR'])
awe> sci = 'sci_1.fits'
awe> Sextractor.sex(sci, params=sexparams, config=sexconf)
```

In general, the first argument of Sextractor.sex is the detection (and measurement) image, the second is an optional measurement image, the third is possible *extra* output parameters other than those specified in the interface in the form of a Parameters object (from astro.main.Config). The final argument is the configuration (a Config object), which can be updated from separate keyword arguments in KEYWORD1='value1', KEYWORD2='value2', etc. format as shown in the 'set_from_keys' call.

The mycatalog.cat catalog is a FITS table. Here follows an example of one way to work with the data in mycatalog.cat.

```
awe> import pyfits
awe> hdu=pyfits.open('mycatalog.cat')
awe> flux_isocor=hdu[2].data.field('FLUX_ISOCOR')
```

The use of the `Catalog` class is discouraged, but explained below for completeness:

```
awe> from astro.main.Catalog import Catalog
awe> from astro.main.BaseFrame import BaseFrame
awe> cat = Catalog(pathname='mycatalog.cat')
awe> cat.frame = BaseFrame(pathname='sci_1.fits')
awe> cat.sexparam = ['FLUX_ISOCOR']
awe> cat.sexconf['DETECT_THRESH'] = 2.0
awe> cat.make()
```

The above can be extended with:

```
awe> cat.make_skycat()
```

This will make a skycat catalog called "mycatalog.scat", which can be overlayed on the FITS image ("sci_1.fits") when using ESO's Skycat viewer.

### 2.5.5 LDAC interface

LDAC (Leiden Data Analysis Center) tools are used in the system to do tasks such as astrometry and photometry. In particular, these tools provide a way to manipulate and associate binary FITS catalogs. Hence catalogs as created in the previous section can be manipulated from Python with the LDAC interface.

```
awe> from astro.external import LDAC
awe> incat = 'science.cat'
awe> outcat = 'science.filtered.cat'
awe> ldac = LDAC.LDAC()
awe> ldac.filter(incat, outcat, table_name='OBJECTS', sel='FLUX_RADIUS > 5.0')
```

These few lines will filter a catalog in file "science.cat" so that only astrophysical objects with a half-light radius larger than 5.0 pixels are placed in the output catalog. Note that LDAC is very picky about the syntax of the "sel" selection statement, so be careful here.

## 2.6 A short example

### 2.6.1 Outline

When reducing data, instrumental footprints are removed from the science data and it is calibrated astrometrically and photometrically. This is done by what we call the image pipeline. This short demo skips the creation of the calibration data (biases, flat-fields etc.) and shows how to reduce science data, and then find and inspect the results in the database.

Start the `awe`-prompt by typing `awe`.

### 2.6.2 The image pipeline

In this case we want to reduce data observed on the night of April, 28, 2000, on ESO's 2.2m WFI telescope, using the Johnson B filter (identifier: #842). This information is necessary to give on the command line for the data to be found in the database:

```
awe> dpu.run('Reduce', d='2000-04-28', i='WFI', f='#842', C=1)
```

The job will be submitted to the queue. Check the DPU web page (Groningen). Wait for the jobs to finish. Logs of the processing can be retrieved as follows:

```
awe> dpu.get_logs()
```

### 2.6.3 Finding the result in the database

Now we want to check the database for the created files, and obtain the reduced images to check the result:

```
awe> query = ReducedScienceFrame.select(instrument='WFI', date='2000-04-28',
                                        filter='#842', chip='ccd50',
                                        object='CDF4_B_3')
awe> for frame in query: print frame.filename, frame.quality_flags,
...                           frame.chip.name, frame.filter.name,
...                           frame.OBJECT, frame.creation_date
...
Sci-DEMO-WFI-----#842-ccd50---Sci-53256.5263356.fits 0 ccd50 #842 CDF4_B_3
2004-09-08 12:38:06.00
```

The last item is the creation date of the ReducedScienceFrame, here you can check that the ReducedScienceFrame(s) selected include those that were just made. It is possible to fully track the history of each image this way.
Do not close the `awe`-prompt at this point (see next section).

### 2.6.4 Retrieving the images to check the results

It is now possible to download the images from the data-server(s). This can be done after selecting the images after doing the above query:

```
awe> q = ReducedScienceFrame.filename == 'Sci-DEMO-WFI-----#842-ccd50---Sci-532
56.5263356.fits'
awe> frame = q[0]
awe> frame.retrieve()
```

Note that the result of the query is in the form of a list, even if the result of the query is only one object. Hence obtain the first element and retrieve the image. The image can now be viewed with your favourite FITS viewer.

## 2.7 A lengthy example

This section will recap the preceding ones to show how to proceed from the point of having a data tape to a question such as: give me a plot of half-light radius versus magnitude for the objects in this field. As data set for this example, 1/4th of the Capodimonte Deep Field (B filter) is used. This area has a size of approximately 30' × 30' (the WFI field of view). The observations for this data set were done on the 28th of April, 2000.

### 2.7.1 Ingesting (skip in case of demo, process on local machine)

It is assumed that we have copied all the data for this date from tape (presumably) to hard disk, so that it is located in for example **/Users/users/data/**. It is now necessary to know the type of (raw) data for each file (bias, twilight flat, dark etc.). The data set needs to be *ingested* into the database: the multi-extension FITS files are split into single CCD parts and stored on the data-server, and RawFrame objects are created in the database.

Since the total amount of files is considerable, it is convenient to list these by type in ascii files. In our case a file called **bias_files.txt** containing the bias file names looks like:

```
WFI.2000-04-28T00:01:00.fits
WFI.2000-04-28T00:01:30.fits
WFI.2000-04-28T00:02:00.fits
WFI.2000-04-28T00:02:30.fits
WFI.2000-04-28T00:03:00.fits
WFI.2000-04-28T00:03:30.fits
WFI.2000-04-28T00:04:00.fits
WFI.2000-04-28T00:04:30.fits
WFI.2000-04-28T00:05:00.fits
```

These biases can be ingested into the database by looping over this file as follows:

```
unixprompt>foreach i ( `cat bias_files.txt` )
foreach? awe $AWEPIPE/astro/toolbox/ingest/Ingest.py -i $i -t bias -commit
foreach? end
```

Repeat (use option -t dome and -t twilight) for the dome- and twilight flats. The raw calibration data should now be present in the database and data-server as RawBiasFrame, RawDomeFlatFrame and RawTwilightFlatFrame instances.

One can check this by using the database viewer (http://dbview.astro-wise.org).

### 2.7.2 Image calibration files

Assuming everything went well, we are ready to start creating calibration files. This will be done using a parallel computing cluster.

Use the following command to create read noise objects, which are necessary to create master biases:

```
awe> dpu.run('ReadNoise', i='WFI', d='2000-04-28', C=1)
```

Check your DPU queue webpage to view the status of your job and wait for them to finish. Then use the following command to create master biases for each CCD:

```
awe> dpu.run('Bias', i='WFI', d='2000-04-28', C=1)
```

Once a job is finished the log file for it can be obtained from the DPU:

```
awe> dpu.get_logs()
```

Once the biases finish the other necessary calibration steps need to be performed (in this order) as follows:

```
awe> dpu.run('HotPixels', i='WFI', d='2000-04-28', C=1)
awe> # Wait for jobs to finish (check web page for queue)
awe> dpu.run('DomeFlat', i='WFI', d='2000-04-28', f='#842', C=1)
awe> # Wait for jobs to finish (check web page for queue)
awe> dpu.run('ColdPixels', i='WFI', d='2000-04-28', f='#842', C=1)
awe> # Wait for jobs to finish (check web page for queue)
awe> dpu.run('TwilightFlat', i='WFI', d='2000-04-28', f='#842', C=1)
awe> # Wait for jobs to finish (check web page for queue)
awe> dpu.run('MasterFlat', i='WFI', d='2000-04-28', f='#842', C=1)
awe> # Wait for jobs to finish (check web page for queue)
```

### 2.7.3 Photometric calibration files

In order to run the image pipeline and do photometry, a **PhotometricParameters** object is required. There are two ways to proceed here, as described in the sections below.

#### Manual values (essentially no photometry)

For relative photometric calibration in the case of WFI observations, values for the zeropoint and extinction can be entered manually:

```
awe $AWEPIPE/astro/toolbox/photometry/ingest_photometrics.py -z 25.00 -c ccd50
                                                             -f #842 -e 0.19
                                                             -start 2000-01-01
                                                             -end 2000-12-31
awe $AWEPIPE/astro/toolbox/photometry/ingest_photometrics.py -z 24.95 -c ccd51
                                                             -f #842 -e 0.19
                                                             -start 2000-01-01
                                                             -end 2000-12-31
etc.
```

where the option "z" is the zeropoint and "e" the extinction. Repeat this for each CCD (ccd50-ccd57) in the WFI detector. With these default PhotometricParameters objects in place it is possible to run the image pipeline (§2.7.4).

**Using standard star fields (absolute photometry)**

See §§19.2, 19.3, and 19.4 for detailed instruction on how to do accurate photometry. This is outside the scope of this example.

### 2.7.4 Image pipeline

Now that all calibration files that are necessary have been produced, we can continue by applying all these to the science data. This is done by running a recipe that represents the so-called image pipeline:

```
awe> dpu.run('Reduce', i='WFI', d='2000-04-28', f='#842', o='CDF4_B_?', C=1)
```

The data used in the case of this example consists of 10 dithered exposures that we intend to coadd into one image. The above example will select RawScienceFrames from the database, using in particular the "like" functionality of the SQL interface in selecting for matches of the OBJECT header keyword, and applies the calibration data. This results in 80 ReducedScienceFrames that are stored in the database. In addition these reduced science frames are resampled to a new grid. The grid centers for this system are fixed so that pixels in these `RegriddedFrame`s can be combined without resampling again first.

After the job completes there should be 80 new ReducedScienceFrames and 80 new Regridded-Frames in the database. One can check this from the `AWE`/Python interpreter as follows:

```
awe> s = ReducedScienceFrame.select(instrument='WFI', date='2000-04-28',
...                                 filter='#842')
awe> len(s)
361
```

If this search turns up more than 80 science frames as above, this means other data has been reduced (possibly by other persons) for this filter and for this night. To get a better idea of what is present in the database for this night one could proceed as follows:

```
awe> for f in s: print f.raw.filename, f.filter.name, f.chip.name,
f.EXPTIME, f.OBJECT
...
```

(press enter when prompted with '...' to close the statement block in the above loop)

### 2.7.5 Coaddition

The RegriddedFrames created in the previous step can be coadded into a single mozaic, to form the intended contiguous region on the sky. In order to coadd the data, one can do the following:

```
awe> dpu.run('Coadd', i='WFI', d='2000-04-28', f='#842', o='CDF4_B_?', C=1)
```

A lot of files now need to be retrieved from the data-server, namely all the RegriddedFrames and all WeightFrames associated with these. After processing finishes, you should now have a nice image of 1/4th of the Capodimonte Deep Field.

### 2.7.6 Source lists

*See §21.6.*

To make a source list of the image we made above, where the information of the sources is available from the database, one can do the following:

```
awe> sl = SourceList()
awe> query = CoaddedRegriddedFrame.filename.like('Sci*Coadd*.fits')
awe> sl.frame = query[0]
awe> sl.frame.retrieve()
awe> sl.name = 'DEMO-sourcelist'
awe> sl.sexconf.DETECTION_THRESHOLD = 2.0
awe> sl.sexparam = ['MAG_AUTO', 'MAGERR_AUTO', 'FLUX_RADIUS']
awe> sl.make()
awe> sl.commit()
```

One can now select the source list from the database and check its global properties or even specific information about the sources in the source list.

```
awe> query = SourceList.name == 'DEMO-sourcelist'
awe> sl = query[0]
awe> print len(sl.sources)
180000
etc.
```

# Chapter 3

# Quality Control

## 3.1 General concepts

Explain:

- verify, compare, inspect paradigm
- timestamps as supreme QC tool.
- how to handle 'bad' data -difference calibration pipeline, image pipeline

### 3.1.1 Timestamps

For the smooth running of the image pipeline, some manual adjustments of the contents of the database are sometimes necessary. This is particularly true for the `timestamping` of the various calibration files, because the selection of the right calibration file depends on these timestamps.

Every calibration file has three timestamps, of which two determine the validity range of the file. These timestamps are `timestamp_start`, `timestamp_end`, and `creation_date`, respectively. The default timestamps that are created in the calibration pipeline are set to reflect the calibration plan of OmegaCAM. However, these timestamps are not really suited for 'random' sets of data, or for data which are not subjected to a rigorous calibration plan. It is therefore necessary to adjust the timestamps of the calfiles produced so that these fit the 'observing schedule' of the data at hand. This can be done using the database calts web-service (see http://calts.astro-wise.org/).

It can happen that timestamp ranges overlap for two or more calibration files. In the pipeline the one with the most recent `creation_date` is used. To make a calibration file valid forever, `timestamp_start` should be set to January 1st, 1990, `timestamp_end` should be set to January 1st, 2030 and the `creation_date` should be set to January 1st, 1990.

## 3.2 Quality control of biases, flat-fields and fringing

Group composition: Juan Alcalà, Ewout Helmich, Roberto Silvotti, Philippe Heraudeau, Mike Pavlov, Alfredo Volpicelli
Leader: Juan Alcalà

tool: tool: general small windows - flatness

### 3.2.1 General scheme

The QC on biases, flat-fields and fringing pattern is based on two different steps:

1. A first simple QC on RawFrames (RawBiasFrame, RawDomeFlatFrame and RawTwilight-FlatFrame) is done during the ingestion, based on the verify method only. If the verify provides a negative result, the frame is flagged as bad and will never be used by the pipeline.

2. The same concept as before is used for the Masters (Bias, Dome, Twilight); moreover a compare method is also defined and used to compare the current master frame with the previous one. Also in this case, if the compare method gives a negative result, the frame is flagged as bad and will never be used by the pipeline. Finally, for what concerns the MasterFlat and the FringingPattern, only the compare method is used.

Note that both the verify and compare methods can host many independents quality controls. The idea is that each of these quality controls should be able to detect specific issues (indicated in square brackets below) and hence check whether this problem is present or not in that particular frame.

Note also that the values of all the parameters and in particular of all the thresholds, whose default values are given in the following sections, will need a fine tuning in order to be optimized for each particular instrument (WFI first and then OmegaCAM).

### 3.2.2 SubWinStat Class

Most of the QC tools described in the next sections make use of the SubWinStat Class, which allows to derive, through eclipse, the statistical properties of a particular region (subwindow) of a frame.
The Class SubWinStat by default uses 4 subwindows in x axis and 8 subwindows in y axis. The user who wants to change the number of subwindows must operate in the following way:

```
awe> from astro.main.BiasFrame import BiasFrame
awe> from astro.main.SubWinStat import SubWinStat
awe> bias = BiasFrame.select(instrument='WFI', date='2000-04-27', chip='ccd50')
awe> subwinstat = SubWinStat()
awe> subwinstat.frame = bias
# change number of windows in x
awe> subwinstat.process_params.NUMBER_OF_WINDOWS_X = 8
# change number of windows in y
awe> subwinstat.process_params.NUMBER_OF_WINDOWS_Y = 16
# calculate stats in subwindows
awe> subwinstat.make()
# commit subwinstat into DB
awe> subwinstat.commit()
```

Note that for all the BaseFrame objects it is possible to use SubWinStat.

### 3.2.3 RawBiasFrame

*Verify method:* the following conditions must be satisfied:

- standard deviation of the whole frame < 20
  [remove noisy biases]
- (Max median in subwindows) - (Min median in subwindows) < 10
  [check whether "flatness" is below a pre-defined value]

### 3.2.4 RawDomeFlatFrame

*Verify method:* the following condition must be satisfied:

- 5000 < mean < 45000
  [remove saturated frames or frames with very low S/N ratio]

### 3.2.5 RawTwilightFlatFrame

*Verify method:* the following condition must be satisfied:

- 5000 < mean < 45000
  [remove saturated frames or frames with very low S/N ratio]

### 3.2.6 BiasFrame (MASTER BIAS)

*Verify method:* the following conditions must be ALWAYS satisfied:

- standard deviation of the whole frame < MAXIMUM_STDEV=10
  [remove noisy master biases]

IF the frame is NOT OVERSCAN_CORRECTED THEN
the following condition must be satisfied:

- (Max median in subwin) - (Min median in subwin) < MAXIMUM_SUBWIN_FLATNESS=10
  [to check whether "flatness" is below a pre-defined value]

ELSE:

- abs(mean) < MAXIMUM_ABS_MEAN=10
  [check whether the mean is close to zero]

*Compare method:* the following condition must be satisfied:

- abs(stdev whole frame - stdev previous frame) < MAXIMUM_STDEV_DIFFERENCE=5
  [compare stdev of MASTER BIAS with that of the previous one]

The thresholds used by BiasFrame QC can be modified in an interactive way, as follows:

```
awe> from astro.main.BiasFrame import BiasFrame
awe> b = BiasFrame.select(instrument='WFI', date='2000-04-27', chip='ccd50')
awe> b.make_subwinstat()
awe> b.process_params.MAXIMUM_STDEV = 5
awe> b.process_params.MAXIMUM_SUBWIN_FLATNESS  = 50.0
awe> b.process_params.MAXIMUM_ABS_MEAN = 8
awe> b.verify()
awe> b.process_params.MAXIMUM_STDEV_DIFFERENCE = 3.0
awe> b.compare()
```

### 3.2.7 DomeFlatFrame (MASTER DOME)

*Verify method:* the following condition must be satisfied:

- (Max median in subwin - Min median in subwin) < MAXIMUM_SUBWIN_FLATNESS=0.05
  [check whether the flatness is below e.g. 5%]

*Compare method:* the following condition must be satisfied:

- MAX (ratio_i) – MIN (ratio_i)< MAXIMUM_SUBWIN_DIFF=0.1
  where ratio_i is given by:
  ratio_i = $\frac{median\ of\ subwindow\ i\ in\ present\ MASTER\ DOME}{median\ of\ subwindow\ i\ in\ previous\ MASTER\ DOME}$
  [check how different the shapes of the two MASTER DOME are]

Example of parameter change:

```
awe> from astro.main.DomeFlatFrame import DomeFlatFrame
awe> d = DomeFlatFrame.select(instrument='WFI', filter='#842',
                              date='2000-04-27', chip='ccd50')
awe> d.make_subwinstat()
awe> d.process_params.MAXIMUM_SUBWIN_FLATNESS = 0.02
awe> d.verify()
awe> d.process_params.MAXIMUM_SUBWIN_DIFF = 0.05
awe> d.compare()
```

### 3.2.8 TwilightFlatFrame (MASTER TWILIGHT)

*Verify method:* the following conditions must be satisfied:

- (Max median in subwin) - (Min median in subwin) < MAXIMUM_SUBWIN_FLATNESS=0.05
  [check whether the flatness is below e.g. 5%]

*Compare method:* the following condition must be satisfied:

- MAX (ratio_i) – MIN (ratio_i)< MAXIMUM_SUBWIN_DIFF=0.1
  where ratio_i is given by:
  ratio_i = $\frac{median\ of\ subwindow\ i\ in\ present\ MASTER\ TWILIGHT}{median\ of\ subwindow\ i\ in\ previous\ MASTER\ TWILIGHT}$
  [check how different the shapes of the two MASTER TWILIGHT are]

Example of parameter change:

```
awe> from astro.main.TwilightFlatFrame import TwilightFlatFrame
awe> t = TwilightFlatFrame.select(instrument='WFI', filter='#842',
                                  date='2000-04-27', chip='ccd50')
awe> t.make_subwinstat()
awe> t.process_params.MAXIMUM_SUBWIN_FLATNESS = 0.02
awe> t.verify()
awe> t.process_params.MAXIMUM_SUBWIN_DIFF = 0.05
awe> t.compare()
```

### 3.2.9 MasterFlatFrame (MASTER FLAT)

*Compare method:* the following condition must be satisfied:

- MAX (ratio_i) – MIN (ratio_i) < MAXIMUM_SUBWIN_DIFF=0.1

  where ratio_i is given by:

  ratio_i = $\frac{median\ of\ subwindow\ i\ in\ present\ MASTER\ FLAT}{median\ of\ subwindow\ i\ in\ previous\ MASTER\ FLAT}$

  [check how different the shapes of the two MASTER FLATS are]

Example of parameter change:

```
awe> from astro.main.MasterFlatFrame import MasterFlatFrame
awe> m = MasterFlatFrame.select(instrument='WFI', filter='#842',
                                date='2000-04-27', chip='ccd50')
awe> m.make_subwinstat()
awe> m.process_params.MAXIMUM_SUBWIN_DIFF = 0.05
awe> m.compare()
```

### 3.2.10 Fringing

*Compare method:* once a reference fringing pattern (e.g. previous one) has been subtracted from the current one:

Difference = FringeFrame – Previous_FringeFrame

the following conditions on frame Difference must be satisfied:

- (MAX average in subwin) – (MIN average in subwin) < MAXIMUM_SUBWIN_MEAN_DIFF=1000. (e.g. presently excluded)

  [check whether there are significant differences between the two FRINGE PATTERNS (local differences, different slopes)]

- MAX (standard dev. in subwindows) < MAXIMUM_SUBWIN_STD=1000. (e.g. presently excluded)

  [check whether there are significant differences between the two FRINGE PATTERNS (global differences, different S/N ratios)]

Example of parameter change:

```
awe> from astro.main.FringeFrame import FringeFrame
awe> f = FringeFrame.select(instrument='WFI', filter='#842',
                            date='2000-04-27', chip='ccd50')
awe> f.make_subwinstat()
awe> f.process_params.MAXIMUM_SUBWIN_MEAN_DIFF= 10
awe> f.compare()
```

### 3.2.11 NOTES (OAC)

PRECONDITIONS: in the current version of the pipeline the minimum number of input frames to produce a Master (bias, dome, twilight) is 2; this seems too low (in particular for the twilight flat frames where residual stars are present). We suggest to increase these numbers to 5 (bias and dome) and 3 (twilight).

The same applies to the fringing pattern: in order to determine a good fringing pattern (i.e. without residual stars and with a sufficient S/N ratio) we need a number of scientific frames of the order of 7-8.

OVERSCAN: presently there are 8 different methods to calculate the OVERSCAN:

**0** – No overscan correction

**1** – Use median of prescan_x

**2** – Use median of overscan_x

**3** – Use median of prescan_y

**4** – Use median of overscan_y

**5** – Use per-row value of prescan_x

**6** – Use per-row value of overscan_x (default)

**7** – Use per-row average of prescan_x (smoothed)

**8** – Use per-row average of overscan_x (smoothed)

### 3.2.12 Quality flags

When any of the previous quality checks fails a flag (bit) will be set. The following gives an overview of the different types of quality flags (bits) for every type of calibration image. First the bit number is given, then flag name:

```
ReadNoise:
    0 READNOISE_TOO_HIGH
    1 BIAS_DIFFERENCE_TOO_HIGH
    2 READNOISE_DIFFERENCE_TOO_HIGH
BiasFrame:
    0 BIAS_ABS_MEAN_TOO_LARGE
    1 BIAS_STDEV_TOO_LARGE
    2 BIAS_STDEV_DIFFERENCE_TOO_LARGE
    3 BIAS_SUBWIN_FLATNESS_TOO_LARGE
    4 BIAS_SUBWIN_STDEV_TOO_LARGE
DomeFlatFrame:
    0 DOME_SUBWIN_FLATNESS_TOO_LARGE
    1 DOME_SUBWIN_DIFF_TOO_LARGE
TwilightFlatFrame:
    0 TWILIGHT_SUBWIN_FLATNESS_TOO_LARGE
    1 TWILIGHT_COUNT_OUTLAYERS_TOO_LARGE
    2 TWILIGHT_SUBWIN_DIFF_TOO_$LARGE
MasterFlatFrame:
```

```
    0 MASTER_SUBWIN_DIFF_TOO_LARGE
GainLinearity:
    0 GAIN_LOW
    1 GAIN_HIGH
    2 GAIN_DIFFER
ColdPixelMap:
    0 COLDPIXELCOUNT_TOO_LARGE
    1 COLDPIXELCOUNT_DIFFERENCE_TOO_LARGE
HotpixelMap:
    0 HOTPIXELCOUNT_TOO_LARGE
    1 HOTPIXELCOUNT_DIFFERENCE_TOO_LARGE
```

## 3.3 Quality control of the astrometry

Group composition: Erik Deul, Mario Radovic, Emmanuel Bertin (TBC)
Leader : Erik Deul

This is a list of parameters that can be used as quality control parameters. For each parameter the name, an example output and sensible limits are given.

### RMS and maximum residuals for reference stars

The RMS and maximum values (`RMS` and `Max`) in arcseconds of the residuals between the calculated position after astrometric calibration application and the known positions of reference catalog gives a good indication of the overall correctness of the astrometric solution.

The actual limiting values for which an astrometric solution is correct depends on a number of input parameters. These are: the RMS of input reference star catalog, the resolution at which the observations are performed and the seeing conditions at observation time. Further parameters driving the astrometric correctness estimate are the number of extracted - reference star pairs used in the astrometric solution and the degree of freedom for the sought polynomial deformation.

The limiting RMS for the solution should not exceed the the square root of the summed squares of the RMS for the reference stars and the positional accuracy for the extracted stars. The latter is, for well defined stellar profiles ($> 5\sigma$) detections 0.1 times the pixel size.

The maximum value of the positional residual, should for a Gaussian distribution at $5\sigma$ accuracy not exceed 5 times the RMS size. An example from one of the test runs from `astrom` yields the following values: `RMS` 0.428727 and `Max` 1.290292.

Given a comfortable limiting range for these values, one should discard astrometric solutions that have: `RMS` $> 0.5 arcsec$ and `Max` $> 1.5 arcsec$.

### Standard deviation on polynomial parameters

Statistical measures on the solution parameters of the astrometric calibration give good information about the quality of the actual solution. Three notions are important here.

First, the statistics on the number of reference objects used in the fitting process. This number is available for the case where a single frame is fitted, but in the case of solving multiple overlapping frames the number of objects used in the overlap is also available.

For a good solution one would like to have enough data elements to restrain the polynomial parameters. For each object pair (extracted - reference for single frames, and extracted - extracted for overlapping frames) both the x and y pixel coordinates are available (yielding to linear equations per pair). The number of required linear equations should well exceed the number of free parameters (polynomial degree = 1: 6 parameters equal 3 pairs, polynomial

degree 2: 12 parameter equals 6 pairs, polynomial degree 3: 20 parameters equal 10 pairs). To make sure individual random positional errors are well smoothed out a factor of 10 overrating the number of required pairs is a minimum requirement.

For a test run on astrometry this may yield:

```
Number of reference stars: 275
Number of overlap stars: 0
```

A certain limit to the number of reference stars used would be: $N_{ref} > 100$

Second, information about the derived offset parameters of the astrometric solution provides quality measure. The standard deviation on the offset should be close to the largest RMS of the input catalogs. An example run could give the following output:

```
            arcseconds          radians                plate center
frame  xoff    yoff std     xoff    yoff       std     ra        dec       err
  1   -2.78    2.04 0.26  -1.3e-05   9.9e-06  1e-06 2.5945597 -0.3620920  1e-06
```

For a `std` value that is $< 0.4 arcsec$ we will definitely throw away all astrometric solutions that are grossly in error or are determined under condition that did not meet the above criteria.

Third, the information about the first and high order terms of the astrometric solution give away precious detail about the correctness of the astrometry. This example is for non-overlap case:

```
   (arcseconds)                    (radians)
     X          Y           err        X          Y          err
  -0.2383   0.000261 +/-  0.00013  -0.002369  2.594e-06 +/- 1.29e-06 X*1*Cheb0(F)
 0.001006     0.2381 +/- 3.61e-05       1e-05   0.002366 +/- 3.59e-07 1*Y*Cheb0(F)
-3.276e-05 -7.776e-05 +/- 3.37e-05 -3.256e-07 -7.728e-07 +/- 3.35e-07 X**2*1*Cheb0(F)
 0.0001374  0.0001613 +/- 1.55e-05  1.366e-06  1.603e-06 +/- 1.54e-07 X*Y*Cheb0(F)
 5.862e-06  6.608e-05 +/- 8.41e-06  5.826e-08  6.568e-07 +/- 8.35e-08 1*Y**2*Cheb0(F)
```

The errors in the parameters should not exceed $0.3/N_{ref}/P_{deg}$; where $N_{ref}$ is the number of reference objects and $P_{deg}$ is the degree of the astrometric polynomial.

For OmegaCAM the set of polynomial parameters (particularly the first order) should be strict and well known after commissioning. So limits on the range of X*1*Cheb0(F) and 1*Y*Cheb0(F) can be established.

### Covariance matrix

A more detailed characterisation of the quality of the astrometric solution is presented by the covariance matrix of the solution parameters. The covariance matrix gives an insight in the dependencies of the individual parameters on each other. Of course each parameter should depend fully on itself, so a unit matrix would be the ideal covariance matrix. However, several of the parameters may depend on each other, such as the linear and second order moments of one coordinate. This is a normal situation. The covariance matrix in `astrom` is constructed such that the lines neighboring the diagonal should all have very small values. If not, two parameters, principally x and y related parameters, depend on each other. This should not be the case for a good astrometric solution.

In `astrom` the covariance matrix is normalised to 1000 to make a representable set of values visible on output. An example run would yield the following output:

```
Matrix:
 1000     0   989     0     0   984
    0  1000     0   842     0     0
```

```
 989    0  1000    0    0  951
   0  842    0 1000    0    0
   0    0    0    0    0    0
 984    0  951    0    0 1000
```

For good astrometric solutions the diagonal elements should all be 1000 and the lines neighboring diagonal should all be 0's.

### Residuals

The individual residuals between the positions of extracted objects and reference stars provides a good measurement of the quality of the astrometric solution. This is a deepening of the first quality control parameter which is only a global measure of these individual residuals. To assess the individual residuals a table of residual positional information is created as an output catalog element from the `astrom` program.

An example of such a residual information table is (This is a full description of the first few objects from the RESIDUALS table.):

```
#   1 DRa                 Right Ascension residual                        [Deg]
#   2 DDec                Declination residual                            [Deg]
#   3 Ra                  Right Ascension object                          [Deg]
#   4 Dec                 Declination object                              [Deg]
#   5 Xpos1               X position first object                         [Deg]
#   6 Ypos1               Y position first object                         [Deg]
#   7 Xpos2               X position second object                        [Deg]
#   8 Ypos2               Y position second object                        [Deg]
#   9 F1                  Field number first object                       [Deg]
#  10 F2                  Field number second object                      [Deg]
#  11 C1                  Camera number first object                      [Deg]
#  12 C2                  Camera number second object                     [Deg]
4.67403e-06 -1.98503e-05 148.896252 -20.732737 -1 -1 1539.47 23.6963 -1 0 -1 0
6.13633e-05 4.90936e-05 148.937179 -20.730045 -1 -1 960.152 65.9206 -1 0 -1 0
-6.7815e-06 3.59786e-05 148.985052 -20.729637 -1 -1 284.43 72.3624 -1 0 -1 0
-0.000305546 3.32008e-05 148.985371 -20.729634 -1 -1 284.43 72.3624 -1 0 -1 0
0.000115075 -6.66176e-06 148.986496 -20.728337 -1 -1 262.184 91.4185 -1 0 -1 0
9.21497e-05 2.76691e-05 148.967871 -20.724937 -1 -1 525.897 143.251 -1 0 -1 0
9.30769e-06 -3.25175e-05 148.924557 -20.724084 -1 -1 1139.5 154.804 -1 0 -1 0
-3.95176e-05 -1.28883e-05 148.863991 -20.723503 -1 -1 1996.41 163.188 -1 0 -1 0
-4.20375e-05 -5.40213e-05 148.866960 -20.721840 -1 -1 1954.53 187.785 -1 0 -1 0
```

To really inspect these residuals for specific inconsistencies one needs to make plots similar to those presented in the OAC report. In fact one should plot each residual information item as a function of all available other parameters to see if there are any systematic effects. Only if no such effects are found to be present, the astrometric solution will have passed this test.

Of course it is almost impossible to inspect for each and every pointing the large number of plots that can be created using this residuals table, therefore the global parameters `RMS` and `Max` residual are created to provide a switch for going to this more tedious inspection of the astrometric solution.

### Individual object statistics

As part of the application of the astrometric solution to each extracted object, each will have an associated set of information giving the statistical accuracy parameter: the RMS from statistical

part of the solution. This does not incorporate measurement errors.

An example run would give (Only a subset of the available column information from the OBJECTS table is presented here.):

```
#   1 Ra                  Right ascension object in world coordinates     [Deg]
#   2 Dec                 Declination object in world coordinates         [Deg]
#   3 RMS_RND             Random positional error astrometric solution    [Deg]
#   4 ERRA_IMAGE          RMS position error along major axis             [pixel]
148.872928 -20.733011 1.65109e-07   0.0186
148.956969 -20.734256 1.27054e-07   0.0255
148.896252 -20.732756 1.36571e-07   0.0124
148.951663 -20.732212 1.23105e-07   0.0218
148.868455 -20.731741 1.71571e-07   0.0172
```

For a comfortable limit to this parameter, providing a good segregation between correct astrometric solution and flawed ones, a limiting value of $< 0.3 arcsec$ would be best.

### 3.3.1 Astrometric calibration using overlap

#### Introduction

At the moment the OmegaCAM pipeline processes individual CCD separately in the astrometric calibration section of the pipeline. Only at the stage where the de-dithered image is created are the individual CCD's (parallel streams in the pipeline) synchronized.

For the purpose of deriving an astrometric solution using overlaps, the parallel threads have to come together at an earlier stage (just before the LDAC `astrom` program is called) and then disentangle after the derivation. The overhead in disk I/O is small because, for the astrometric solutions, only catalogs are used. Overhead is created because the astrometric solution runs on one CPU, while generally the data is spread over 32 CPU's that have local disk storage.

#### Software Implications

The astrometric calibration procedure (AstrometricCatalog.do_astrometry) will have to have a synchronisation point. This can be generated by separating the procedure into two consecutive procedures that are 'called' from the commanding make and by rearranging the top-level.

This way the astrometric calibration application (second procedural part) will be very similar to the photometric calibration application.

## 3.4 Quality control of the photometry

Group composition: Ronald Vermeij, Mark Neeser, Roberto Silvotti, Juan Alcala
Leader: Ronald Vermeij

tool: tool: + colour - colour plots

### 3.4.1 Catalog creation

Quality control measures (to be) taken in the creation of catalogs. The first two QC measures are taken during the catalog creation itself (it is part of the *make*), whereas the last item is more of a check after the fact (part of the *verify*).

#### Correctly identifying standard stars

The proper identification of standard stars on a frame obviously depends both on the quality of the astrometric calibration, as well as on the accuracy of the catalogued coordinates of the standard stars. Currently, the robustness of identification is improved by removing artifacts from the catalog before association with the standard star catalog (hot pixels). Also, whenever possible the catalogued coordinates of the standard stars are improved.

#### Removing sources that are 'unfit for purpose'

Before association, saturated sources, blended sources, and sources with clipped or otherwise de-formed apertures are removed from the catalog. This step serves the double purpose that it does not only lower the chance of a mis-identification, but it also ensures that 'corrupt' standard stars are removed before the final catalog enters the photometric pipeline. The cleaning of the catalog depends on the flags set by SExtractor.

#### Checking the quality of the aperture photometry performed

As far as the quality control for the aperture photometry is concerned, things are under development. The final catalog stores the MAG_BEST output from SExtractor. The MAG_BEST output is either a value derived from aperture photometry using Kron-type apertures (MAG_AUTO), or a corrected isophotal magnitude (MAG_ISOCOR). The choice for either one of these is determined by SExtractor based on the effects of neighbouring stars on the measured magnitude of the source.

To check the quality of the aperture photometry and to get a handle on the amount of flux that could be missing, a selection of suitable standard stars from the final catalog could be measured again by SExtractor but now using a set of apertures with increasing radii. The flux measured at the 'stabilization' point could then be compared with the value stored in the previously derived catalog. The results obtained from the whole ensemble of stars provides the quality check. (This sounds like a complicated and time-consuming business that should be delegated to a dedicated QC Task. Also, the number of standard stars used for this exercise should be limited. Reprocessing hundreds of standard stars this way would be too costly.)

### 3.4.2 Atmospheric extinction

Quality control measures (to be) taken in deriving the atmospheric extinction, and simple QC flags to be raised during monitoring.

#### Monitoring

In daily routine, the atmospheric extinction will be dealt with by the monitoring requirement of the OmegaCAM pipeline (req 562). This requirement produces a report containing a measurement of the extinction for every image of the polar field that goes into making it (at least three). This measurement is obtained by fitting a standard extinction curve to the data that has been simultaneously observed in every one of the four key-bands (composite filter). The QC measures/flags in place are (broken down in the processing steps involved):

1. the measured zeropoints for all the stars observed in one kwadrant of the composite filter are clipped for outliers before a single value for the extinction is determined from these. This clipping will also remove any sources from the pipeline that have mistakenly been identified as a standard star. After determining this value, the result is checked against a standard extinction curve in the *verify*. This is a kind of sanity check on the result.

Raising a QC flag at this point is only a first warning that something fishy might be going on with the atmosphere. This QC warning is only send to the log. The error associated with the derived extinction is, of course, also a QC parameter.

2. when for all the four kwadrants of the composite filter an extinction has been derived, an extinction curve is fit to the four points to derive the ultimate extinction for that particular moment in the night. The fit parameters are send to the log for perusal. These fit parameters can e.g. be used to detect an unusual amount of reddening in the *u'* band (note that in that case a QC warning would already have been raised in the previous step while processing the *u'* kwadrant of the composite filter.)

3. after having processed all the polar images as described above, an assesment is made of the quality of the night (QC = SHIFT_OF_CURVE_VARIED), of the stability of the overall transmission properties of the atmosphere (QC = SHAPE_OF_CURVE_VARIED), and of the validity of the assumed shape of the standard extinction curve (QC = INVALID_ASSUMPTION_FOR_SHAPE_OF_CURVE). These checks are done in the *verify* of the overall monitoring recipe and is done using the whole collection of derived values for the atmospheric extinction (which span a 2-dimensional space, in time and in photometric band). The QC-flags are stored in the DB together with the separate values for the extinction through the night, and the fitted shifts of the extinction curve.

It should be emphasized that many of the QC measures described deal with assesing the quality of the night, i.e. conclusions drawn from the whole bunch of values for the atmosperic extinction. The quality of *one* particular value of the atmospheric extinction is dealt with by the *verify* functionality of the appropriate child of the `BaseAtmosphericExtinction` class.

### 3.4.3 Zeropoint

The zeropoint for a given chip/filter combination is derived by combining the extinction results from the monitoring with the raw zeropoints contained in the catalog for that particular chip/filter combination. Note that these catalogs are derived from equatorial standard field data. The error bar on the resulting zeropoint serves as a QC parameter; this number should be within the specs as given in the requirements. For further QC, a comparison can be made with the zeropoint derived the day before. To improve the result for the zeropoint, the input raw zeropoint should be clipped. This again helps in removing mis-identified standard stars from the mix. It might also be interesting to make a residual plot for validation.

#### Atmospheric extinction revisited

Although the recipe that derives the zeropoint by default uses the results from monitoring the atmosphere, it is also capable of deriving the atmospheric extinction by itself in the 'classical' way, i.e. by combining two images of the same standard field observed at two different airmasses. For this modus operandi, the same QC measures for deriving the atmosperic extinction are in place as described in §3.4.2 under point 1.

### 3.4.4 Suggestions and comments

This space is for suggestions and comments from the community. Feel free to add them.

Comments from OAC:

1. In order to evaluate the quality of the photometric calibration, it would be usefull to know the number of standard stars actually used to derive the ZP and the relative RMS.

Moreover, even more important, we suggest to take also into account the color term of each standard star: we propose to make a fit of the standard stars in a colour-mag plane: MAG1-mag1=CT(MAG1-MAG2)+ZP1
where MAG1 and MAG2 are the tabulated magnitudes and mag1 is the instrumental one (see for example http://www.na.astro.it/oacdf/OACDFPAP/node7.html). Note that it is not necessary to use the instrumental colours (as we did in the OACDF). Note also that, although the fit can be done even using only one free parameter (ZP), it is much more convenient to assume two free parameters (ZP and CT). In this way CT can be compared with the "standard" values stored in the DB, giving the possibility to detect "strange" CT values and allowing also long-term trend analysis.

*The raw zeropoints that go into deriving the zeropoint are now sigma-clipped with a configurable sigma-clip level. Both the number of raw zeropoints (stars) that are present in the input catalog, and the actual number of raw zeropoints used after sigma-clipping are logged and stored in the database.*

2. We suggest to take also into account the color term of each standard star: we propose to make a fit of the standard stars in a colour-mag plane : MAG1-mag1=CT(MAG1-MAG2)+ZP1, where MAG1 and MAG2 are the tabulated magnitudes and mag1 is the instrumental one (see for example http://www.na.astro.it/oacdf/OACDFPAP/node7.html). Note that it is not necessary to use the instrumental colours (as we did in the OACDF). Note also that, although the fit can be done even using only one free parameter (ZP), it is much more convenient to assume two free parameters (ZP and CT). In this way CT can be compared with the "standard" values stored in the DB, giving the possibility to detect "strange" CT values and allowing also long-term trend analysis. *If color terms are needed, they are applied at the time the catalogs are created. A two-parameter fit to the ZP, therefore, is unnecessary. However, making a plot of (MAG1 - mag1) vs (MAG1 - MAG2) is an excellent idea for QC and should definitely be included in the system. (To be implemented by OAC.)* >>>>>>> 1.9

3. Concerning the usage of MAG_BEST, it seems that its usage is not recommended (please ask Emmanuel) and we think that aperture photometry is more appropriate in case of standard (i.e. relatively bright) stars. This is related to the curve of growth: for standard stars it is important to measure the entire flux before comparing the instrumental magnitudes with the tabulated ones, otherwise the ZP will not be accurate. Note also that, in any case, it will be sufficient to run SExtractor only once, with different parameters, and this should not take significant CPU time. *We will stick to aperture photometry for now using a fixed aperture (using MAG_APER from SExtractor). The aperture size to use will be determined by the overall PSF model derived for the 'input' science frame, and a configurable scaling factor. (To be implemented by RV in collaboration with USM.)*

4. Finally, as a final check on the photometric accuracy, we suggest to plot the stars from the final coadded Science images in a colour-colour plane and compare them with stellar tracks from models. Obviously this could be done only at the catalog level, combining data from different bands. *This is a QC task that has to be done in the backend of the system. It cannot be part of the photometric pipeline itself. (To be implemented by OAC.)*

### 3.4.5 The `inspect` methods

The catalogs created for use in the photometric pipeline and the final zeropoints derived from these can be viewed on screen. This is achieved by calling the `inspect` method of the objects that represent these two items. The results are shown in Figure 3.1.

**Making EPS files from the plots generated by `inspect`**

The `inspect` method always produces an output to screen. However, it is also possible to save these plots as an EPS file. To do this, one has to work from the `awe`-prompt.

For making plots of the catalogs, go to the `awe`-prompt and do (for example):

```
awe> from astro.main.PhotSrcCatalog import PhotSrcCatalog
awe> from astro.plot.PhotometryPlot import PhotcatPlot
awe> photcats = (PhotSrcCatalog.filter.name == '220')
awe> photplot = PhotcatPlot()
awe> photplot.plot_params.POSTSCRIPT_OUTPUT = 1
awe> for photcat in photcats:
...      photplot.plot(photcat)
```

where in the third line the database is queried for the catalogs, and in the fifth line the `PhotcatPlot` object is configured to save the plot to a file.

For making plots of the zeropoints, go to the `awe`-prompt and do (for example):

```
awe> from astro.main.PhotometricParameters import PhotometricParameters
awe> from astro.plot.PhotometryPlot import PhotomPlot
awe> photoms = (PhotometricParameters.filter.name == '220')
awe> photplot = PhotomPlot()
awe> photplot.plot_params.POSTSCRIPT_OUTPUT = 1
awe> for photom in photoms:
...      photplot.plot(photom)
```

where in the third line the database is queried for the zeropoints, and in the fifth line the `PhotomPlot` object is configured to save the plot to a file.

*Note 1 : for this to work properly, the objects retrieved from the database should, of course, be fully qualified. That is, their* `make` *method should have been called.*

*Note 2 : the* `inspect` *methods themselves actually delegate the plotting to the same* `PhotometryPlot` *class. In that case, however, the plot object is configured to put the result on screen.*

## 3.5 Quality control of the image pipeline

Group composition: Ewout Helmich, Edwin Valentijn, Mike Pavlov, Roberto Silvotti, Jan Snigula
Leader: Ewout Helmich
tool: PSF tool: general small windows - flatness

### 3.5.1 General ideas

The established way of checking the overall quality of the image pipeline is to check for PSF variations over the field of view before and after the regridding step in the pipeline. The check could be done in several ways:

1. Use existing SExtractor parameters as a measure of PSF for whichever stars are present in regions of an image and associate those from the ScienceFrame (before regridding) and the RegriddedFrame (after regridding).

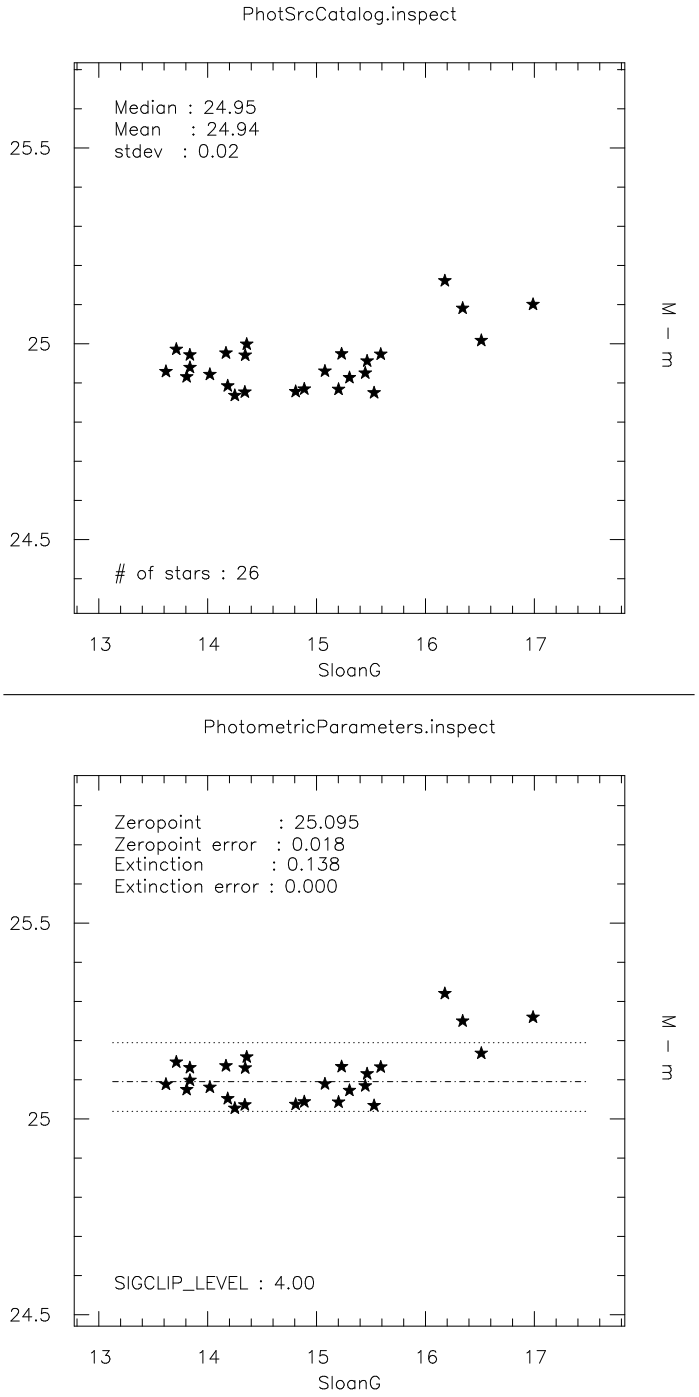



Figure 3.1: The results of calling the *inspect* methods of PhotSrcCatalog (top panel) and PhotometricParameters. These two classes represent the start and end point of the photometric pipeline, respectively.

2. Make PSF models of cutouts of the same regions in the ScienceFrame and RegriddedFrame (by using PSFEx through the PSFModel class ? – RV).

A common aspect is that in order to get PSF Anisotropy information (PSF variations as a function of X/Y position in the focal plane) a subdivision of each image is necessary, either in the form of defined areas, or in the form of cutouts. Method (i) should be preferable computationally. Only stars should be taken into account. This complicates matters, because a star-galaxy separation is necessary.

In addition a check on the flatness of the background of the ScienceFrames could be useful.

### 3.5.2 Comments from OAC (Mario & Roberto)

Concerning the PSF variations, we do not see particular reasons why the PSF should change after the regridding. A wrong astrometric solution would produce not perfectly overlapping sources, but the shapes of sources in individual frames could be not affected at all. Troubles in the astrometric solution would be therefore much more likely detected in the coadded image. We propose to proceed as follows: - Measure the PSF in each ScienceFrame: from all ditherings, derive an average PSF and RMS (¡PSF¿, RMS), and possibly min(PSF) and max(PSF); - Measure the average PSF in the coadded image ¡PSF_coadd¿; - Compute the value r=abs(¡PSF_coadd¿-¡PSF¿)/RMS: the result is acceptable if e.g. r¡=3 and min(PSF) ¡= ¡PSF_coadd¿ ¡= max(PSF). A more complex possibility would be to check the PSF in different CCDs separately or as a function of the X/Y position: in this case the position in each ScienceFrame should be mapped to the coadded image.

## 3.6 Quality control of the PSF

Group composition: Edwin Valentyn, Roberto Silvotti, Mark Neeser, Koen Kuijken, Ronald Vermeij
Leader: ???
tool: tool:

# Chapter 4

# Development

In the next several sections we will describe some key concepts in the implementation of the Astro-WISE library.

We assume a reasonable amount of familiarity with Python. In particular with the Python notation (significance of white space), basic program control statements (`if...elif...else`, `for...in`), data structures (lists, tuples, dictionaries), defining functions and classes (`def`, and `class`), instantiating objects, importing modules, and the meaning of dots in names. If you are not familiar with one or more of these, you may want to have a look at one of the introductions to Python, found at:

http://www.python.org/doc/Newbies.html

This document does not aim to give a comprehensive description of the Astro-WISE library. Documentation for the library can be obtained using the pydoc documentation server. To browse the documentation, start the server from the unix commandline:

```
>pydoc -p 8080
```

and point your browser to:

http://localhost:8080/

If the Astro-WISE library (i.e., the directory `awe`) is in your `PYTHONPATH`, then you should see a link, near the top of the page, to the `astro` package containing the astronomy specific modules, and the `common` package containing common modules. You can also simply point your browser to: http://doc.astro-wise.org/.

## 4.1 Key concepts

### 4.1.1 Persistent classes

Astro-WISE data processing is performed by executing methods on instances of **persistent classes** (persistent objects). This means that all processing results are recorded in the persistent attributes of these objects, and the persistence mechanism ensures that these results are stored in the data base. (See Chapter 6 for further details)

#### targets, dependencies, make

At the highest level, Astro-WISE data processing can be understood in terms of **targets**, **dependencies** and **make**. To illustrate these concepts, let's start with an example:

```
 1 # example1.py
 2 from astro.main import BiasFrame
 3 from astro.main import RawFrame
 4
 5 def makebias(raw_bias_names, bias_name):
 6    '''Make a master bias
 7    raw_bias_names -- a list of names of raw bias FITS files
 8    bias_name -- the name of the master bias FITS file
 9    '''
10    bias = BiasFrame.BiasFrame(pathname=bias_name)
11    for name in raw_bias_names:
12        raw = RawFrame.RawBiasFrame(pathname=name)
13        bias.raw_bias_frames.append(raw)
14    bias.make()
15    return bias
```

The example defines one function (`makebias`) to make a bias frame. It takes a list of the names of raw bias files and the name of the output file as arguments and returns a `BiasFrame` object. This example illustrates how a user would use the library to process his own data. For example, from the Python command line:

```
>>> from example1 import makebias
>>> bias = makebias(['ima01.fits', 'ima02.fits', 'ima03.fits'], 'bias.fits')
```

Let's go over this piece of code line by line (note that Python source code does not include line-numbers):

**lines 2-3** import two modules (`BiasFrame` and `RawFrame`) from the package `astro.main`

**line 5** define a function (`makebias`) taking two arguments (`raw_bias_names` and `bias_name`)

**lines 6-9** the documentation for the function.

**line 10** create a `BiasFrame` object. The `BiasFrame` class is defined in the `BiasFrame` module which we imported in line 1. The `BiasFrame` object is defined with one argument; the name of the bias image (`bias_name`)

**line 11** loop over the names contained in the list `raw_bias_name`, assign each name to `name`

**line 12** create a `RawBiasFrame` object from each name

**line 13** add the raw bias object to list of input frames (called `raw_bias_frames`) of the master bias object `bias`

**line 14** "make" the master bias object. By calling the method `make`, the processing necessary to create the master bias data is executed.

**line 15** we are ready, and return the result.

this description probably doesn't add much to your understanding of the example. If you don't have the feeling that the description and the example are really equivalent, you should probably first try to get to know Python a little bit better.

The example illustrates the fundamental steps in processing data:

1. create the **target** object (line 10)
2. assign objects to the **dependencies** (lines 11-13)
3. execute the **make**-methods (line 14)

In this case the target (a `BiasFrame` object), only depends on the raw data (`RawBiasFrame` objects). In other cases the target may also depend on additional objects, including calibration data and processing parameters. For example, to reduce science data we need a considerable number of other objects besides the raw data, i.e. all calibration objects.

Note that a `BiasFrame` object is not a FITS file, it is an entity that describes a FITS file and may execute a number of operations on FITS files. All pipeline processing is done by calling methods on these kinds of objects.

The following methods can be used to inspect targets:

`is_made()` return 1 if `make()` has been executed on a target and 0 otherwise. The value of the special attribute `process_status` is inspected to determine this.

`set_made()` indicate that the target has been made. This is usually called from the `make()` method. The value of the special attribute `process_status` is updated to record this.

`get_dependencies()` returns a list of attribute names on which the target depends.

The case of making a bias is probably the simplest example. Other examples can be found by looking at the other recipes in the directory `filerecipes` of the library. Have a look at these recipes, including the Bias recipe, to see that all look extremely similar to this example.

### 4.1.2 Verification and quality control

In order to verify the results of the data processing, makable objects (will) have `verify()`, `compare()`, and `inspect()` methods. These methods implement basic quality control mechanisms.

**verify** The `verify()` method inspects the values of various attributes of the object to see if these are within the expected range for that object. The purpose of this method is mostly to perform sanity-checks on measured results. It is assumed that the required measurements (for example image statistics) are done during data reduction (i.e. while executing `make()`), and stored in persistent attributes.

**compare** The `compare()` method is used for default trend analysis. This is done by comparing with a previous version of the same object (last weeks bias, for example). This may be as simple as comparing attribute values, but may also involve more complex computations (e.g., subtracting the two images, and analysing the residuals)

**inspect** Visual inspection of the data remains a powerful tool in quality control. The inspect() method provides the mechanism to record the results of visual inspection for posterity.

The `make()` and quality control methods set a flag in the `processing_status` attribute to record if these methods have been run. The following methods are available to inspect the processing status of makable objects:

`is_verified()` returns `1` if `verify()` has been succesfully executed, `0` otherwise.

`is_compared()` returns `1` if `compare()` has been succesfully executed, `0` otherwise.

`is_inspected()` returns `1` if `inspect()` has been succesfully executed, `0` otherwise.

The quality control methods record their results by setting quality control flags. These quality control flags are given in the class definition using the `QCFlag()` property. Flags are stored in the special attribute `quality_flags`, using bit masking. The following methods are available to inspect these flags:

`get_qcflags()` This method returns a list of the names of all possible flags.

`get_qcflags_set()` This method returns a list of those flags that have been set. If no flags have been set, this returns an empty list.

The following class defines two quality control flags and a `verify` method that may set these flags.

```
class MyScienceResult(DBObject, ProcessTarget):
    TOO_MANY_GALAXIES = QCFlag(0, 'This is a bad sign')
    TOO_FEW_STARS = QCFlag(1, 'This is a really bad sign')

    def verify(self):
        if self.galaxy_count > 1e6:
            self.TOO_MANY_GALAXIES = 1
        if self.star_count < 100:
            self.TOO_FEW_STARS = 1
        self.set_verified()
```

Here is an example session using this class

```
>>> m = MyScienceResult()
>>> m.galaxy_count = 10000
>>> m.star_count = 1000
>>> m.verify()
>>> m.is_ok()
1
>>> m.star_count = 10       # simulate a problem
>>> m.verify()
>>> m.is_ok()
0
>>> m.get_qcflags_set()
['TOO_FEW_STARS']
>>> m.galaxy_count = 10000000
>>> m.verify()
>>> m.TOO_MANY_GALAXIES
1
>>> m.get_qcflags_set()
['TOO_MANY_GALAXIES', 'TOO_FEW_STARS']
>>> m.quality_flags        # bits 0 and 1 were set
3
```

## 4.2 The Astro-WISE class hierarchy

At the heart of the Astro-WISE library lies the persistence class hierarchy, These classes provide the definition of the object that are operated on when processing and analyzing data. Since

these classes are persistent, the results of these operations will be automatically saved in a database.

Figure 4.1 gives an overview of this class hierachy. Remember that derived classes inherit attributes and methods from base classes. For example, `BaseFrame`, the base class for all classes representing image data, inherits from `DataObject`, which represents all objects that represent some sort of data on disk.

The following key classes are defined in the hierarchy

**DBObject** All objects deriving from DBObjects are persistent. Hence, all these objects can be saved and retrieved from a database

**DataObject** All DataObject objects have associated bulk data (FITS files, catalogs) which can be stored and retrieved from a central data server

**BaseFrame** All objects derived from BaseFrame describe the contents of associated FITS images and have methods that operate on these images

**Catalog** All Catalog objects describe LDAC catalogs and define methods that operate on these catalogs

**Config** These objects store the contents of the configuration files of external packages (SExtractor, SWarp, LDAC)

All classes that define a make method (marked by asterisk in Fig. 4.1) also derive from the mixin[1] class `ProcessTarget`

**ProcessTarget** Classes that also derive from this class have `make()` methods and quality control flags (`QCFlag()` properties)

[1] A 'mixin' class is a class that adds behaviour to another class in a derived class using multiple inheritance


Figure 4.1: The class hierarchy of Astro-WISE. Asterisks (*) mark object that have a `make()` method

# Chapter 5

# Database Tasks

## 5.1 Setting up the database for general use

Below the steps are described that need to be taken to set up the database. It is assumed that the pipeline is already installed according to the README (see also CVS HOW-TO §7.3) and that the environment can be started with the `awe` command.

- In your `~/.awe/Environment.cfg` set

  ```
  database_name :  <tnsname of your database here>
  database_user :  <name of the database account you created previously>
  ```

  To test this you should start `AWE` and do

  ```
  import common.database.DBMain
  ```

  You should get a prompt asking for your database password. If you are able to connect you can continue with the next step.

  You may also enter your password in the `~/.awe/Environment.cfg` in the following way

  ```
  database_password:  <your Oracle password>
  ```

  but for obvious reasons this is discouraged.

- Create the `AWOPER` database account, the `AWUSER` and `AWREADONLYUSER` roles, as well as the `AWNORMAL` and `AWLIMITED` profiles. In addition the AWCONTEXT context is enforced and the required `AWEUSER`, `AWEPROJECTS`, `AWEPROJECTUSERS` tables are created.

  ```
  awe awe/common/toolbox/dbawoper.py
  ```

- Create the shared `AWOPER` schema for all database users using

  ```
  awe awe/common/toolbox/dbimportall.py
  ```

- Create the Package Header and Body for the Oracle interface to the htm library using

```
awe awe/common/toolbox/dbawutil.py
```

Note that you MUST have compiled and installed the _ohtm library in the Oracle tree.

- Grant `SELECT/INSERT/UPDATE` permission on the `AWOPER` schema to the `AWUSER` role

```
awe awe/common/toolbox/dbgrants.py
```

If new persistent classes are defined you should run this script as well to make sure that users have access to the new tables.

- Add a new user to the database. The username should consist of the first initial followed by the complete surname of the user.

```
awe awe/common/toolbox/dbnewuser.py mjackson
```

Note that this will create a database account with `AW` as a prefix. In the above example the database account would be called `AWMJACKSON`.

## 5.2 Keeping database synchronized with STABLE sources

It can occur that someone wants to change the definition of a persistent class. Unlike introducing new persistent classes, changing a persistent class requires intervention by the central database administrator. The reason is that users may still be using the persistent class as it was before the change. The problem is that the class definition in Python and in the database have to match each other. This means that whenever the Python class is changed the database has to be changed accordingly. Because the database is used by multiple users, all of these users have to adopt the new Python class definition in unison with the database definition. The process to handle this problem requires human intervention at certain stages and is completely described in this section.

- An improved persistent class has been tested in a separate schema and has been committed. After a while these changes are considered to be the STABLE version.
- When a change to a persistent class is considered STABLE, the main database administrator has to be contacted. The information that needs to be given is the revision number of the source file that are involved.
- The main database administrator contacts the administrators of all other databases in the federation that a database type evolution is bound to happen.
- The main database administrator waits until *all* administrators of the databases in the federation have responded and agreed a time when they can update the database schema. Alternatively, they can respond that they disable automatic updates of the CVS sources.
- The main database administrator checks out all (STABLE) sources except for those that are meant to change. In the case of the sources that are to be changed, the revisions that have been given earlier are checked out. This will provide the database administrator with an environment that will be exactly like the next STABLE environment that is being created. The `PYTHONPATH` has to be set to this checked out version.

- With a Python environment that represents the upcoming STABLE environment the main database administrator evolves the database schema. The goal is to make sure that the Python and database environment match. All type evolution statements and all other issues that come up are gathered by the main database administrator during the process. The collection of SQL statements and things to watch out for is then sent to the local database administrators.
- The local database administrators now have to set up an environment in the same way as explained for the main database administrators. In this environment the local database administrators can evolve their local schema. They can use the SQL statements and remarks as sent to them by the main database administrator.
- Once a local database administrator has succesfully evolved the schema, a confirmation needs to be sent to the main database administrator. Only when all confirmations have arrived the STABLE version can be updated.
- The main database administrator attaches the STABLE tag in CVS to the revisions as they were supplied. This should only be done if all local database administrators have reported successful schema evolution. The command to be used is:

  `cvs tag -r` *revision_number source_file*

## 5.3 Database Type Evolution

### 5.3.1 Database Type Evolution

Persistent classes and attributes are defined in Python. The SQL definition in Oracle is derived from the Python definition. This means that whenever a change is made to a persistent class the corresponding SQL definition has to be changed accordingly. This requires manual intervention and details are given in this section on what do.
Always make a backup first. If you do not have set up `RMAN` you can shutdown the database and make an off-line backup. Otherwise log in to the Recovery Manager as follows

```
$ORACLE_HOME/bin/rman target sys@aw98
```

From the `RMAN` prompt type

```
RMAN> backup database plus archivelog;
```

### 5.3.2 Overview

There are different categories of database type evolution. Some of these require a simple SQL statement, but most require close attention. In general type evolution requires extreme care, especially when changes are made in a database that is populated. Only when a backup is available it is possible to retrieve types or attributes that have been removed.

All database type evolution operations fall in one of three categories: *Adding*, *Removing*, *Changing*. Each operation can be applied to a persistent class or to a persistent attribute. The simplest operation is *Adding*, the most dangerous one is *Removing* and the most complicated one is *Changing*.

### 5.3.3 The SQL representation of persistent Python class

For the following detailed type evolution descriptions it is useful to keep in mind that for each persistent class "Demo" in Python an Oracle TYPE called "Demo$", an object TABLE called "Demo" and a VIEW called "Demo+" exist.

When adding or changing attributes it is necessary to know who their Python type translates into an SQL type. The module astro.database.oraclesupport contains a dictionary called typemap for this purpose.

For list attributes an additional type in SQL is created which is a nested table of the type of the list attribute. If "Demo" has a list attribute which is defined as p = persistent('', int, []), then "Demo$p" is a TYPE defined as TABLE OF SMALLINT.

Link attributes in Python are represented in SQL by a REF to the type the attributes links to and link list attributes are represented by a type that is defined as a TABLE OF REF ¡*type-being-referenced*¿.

### 5.3.4 Finding information about the SQL types, tables and views

There are several system views in the database that can be use to inspect existing definitions of structures such as types, tables and views. To find the definition of a structure in SQL*Plus the describe command can be used.[1]

The USER_OBJECT_TABLES view contains all the object tables in the users schema. Likewise, USER_VIEWS contains all views and USER_TYPES contains all types. The USER_TYPE_ATTRS view contains all attributes and their definition for all types.
To get the names of all types that contain Demo

```
SELECT TYPE_NAME FROM USER_TYPES WHERE TYPE_NAME LIKE '%Demo';
```

The USER_TYPES views also has a column SUPERTYPE_NAME with the name of the type from which the TYPE_NAME is derived.

### 5.3.5 Adding a persistent class

To add a persistent class import the class in Python as the AWOPER database user that owns the schema. To make the new class visible to other database users the Toolbox/dbgrants.py script needs to be run. The script will run as AWOPER and ask for its password. Note that no manual SQL is required.

### 5.3.6 Removing a persistent class

Check that no classes are derived from the class you are trying to remove. If classes are derived from the class or if attributes in other classes refer to instances of the class you cannot use the following commands to remove the database type. Instead you'll have to follow the procedure described in §5.3.11.
To remove a persistent class "Demo" use the following commands in the specified order.

```
DROP VIEW "Demo+";
DROP TABLE "Demo";
DROP TYPE "Demo$";
```

After these elements have been dropped you have to check whether "Demo$" has list attributes which have to be removed. The types for such attributes have to be dropped as well using the

[1] Use help describe from the SQL*Plus prompt

DROP TYPE command. The names of the types of these attributes, e.g. for "Demo$", can be found with

```
SELECT TYPE_NAME FROM USER_TYPES WHERE TYPE_NAME LIKE 'Demo$%;'
```

### 5.3.7 Adding persistent attributes to a class

To add persistent attributes to a class you need to know their name and their type. If attributes x and y are added with

```
x = persistent('This is x', int, 3)
y = persistent('This is y', float, 4.2)
```

then the following command will add these attributes to the type in the database.

```
ALTER TYPE "Demo$" ADD ATTRIBUTE ("x" SMALLINT, "y" DOUBLE PRECISION) CASCADE;
```

Then the attributes of existing objects have to be given their default values.

```
UPDATE "Demo" SET "x"=3, "y"=4.2;
```

### 5.3.8 Removing presistent attributes from a class

Removing one or more attributes is perhaps the simplest, but not less hazardous, operation of all. To remove x and y it is sufficient to execute

```
ALTER TYPE "Demo$" DROP ATTRIBUTE ("x", "y") CASCADE;
```

*Be careful to also drop any list types that have been defined in the given type! See also §5.3.3.*

### 5.3.9 Renaming a persistent attribute

To rename a persistent attribute the procedures described in §5.3.7 and §5.3.8 are combined. In the next example the name of an attribute is changed from x to z

```
ALTER TYPE "Demo$" ADD ATTRIBUTE "z" SMALLINT CASCADE;
UPDATE "Demo"SET "z"="x";
COMMIT;
ALTER TYPE "Demo$" DROP ATTRIBUTE "x" CASCADE;
```

### 5.3.10 Changing the type of a persistent attribute

Changing the type of a persistent attribute is done in different ways for different types. The basic procedure is however always the same.

- First add a dummy attribute with the eventual type for the attribute. This is done, like for any other attribute, following the steps in §5.3.7.
- The next thing to do is to copy the value old attribute to the new attribute while converting it to the new type. Depending on the types that are involved, the conversion can be simple or complicated. The guideline is the purpose of the typechange and the person requesting the type change will know best what this purpose is.
- After a succesful copy the old attribute can be removed according to the outline given in §5.3.8.

- Before the dummy attribute can be removed, the attribute whose type is changed needs to be added with its final type, as explained in §5.3.7.
- Now the dummy attribute has to be copied to the attribute for which the type has changed. This can be done with a simple `UPDATE` statement.
- Finally the dummy attribute can be removed using the procedure shown in §5.3.8.

### 5.3.11 Moving a persistent subclass to a different parent class

When a persistent subclass needs to be moved to a different place in the class hierarchy a combination of many of the previously called techniques is needed.

### 5.3.12 Error messages

`ORA-22337: the type of accessed object has been evolved` Stop the current SQL session and start a new one.

# Chapter 6

# Persistency Interfaces

This is a direct copy of Appendix A of the Astro-WISE Architectural Design document and is use to give a more detailed explanation of how the Astro-WISE system works at its lower levels.

## 6.1 Introduction

This chapter describes the specification and Python implementation of persistent objects on top of a relational database back end. The aim of this implementation is twofold:

1. Provide a transparent mapping from a definition of a persistent class to a table in a relational database, preserving inheritance relationships, and allowing attributes to refer to other persistent objects.

2. Provide a native Python syntax to express queries, and leverage the advantages of the relational model (SQL) when using persistent objects.

In this paper we will first introduce a number of concepts from Object Oriented Programming (OOP) and Relational Database Management Systems (RDBMS), in order to clarify the problem we wish to solve. We will then provide the specification of the database interface provide by the Astro-WISE prototype. Finally, we will clarify some of the implementation issues addressed by the current prototype.

## 6.2 Background

### 6.2.1 Object Oriented Programming

It is difficult to give a meaningful definition of "object". However, the following "definition" introduces some intimately related terms that will be used throughout this document:

**object** An object is something that comprises **type**, **identity** and **state**. The type of an object, specifies what kind of object it is, specifically what kind of behavior the object is capable of. The identity is what distinguishes one object from another. The state of an object specifies the values of the properties of the object.

In Object Oriented Programming (OOP) we have an operational definition of objects:

**object** An object is an **instance** of a **class**, and **encapsulates** both data and behavior

The class defines what operations (**methods**) can be performed on its instances, and what **attributes** those instances will have. In general 'class' and 'type' are synonymous, as are 'instance' and 'object'. That is, when we talk about the type of an object we mean the class of which it is an instance.

It is important to note that the values of the attributes of an object will themselves be objects, although most programming languages distinguish between (instances of) primitive data types (integers, strings, etc) and instances of classes.

**Inheritance** is the mechanism by which one can use the definition of existing classes to build new classes. A child (derived) class will inherit behavior from its parent (base) class. In defining the child class the programmer has the opportunity to extend the child class with new methods and attributes, and/or modify the implementation of methods defined in the parent class. However, the child class is expected to conform to the interface (specification) of the parent class, to the extent that instances of the child class can behave as if they are instances of the parent class. In particular it is expected that procedures taking an object of a base type as argument, should also work when given a derived type as argument. This key property of objects is called **polymorphism**

### 6.2.2 Persistency

An object is said to be **persistent** if it is able to 'remember' its state across program boundaries. This concept should not be confused with the concept of a program saving and restoring its data (or state). Rather, persistency, implies that object identity is meaningful across program boundaries, and can be used to recover object state.

Persistency is usually implemented by an explicit mapping from (user-defined) object identities to object states and by then saving and restoring this mapping. However, this implementation assume that the object identity of the object one is interested in can be independently and easily obtained. For many applications this is not the case. On the contrary, one usually has a (partial) specification of the state, and are interested in the corresponding objects that satisfy this specification. That is, many interesting applications depend on a mapping of a partially specified object state to object identity (and then to object). This is the domain of the relational database.

### 6.2.3 Relational Databases

A relational database management system (RDBMS) stores, updates and retrieves data, and manages the relation between different data. A RDBMS has no concept of objects, inheritance and polymorphism, and it is therefore not a-priory obvious that one would like to use such a database to implement object persistence. However, using the following mapping

| | | |
|---|---|---|
| **type** | $\longleftrightarrow$ | **table** |
| **identity** | $\longleftrightarrow$ | **row index** |
| **state** | $\longleftrightarrow$ | **row value** |

it is (hopefully) obvious that one might, at least in principle, implement object persistency using a relational database. That is, given a type and object identity, one can store and retrieve state from the specified row in the corresponding table.

Relational databases provide a powerful tool to view and represent their content using structured queries. It would be extremely useful if we were able to leverage this power to efficiently search for object whose state matches certain criteria. Special consideration has to be given to inheritance in this case.

Assume, for example, that we define a persistent type `DomeFlatImage`, derived from a more general type `FlatfieldImage`. A query for all R-band flatfield images, should result in a set

including all R-band domeflat images. This behavior of queries is what inheritance means in a relational database context. Hence, a query for objects of a certain type maps to queries (returning row indices/object identities) on the tables corresponding to that type, and all of its subtypes. The results of these queries are then combined in to a single set of all objects, of that type or one of its sub types, that satisfy the selection.

## 6.3 Problem specification

The implementation of the interface (should) address(es) the following issues:

**defining a persistent class** Defining a persistent class (type), will give its instances the property of being persistent. The class definition should provide sufficient information about the attributes (possible state) of the objects to build the corresponding database table. This table should be present in the database when the first object of the class is instantiated. Presently, this is achieved by dynamically creating the table (if it doesn't yet exist), when processing the class definition[1]

**retrieving state of persistent object** Instantiating a persistent object with an existing object identity should result in retrieval of state from the database.

**saving state of persistent objects** Persistent objects, whose state has been modified, should save their state to the database before they cease to exist.

**references** persistent objects will contain references to (read: instances of) other persistent objects. Care has to be taken that instantiation of a persistent object does not recursively instantiates all objects it refers to. Only when the attribute corresponding to the reference are accessed should the corresponding object be instantiated.

**expressing selections** It should be possible to express selections of the form

$$\{x|x \in X \wedge (x.attr1 \in A \wedge x.attr2 \in B \vee x.attr3 \in C...)\} \tag{6.1}$$

i.e.: the set of all objects of type $X$ whose attributes have certain properties. This set should be translated in to an SQL query to the database, and result in an iterable sequence of objects satisfying the selection.

In addition, the following issues need to be addressed, though not necessarily by the interface to persistent objects.

**managing database connections** The interface does not specify how or when the database connection is established.

**transactions** The interface doesn't specify if and how transactions are implemented

**efficiency** No effort has yet been made to maximize performance and/or scalability. Initial efforts has focussed on a demonstration of technology and simplicity of implementation.

## 6.4 Interface Specification

In this section we describe how to implement and use persistent objects, using the interface defined in the Astro-WISE prototype. This section includes Python source code fragments. For those not familiar with Python we advise that they have a look at the main web site at http://www.python.org/ and at the Python tutorial at http://docs.python.org/tut/tut.html

[1] This implementation neatly avoids the problem of having to maintain both the class hierarchy and the corresponding database schema

### 6.4.1 Persistent classes

Persistent objects are instances of persistent classes, which specify explicitly which attributes (properties) are saved in the database. We call these attributes persistent properties. Executing a program defining

#### Defining persistent classes

A new persistent class is defined by deriving from an existing persistent class, or by deriving from the root persistent class DBObject. E.g.:

```
#example1.py
from common.database.DBMain import DBObject
class A(DBObject):
    pass
class B(A):
    pass
```

specifies two persistent classes (A and B). Neither of them extends their parent classes, so instances of A and B will behave exactly like instances of DBObject.

#### Defining persistent properties

A persistent property is defined by using the following expression in the class definition:

```
prop_name = persistent(prop_docs, prop_type, prop_default),
```

where, `prop_name` is the name of the persistent property, and `persistent` is constructed using three arguments: the property documentation, the type of the property, and the default value for the property respectively. For example:

```
#example2.py
from common.database.DBMain import DBObject, persistent
class Address(DBObject):
      street = persistent('The street', str, '')
      number = persistent('The house number', int, 0)
```

This program defines a persistent class 'Address', with two persistent properties, 'street' and 'number', of type str(ing) and int(eger) respectively.

We distinguish between 5 different types of persistent properties, based on the signature of the arguments to `persistent()`

**descriptors** If the type of the persistent property is a basic (built-in) type, then we call the persistent property a descriptor. Valid types are: integers (int), floating point numbers (float), date-time objects (datetime), and strings (str).

**descriptor lists** Persistent properties can also be homogeneous variable length arrays of basic built in types, called descriptor lists. Valid types are the same as those for descriptors. descriptor lists are distinguished from descriptors by the property default. If the default is a Python list, the the property is descriptor list, else it is a simple descriptor.

**links** Persistent objects can refer to other persistent objects. The corresponding properties are called links. If the type of the persistent property is a subclass of DBObject, then the property is a link.

**link lists** Persistent properties can also refer to arrays of persistent objects, in which case they are called link lists. Link lists are distinguished from links by the property default. If the default is a Python list, the the property is link list.

**self-links** A special case of links are links to other objects of the same type. These are called self-links. if no type and default are specified for the call to `persistent`, then the property is a self-link.

#### Keys

It is possible to use persistent properties as alternative object identifiers for the default object identifier (`object_id`). Only descriptors can be used as keys. Keys are alway unique and indexed.

The special attribute `keys` contains a list of attributes and tuples of attributes tuples, each specifying one key. For example:

```
#example3.py
class Employee(DBObject):
    ssi = pesistent('Social Security Number', str, '')
    name = persistent('Name', str, '')
    birth = persistent('Birth data', datetime, None)
    keys = [('ssi',), ('name', 'birth')]
```

In this example `ssi` is a key. The pair of attributes `('name', 'birth')` is also a key.

#### Indices

Databases use indices to optimize queries. It is possible to specify which persistent properties should be used as indices.Only descriptors can be used as indices.

The special attribute `indices` contains a list of attributes which should be indexed. E.g.:

```
# example4.py
class Example(DBObject):
    attr = persistent('A measurement', float, 0.0)
    indices = ['attr']
```

### 6.4.2 Persistent Objects

Having specified persistent classes, we can now use these classes to instantiate and manipulate persistent objects. In most respects these objects behave just like instances of ordinary classes. There are two exceptions: special rules for instantiation, and special rules for assigning values to persistent properties.

#### Object instantiation

We can distinguish between three different modes of instantiating a persistent object.

**New** We are creating a new persistent object, for which the `object_id` needs to be generated. This can be accomplished by instantiating an object without specifying `object_id`.

**Existing** We are using an existing object. If the object has already been instantiated in this application we want a copy to its reference, otherwise we want an instance, whose state has been retrieved from the database. This can be accomplished by instantiating the object with an existing `object_id`.

**Transient** it may be useful to build an object of a persistent type that is not itself persistent (whose state, will not be save to the database). This can be accomplished by instantiating the object with an `object_id` equal to 0 (zero)

or, in code:

```
a = MyObject()                # A new instance of MyObject
b = MyObject(object_id=1000) # An existing instance of MyObject
c = MyObject(object_id=0)    # A transient instance of MyObject
```

In practice, objects are rarely instantiated with an explicit `object_id`, because, we will generally not know the `object_id` of the objects we are interested in. Rather, objects are instantiated using keys or as the result of a query (see below)

Instantiating an object using a key, will result a restored object (if an object of that key did exist before) or a new object. In code:

```
class Filter(DBObject):
    band = persistent('the band name', str, '')
    keys = ['band']

f = Filter(band='V')       # The V-band filter
```

**Assigning values to properties**

Python is a dynamically typed language. This means that there is no such thing as the type of a variable. However, since database values (e.g. columns) are statically typed, the interface performs type checks when binding values to object attributes. The type is specified in the property definition, as outlined earlier.

### 6.4.3 Queries

In order to represent selections in native Python code, we have defined a notation that is based on the idea that a class is in some sense equivalent to the set of all its instances. To illustrate the concept, let us give a few examples.

Given a persistent class `X` with persistent property `y`, then the expression

```
X.y == 5
```

represents the set of all instances `x` of `X`, or subclasses of `X`, for which `x.y==5` is true. To obtain these objects the expression needs to be evaluated, which can be done by passing it to the `select` function, which returns a list of objects satisfying the selection.

Given a class `X` with a descriptor `desc`, a descriptor list `dsc_lst`, and a link `lnk`, then

```
select(X.desc > 2.0 && X.dsc_lst[2]=='abc' and X.lnk.attr == 5)
```

will return a list of instances `x` of `X`, or subclasses of `X`, for which

```
x.desc > 2.0 and x.dsc_lst[2]=='abc' and x.lnk.attr == 5
```

is true.

### 6.4.4 Functionality not addressed by the interface

New persistent objects may have an owner. The owner can defined as the user running the process in which the persistent object is created or it can be defined as an attribute of the persistent object. In either case, it is the responsibility of the implementation of the interface for a certain database to handle ownership of persistent objects.

# Part II

# HOW-TOs

# Chapter 7

# Getting Started

## 7.1 HOW-TO Get Started

To start using the Astro-WISE Environment, AWE, a few initial steps need to be taken. Access is required to the AWE database, to the AWE software and to the AWE dataservers.

### 7.1.1 Access to the AWE database

A database account is needed in order to use the AWE system. To get a database account, ask your local AWE representative[1] to create one for you. Your database username will consist of AW, followed by your first initial, followed by your surname.

### 7.1.2 Preparing the Astro-WISE Environment

In order to automatically log in to the database when you start up AWE you need to create a directory .awe in your home directory as follows:

```
cd
mkdir .awe
cd .awe
```

Now create a file here called **Environment.cfg** with the following content:

```
[global]
database_user     : <your database username>
database_password : <your database password>
```

Make sure only you can read the contents of the file:

```
chmod a-rwx,u+rw Environment.cfg
```

At OmegaCEN/Kapteyn Astronomical Institute, the AWE software is set up by executing the appropriate one of the following commands from the shell prompt:

```
# At the Kapteyn Institute in Groningen
...]$ module add awe
```

Tip: you can add this source command to your .cshrc or .bashrc file to have the command executed automatically for future awe sessions.

[1] At OmegaCEN/Kapteyn Astronomical Institute, contact danny@astro.rug.nl or kgb@astro.rug.nl

### 7.1.3 Starting the Astro-WISE Environment

To start AWE type

```
awe
```

and you will be welcomed by an awe-prompt similar to this

```
Python 2.7.6 (default, Jan  8 2014, 14:02:13)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.


               Welcome to the Astro-WISE Environment

|       You are running the AWBASE version

Importing Astro-WISE packages. Please wait...

Distributed Processing Unit: dpu.hpc.rug.astro-wise.org
Dataserver: ds.astro.rug.astro-wise.org

Current profile:
- username : <your database username>
- database : db.astro.rug.astro-wise.org
- project  : <your active project>
- current privileges : 1 (MyDB)

awe>
```

From this moment on the Astro-WISE Environment is at your disposal.

### 7.1.4 Access to the AWE software

After gaining experience with AWE you have the option to change the source code and make the changes available to AWE, so they are shared with other users. This is done via CVS, and the following sections explain how to get read access and write access to the AWE software.

#### AWBASE and test version

It is important to note at this point that we maintain a CVS version of the code which is tagged as "AWBASE". This version is tested and qualified and we therefore strongly recommend that you use this version if you do checkout your own code. The lastest CVS version is referred to as "current" or "test".

#### Read-only access to the AWE software

If you'd like to change the AWE software, you first need to get it via CVS.

```
cvs -d :pserver:anoncvs@cvs.astro-wise.org:/cvsroot login
cvs -d :pserver:anoncvs@cvs.astro-wise.org:/cvsroot checkout -r AWBASE awe
```

Usage of CVS is explained in §7.3 (HOW-TO Use CVS).

Note that in order to **use** your checkout of awe instead of the central checkout, you have to set an environment variable **AWEPIPE** to point to the **awe** directory. C-shell example:

```
setenv AWEPIPE /path/to/your/awe
```

It may be convenient to add this line to your ∼/**.cshrc** configuration file.

#### Write access to the AWE software

If you are granted write access, you can share your improvements of and additions to the source code with the Astro-WISE community.

Setting up write access to the CVS repository is further explained in §7.3 (HOW-TO Use CVS).

### 7.1.5 Access to the AWE dataservers

Access to the dataservers is automatically taken care of and does not require any action on the part of the user.

## 7.2 HOW-TO Documentation

Documentation for the Astro-WISE Environment is copius. Wading through it can be a daunting challenge. To make learning the system as simple as possible, several levels of documentation have been compiled. This HOW-TO explains all about AWE documentation.

### 7.2.1 HOW-TOs

The first place any user, especially new users, should go is the AWE HOW-TOs. These are task-specific documents designed to give the user good knowledge of a relatively small part of the system.

### 7.2.2 The Manual

The AWE Manual contains all the HOW-TOs in the same hierarchical form as the web page, but also includes more general and advanced documentation about the system. If the HOW-TO approach is not working for you, please try the Manual.

### 7.2.3 Documentation from the Code

Throughout the code **docstrings** are placed that indicate the purpose and functionality of packages, modules, classes, methods, functions etc. These can best be viewed in one of two ways: using a **PyDoc** server or by using the help functionality of Python while using the Python (AWE) command-line interface.

#### Online documentation

A pydoc server of a standard code checkout is always accessible online at

http://doc.astro-wise.org

A local pydoc server can be started with the following command:

```
pydoc -p <port>
```

or

```
awedoc -p <port>
```

This server will search any installed Python code and create HTML pages so that the code can be browsed with a webbrowser. The majority of the first page displayed will show installed Python libraries. The Astro-WISE code is marked by the location of your awe checkout. The server is accessible as

```
http://localhost:<port>
```

#### Inline documentation

It is possible to access docstrings from the Python/AWE prompt. This is often more convenient and faster than using a pydoc server.

```
awe> import astro.main.BiasFrame
awe> help(astro.main.BiasFrame)
```

Usually however you will call a similar page by calling help on a *class* rather than the *module* which contains this class (e.g. in this case, module BiasFrame.py also includes the class BiasFrameParameters):

```
awe> from astro.main.BiasFrame import BiasFrame
awe> help(BiasFrame)
```

Similarly one can get the docstring of individual *methods*:

```
awe> help(BiasFrame.make_image)
Help on method make_image in module astro.main.BiasFrame:

make_image(self) unbound astro.main.BiasFrame.BiasFrame method
    Make a master bias image.

    Requires:
        raw_bias_frames -- A list of raw bias frames
        read_noise      -- A ReadNoise object

    Do a trim and overscan correction on the input frames, compute
    a first estimate of the mean using a median of all trimmed and
    overscan-corrected raw biases. For each input bias reject
    pixels which deviate more than SIGMA_CLIP * read_noise from
    the median, and use the remaining pixels to compute a mean.
```

Note that a list of all methods, properties etc. of a class can be obtained with the "dir" command:

```
awe> dir(BiasFrame)
```

There is also a special Python help environment that can be started with this command:

```
awe> help()
```

The next command then gives an overview of the most important modules for our system:

```
help> modules astro.main
```

This (takes a little while) displays the following:

```
awe> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> modules astro.main
```

```
Here is a list of matching modules.  Enter any module name to get more help.

astro.main.AssociateList
astro.main.Astrom - the world coordinate system of a FITS file
astro.main.AstrometricCatalog
astro.main.AstrometricCorrection - Astrometric Correction
astro.main.AstrometricParameters
astro.main.AstrometricParametersFactory
astro.main.AtmosphericExtinction
astro.main.AtmosphericExtinctionCurve
astro.main.AtmosphericExtinctionFrames
astro.main.AtmosphericExtinctionZeropoint
astro.main.BaseCatalog - defines the base class for all catalogs
astro.main.BaseFlatFrame - defines the base class for all flat-fields
astro.main.BaseFrame - defines the base class for all frames (images)
astro.main.BaseWeightFrame - defines the base class for all weights
astro.main.BiasFrame - bias (req541)
astro.main.Catalog - defines a class for (SExtractor) catalogs
astro.main.Chip - class used to identify a single chip
astro.main.ColdPixelMap - cold pixel maps (req535)
astro.main.Config - provides a persistency mechanism for configuration parameters
astro.main.CosmicMap - defines classes used in detecting cosmic ray impacts
astro.main.DarkCurrent - dark current (req531) and particle event rate (req532)
astro.main.DataObject - objects with an associated data file
astro.main.DomeFlatFrame - dome flat (req542), master dome flats
astro.main.Filter - class used to identify the filter
astro.main.FringeFrame - fringe images (req545)
astro.main.GAstrometric
astro.main.GainLinearity - gain (req523) and linearity (req533)
astro.main.GainLinearity2 - gain (req523) and linearity (req533)
astro.main.HotPixelMap - hot pixel maps (req522)
astro.main.IlluminationCorrection
astro.main.IlluminationCorrectionFrame
astro.main.Imstat - image statistics
astro.main.Instrument - class used to identify the instrument
astro.main.Lamp
astro.main.LinearityMap - non-linear pixels (req533)
astro.main.LongAstrom - astrometry including PV matrix
astro.main.MasterFlatFrame - defines the final flat-field to be used (req546)
astro.main.NightSkyFlatFrame - night-sky flat frames (req544)
astro.main.ObservingBlock
astro.main.OnTheFly
astro.main.PhotExtinctionCurve
astro.main.PhotRefCatalog
astro.main.PhotSkyBrightness
astro.main.PhotSrcCatalog
astro.main.PhotTransformation
astro.main.PhotometricExtinctionReport
astro.main.PhotometricParameters
astro.main.PhotometricReport
astro.main.PhotometricSkyReport
astro.main.PixelMap - defines a class for pixel maps
astro.main.ProcessTarget - processable objects with quality control flags
astro.main.QuickCheckFrame - quick check (req547)
astro.main.RadioCube
astro.main.RawFinder
astro.main.RawFitsData - completely raw data
astro.main.RawFrame - classes for the different kinds of raw frames
astro.main.ReadNoise - read noise (req521) in ADU
astro.main.ReducedScienceFrame - de-bias & flat-field (seq632), apply astrometry (seq634/req555)
astro.main.RegriddedFrame
astro.main.RegriddedFrame_new - the support classes for regridding and coaddition
astro.main.SatelliteMap - maps of satellite tracks (if any)
astro.main.SaturatedPixelMap - maps of saturated pixels
```

```
astro.main.ScienceFrame - apply photometry (seq635)
astro.main.ShutterCheckFrame - shutter timing (req561)
astro.main.SourceList
astro.main.SubWinStat - SubWinStat
astro.main.SubWinStatFactory
astro.main.TwilightFlatFrame - twilight flat (req543), master twilight flats
astro.main.VariabilityFrame - Variability tool
astro.main.WeightFrame - individual weights (seq633)
astro.main (package) - The Persistent Object Hierarchy
astro.main.aweimports - automatic imports for the interpreter

help>
```

The following command can be given to get help on a specific module:

```
help> astro.main.BiasFrame
```

This will display the following page (don't bother reading it in any detail at this point):

```
Help on module astro.main.BiasFrame in astro.main:

NAME
    astro.main.BiasFrame - bias (req541)

FILE
    /data/users/helmich/awe/astro/main/BiasFrame.py

DESCRIPTION
    This module contains class definitions for BiasFrameParameters
    and BiasFrame.

    BiasFrame is the class that defines a master bias object.

    BiasFrameParameters is a class with parameters that are used, e.g., in
    trend analysis.

CLASSES
    astro.main.BaseFrame.BaseFrame(astro.main.DataObject.DataObject, astro.main.
ProcessTarget.ProcessTarget)
        BiasFrame
    common.database.DBMain.DBObject(__builtin__.object)
        BiasFrameParameters

    class BiasFrame(astro.main.BaseFrame.BaseFrame)
     |  Class for the master bias.
     |
     |  This class defines the master bias frame and provides the ability
     |  to reduce a list of raw bias input frames.  The reduction consists
     |  of averaging the trimmed and overscan-corrected raw bias frames
     |  and calculating the statistics of the derived frame.  Instances
     |  of this class have links to:
     |
     |  raw_bias_frames - List of raw bias objects.
     |  process_params  - Bias frame parameters object.
     |  prev            - Previous master bias object.
     |  imstat          - The Imstat object containing image statistics for the
     |                    reduced bias frame object.
     |  instrument      - The Instrument object describing which instrument the
raw
     |                    bias frame was observed with.
     |  chip            - The Chip object for the CCD with which the raw bias fr
ame
     |                    was observed.
     |  observing_block - The ObservingBlock object to which this bias
     |                    observation belongs.
     |
     |  Method resolution order:
     |      BiasFrame
     |      astro.main.BaseFrame.BaseFrame
     |      astro.main.DataObject.DataObject
     |      common.database.DBMain.DBObject
     |      astro.main.ProcessTarget.ProcessTarget
```

```
      |      common.database.DBMeta.DBMixin
      |      __builtin__.object
      |      astro.main.OnTheFly.OnTheFly
      |
      |  Methods defined here:
      |
      |  __init__(self, pathname='')
      |
      |  build_header(self)
      |      Extends BaseFrame build_header method
      |
      |  check_preconditions(self)
      |
      |  clean_up(self)
      |      This methods deletes intermediate products (like trimmed versions of
      |      raw frames that may cause problems when the BiasFrame is made a seco
nd
      |      time) from memory.
      |
      |  compare(self)
      |      Compare the results with a previous version.
      |
      |      Requires:
      |          prev -- A BiasFrame object for the previous Bias measurement
      |
      |      The folowing flags may be set in the status attribute:
      |         MEAN_DIFFER -- (mean of bias-mean of previous) > MAXIMUM_BIAS_DIF
FERENCE
      |
      |  copy_attributes(self)
      |
      |  derive_timestamp(self)
      |      Assign the period for which this calibration frame is valid.
      |
      |  get_canonical_name(self)
      |      Generate the unique filename for this BiasFrame.
      |
      |  make(self)
      |      Make a master bias frame.
      |
      |      Requires:
      |          raw_bias_frames -- A list of raw bias exposures.
      |
      |      Trims and applies overscan correction to the raw input bias
      |      frames. Averages these frames to derive the master bias
      |      frame. Calculates the image statistics on the resulting
      |      frame. Creates the FITS header and saves it together with the
      |      FITS image.
      |
      |  make_image(self)
      |      Make a master bias image.
      |
      |      Requires:
      |          raw_bias_frames -- A list of raw bias frames
      |          read_noise      -- A ReadNoise object
      |
      |      Do a trim and overscan correction on the input frames, compute
      |      a first estimate of the mean using a median of all trimmed and
      |      overscan-corrected raw biases. For each input bias reject
      |      pixels which deviate more than SIGMA_CLIP * read_noise from
      |      the median, and use the remaining pixels to compute a mean.
      |
      |  read_header(self)
      |      Extends the read_header method of BaseFrame
      |
      |  set_overscan_parameter(self)
      |      Sets the OVERSCAN_CORRECTION attribute of the BiasFrameParameters
      |      object associated with the BiasFrame based on OVSC_COR header keywor
d
      |
      |      Necessary for checking of inconsistencies between overscan correctio
n
      |      methods of (input) BiasFrame and (target) frame (e.g. a RawScienceFr
ame)
      |      that uses the BiasFrame.
      |
```

```
     |  verify(self)
     |      Verify the results.
     |
     |      The following flags may be set in the status attribute:
     |        BIAS_ABS_MEAN_TOO_LARGE : if the mean is significantly different f
rom zero
     |        BIAS_STDEV_TOO_LARGE    : if the stdev is too large
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
etc.
etc.
etc.
```

Evidently these pages can be quite lengthy. The structure of the pages is always the same however: aside from the docstring for the current module, the docstrings of the classes, methods, functions and properties as well as those of any superclass(es) of the class in question are displayed.
Exit help as follows:

```
help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
awe>
```

These pages contain exactly the same information as the HTML documentation created by a pydoc server.

### 7.2.4 The Code Itself

If your troubles (or just your curiosity) require more information than all the above documentation can provide, the last place to look is at the code itself. If you haven't already done so, get a CVS checkout and load the relevant files into your favorite text editor/viewer and have fun. Most software of interest will be located in `$AWEPIPE/astro/main`, `$AWEPIPE/astro/recipes`, or `$AWEPIPE/astro/toolbox`.

## 7.3 HOW-TO Use CVS

Projects for which different people work on the same set of files benefit from organizing these files and keeping track of any changes to them. To this end, a program called CVS (Concurrent Versions System) is used. In this chapter, a short introduction is given with regards to CVS functionality that we most often use. For the Astro-WISE project, source code, webpages, and documentation files are stored using CVS.

### 7.3.1 AWBASE and test version

It is important to note at this point that we maintain a CVS version of the code which is tagged as "AWBASE". This version is tested and qualified and we therefore strongly recommend that you use this version if you do checkout your own code. The lastest CVS version is referred to as "current" or "test".

### 7.3.2 Getting access

**Read-only** access is available using the following commands. The password to login can be requested from your Astro-WISE representative. Logging in can take a little time.

```
cvs -d :pserver:anoncvs@cvs.astro-wise.org:/cvsroot login
# base version
cvs -d :pserver:anoncvs@cvs.astro-wise.org:/cvsroot checkout -r AWBASE <module>
# test version
cvs -d :pserver:anoncvs@cvs.astro-wise.org:/cvsroot checkout <module>
```

where module can be:

- awhtml - for the webpages
- awe - for the pipeline (documentation is found in awe/astro/docs and various README files)

In order to get **write** permission you first need ssh (secure shell) access to the repository. First you will have to generate a private/public DSA key pair. Make sure you are using secure shell protocol 2! Recent versions of ssh should default to protocol 2, e.g. OpenSSH 2.9 and newer. The public/private keypair is generated with:

```
ssh-keygen -t dsa
```

You will be asked for a passphrase. This passphrase can be empty or, for additional security, a phrase. Make sure you remember the passphrase if you use one, because you will need it when authenticating yourself later. If you are behind a firewall check that it does not block outgoing ssh connections.

The next step is to send the file called:

```
~/.ssh/id_dsa.pub
```

to danny@astro.rug.nl and wait for further instructions.
While waiting for instructions add the ssh-rsa public key for the cvs.astro-wise.org host to the file called:

```
~/.ssh/known_hosts
```

Depending on the version of your secure shell you might have to add the key to a different file instead, e.g. `~/.ssh/known_hosts`. In addition, ensure that your version of ssh uses the same protocol as cvs.astro-wise.org server. Under some circumstances you'll have to create or edit a file called `~/.ssh/config` and add a line stating:

```
Protocol 2
```

The ssh-rsa public key for cvs.astro-wise.org can be found here.
Once you have write access, the code can be obtained with these commands:

```
setenv CVS_RSH ssh   (csh syntax)
  or
CVS_RSH=ssh; export CVS_RSH   (sh syntax)

cvs -d cvs.astro-wise.org:/cvsroot checkout -r AWBASE <module> # base version
cvs -d cvs.astro-wise.org:/cvsroot checkout <module>           # test version
```

where module can be:

- awhtml - for the webpages
- awe - for the pipeline (documentation is found in awe/common/docs, awe/astro/docs, and various README files)

### 7.3.3 Using your CVS checkout

Set the `$AWEPIPE` environment variable to the directory of the `awe` module to use your CVS checkout with the `awe`-prompt. That is, set `$AWEPIPE` to the directory with the `common` and `astro` directories. E.g. in csh

```
setenv AWEPIPE /Users/users/myname/myawe/awe
```

or in bash

```
export AWEPIPE=/Users/users/myname/myawe/awe
```

### 7.3.4 Using CVS

After checking out a module the next two CVS commands are most often used (there is a module called Demo on cvs.astro-wise.org that you can use to practice your CVS skills):

- cvs update

This command will update the locally checked out version of the module. The local version is merged with the latest version in the CVS repository. Usually it is a good idea to use CVS update -d which will ensure that directories that have been added to the module in the repository are created. Without the -d option only the directories that currently exist in the locally checked out module will be updated.

- cvs commit

This command will commit local changes to the CVS repository. You will be asked to supply an informational text, where you can enter e.g. what the reason for the change was. If used without options everything that was changed will be committed. If one or more files and/or directories are specified as arguments to CVS commit only those are committed.

In short, if you want to be in sync with changes that others have made you use CVS update, and if you want to propagate your own changes to others, use CVS commit.

NB. It is very important to regularly do an update and not to wait too long with a commit, especially when there is a high probability that you are not the only person working on a file. In particular, before you commit major changes it is important to update. If you do not, there is a risk that CVS will inform you of a conflict which has to be solved by hand.

A few examples:

```
cvs -nq update -dPA
```

Does not actually update (-n option), just does the regular checks. Runs in quiet mode (q). Creates new directories if any are found in the repository (-d option). Removes directories that are empty in the local checkout (-P option). Resets any tags, updates to the newest versions (-A option).

```
cvs commit -m "Fixed <error> in <place>" astro/toolbox/<something>.py
```

**Switching between the AWBASE and test versions**

To obtain the `AWBASE` version in your own CVS checkout, do

```
cvs -q update -r AWBASE -dP
```

This will replace all files with the `AWBASE` version. Note that files, that you have changed, will NOT be replaced. Subsequent updates with `cvs update` will continue to retrieve the files tagged as `AWBASE`. To get the most recent version again, do

```
cvs -q update -dPA
```

If you want the latest versions you will first have to use the "-A" option!

**Selecting the version for a given time**

It is possible to get a particular version of file from CVS for a given time. This makes it easy to resort to the version of a week ago, in case the most recent version appears to have problems. To obtain the version of a given time, do

```
cvs -q update -dP -D <date>
```

The `<date>` can be specified in a large variety of formats, including ISO-formatted (1972-09-24 20:05), MM/DD/YY (or DD/MM/YY, depending on your desktop settings). Some examples are

```
'1 month ago'
'2 hours ago'
'400000 seconds ago'
'last year'
'last Monday'
'yesterday'
'a fortnight ago'
'3/31/92 10:00:07 PST'
'January 23, 1987 10:05pm'
'22:00 GMT'
```

### 7.3.5 Moving the AWBASE tag

The AWBASE tag can be moved to a different revision of a file in CVS. When doing that, it is no longer possible to know to which revision of a file AWBASE was previously pointing. To be able to revert to the previous AWBASE the following steps have to be taken when moving the AWBASE tag in CVS.

1. Add a tag named AWBASE<yymmdd>, where <yymmdd> is the current date. For example, the tag for July 14th, 2006 becomes AWBASE060714. A convenient way to do that from the unix prompt is

   ```
   cvs tag AWBASE‘date +%y%m%d‘ filename
   ```

   *Be sure to supply a filename or all files will be tagged recursively!*

   The above example will tag the latest version. If you intend to move the AWBASE tag to a specific revision, other than the latest version, you have to include the -r option. To move to revision 1.20 of interesting.tex, for example, do

   ```
   cvs tag -r1.20 AWBASE060714 interesting.tex
   ```

   The cvs log command can be used to get an overview of all revisions of a file.

2. Move the AWBASE tag to the tag that was created in the previous step. The following command can be executed from the unix prompt to do this conveniently

   ```
   cvs tag -r AWBASE‘date +%y%m%d‘ -F AWBASE filename
   ```

   *Be sure to supply a filename or all files will be tagged recursively!*

3. Remove the sticky tag that was created by the cvs tag command.

   ```
   cvs -q update -A filename
   ```

   This step is higly recommended for beginning CVS users and optional for expert CVS users that know about “stickyness”. If you have no idea about your level of CVS expertise, always perform the last step.

   Of course, the above commands can be combined in an alias for convenience. An example for Unix in C Shell is:

   ```
   alias cvstag ’cvs tag AWBASE‘date +%y%m%d‘ \!* ; cvs tag -r AWBASE‘date
                                +%y%m%d‘ -F AWBASE \!* ; cvs -q update -A \!*’
   ```

## 7.4 HOW-TO Schedule Astro-WISE compliant observations

The Astro-WISE system is a database driven environment for the reduction of astronomical image data. The system is capable of processing data from many different Wide-Field cameras. Currently, the system is using data from

- *OmegaCAM* imager at the VLT survey telescope
- *WFI*, the Wide-Field Imager at the 2.2m ESO telescope, La Silla
- *WFC* the Wide Field Camera at the INT, La Palma
- *MDM8K* at the MDM observatory

The Astro-WISE system was primarily created for the reduction of *OmegaCAM* data. This is reflected in the design of the system, and has consequences for any observations to be scheduled for instruments other than *OmegaCAM* itself.

This chapter provides guidelines for scheduling observations that are in agreement with the data model and data requirements of the Astro-WISE system. These guidelines not only facilitate easy processing by the Astro-WISE system, but also allow a smooth insertion of this data into the database (i.e. ingestion). It is, therefore, strongly advised to adhere as much as possible to the guidelines provided in this chapter. A short summary of the guidelines is given in Table 7.1. If you plan to use the Astro-WISE system for the reduction of data from instruments other than the ones mentioned above it is advised to contact the Astro-WISE group in Groningen.

### 7.4.1 Data requirements

In the following subsections, short descriptions are given of the minimum requirements one has to meet to smoothly work with the Astro-WISE system.

#### Read noise and bias

In the *OmegaCAM* calibration plan, the read noise is determined as part of the daily health check procedures. For the Astro-WISE system, this means that there is a processing step devoted to deriving this read noise. This step in the data reduction can not be skipped, because the read noise is a necessary ingredient in creating the master bias.

For deriving the read noise and the master bias, bias exposures are required. For the read noise, exactly **two** bias exposures are required, for the master bias at least **five** (the *OmegaCAM* calibration plan even specifies as many as ten). Please, make sure that at least once during the observing run read noise observations are taken (otherwise the Astro-WISE system would have to fall back on pre-fab values of the read noise which might not always be the most appropriate). To easily distinguish between these sets of bias images in preparation for the ingest, it is advisable to use the different entries for the OBJECT header key word as given in Table 7.1.

#### Dome flats and twilight flats

In the Astro-WISE system, the science data is flat-fielded with a master flat that has been derived from dome flats and/or twilight flats. Optionally, one can, in addition, use flats derived from the nightsky to construct the master flat.

For deriving a master dome flat, the system requires at least **five** exposures for every filter in which science observations are taken. The same requirements are in place for deriving a master twilight flat. To easily distinguish between the two sets of flat-field images in preparation for the ingest, it is advisable to use the different entries for the OBJECT header key word as given in Table 7.1.

Table 7.1: Data requirements for Astro-WISE compliant processing. All entries in the table are mandatory, except for the last one; the extinction measurement is optional. The first and second columns give the purpose and type of the data. The minimum observing frequency and number of exposures per observation are given in the third and fourth columns. The last column gives the recommended header value of the OBJECT key word.

| Purpose | Data type | Frequency | # Exps. | OBJECT |
|---|---|---|---|---|
| readnoise | bias | ≥1/run | =2 | BIAS, READNOISE |
| bias | bias | ≥1/night | ≥5 | BIAS |
| dome | flatfield | ≥1/run | ≥5/filter | FLAT, DOME |
| twilight | flatfield | ≥1/night | ≥5/filter | FLAT, SKY |
| photom | science | =1/night | =1/filter | STD, ZEROPOINT |
| | science | optional | =2/filter | STD, EXTINCTION |

### Photometric standard fields

This section concerns itself with observations for establishing a zeropoint for the night.

A description of the contents of the most recent standard star catalog in the Astro-WISE system can be found in the relevant chapter on the photometric pipeline (see 19.1.1). There it is also described how to do photometric calibration based on other standard stars which requires ingestion of their standard magnitudes in the Astro-WISE system.

In order to be able to derive the zeropoint for the night with the Astro-WISE system, it is necessary to observe one of the prescribed photometric standard fields **once** in the middle of every night, for every filter in which science observations are taken during that night (no dithering is required, one exposure will suffice).

### Photometric monitoring

The *OmegaCAM* calibration plan contains observations of a field near the South Equatorial Pole for the purpose of monitoring the stability of the atmosphere. The selection of the field is currently in progress, but the field will most likely be near ra=51.4286deg, dec=89.0426deg. Currently, such photometric monitoring observations are not mandatory for successful data reduction with the Astro-WISE system. Furthermore, the Astro-WISE system facilitates the derivation of the extinction using two standard star fields observed at different airmasses.

### Gain

Gains have already been determined for all instruments currently supported by the Astro-WISE system. Optionally, one can measure the gain using a very specific serie of dome flat measurements.

### 7.4.2 Notes on specific instruments

#### WFI@ESO2.2m

No instrument specifics at present.

#### WFC@INT

No instrument specifics at present.

#### MDM8K@MDM2.4m

Make sure that all the output from the instrument is in Multi-Extension FITS format, and that both the calibration and science data are binned to 1024x2048 pixels.

### 7.4.3 Standard tiling and pixelation of the sky

A preliminary tiling and pixel grid scheme has been defined for *OmegaCAM* (1 square degree FOV). It involves 22717 Square degree fields for the Southern hemisphere. A main aim is to facilitate the stacking/co-adding/differencing of images taken at different epochs/bands/projects even when combining data from different Wide-field-imager data in the Astro-WISE system Therefore it might be prudent to use the pointings of the tiling grid in observations with any instrument. Please refer to this white paper on Tiling the sky for more details:
http://www.astro.rug.nl/∼omegacam/dataReduction/Tilingpaper.html

### 7.4.4 Viewing observations already in the Astro-WISE system

To get an overview of the pointings of observations already present in the Astro-WISE database one can use the ObsView module. Usage of ObsViewer is explained in §23.6 (HOW-TO ObsViewer) To get more details on the observations one can use the DBViewer. Usage of the DBViewer is explained in §23.4 (HOW-TO DBViewer)

## 7.5 HOW-TO Ingest raw data into the database

Before any data can be processed by the Astro-WISE system, it must be ingested. Ingestion in this case means: splitting the images, storing these on the fileserver, and making an entry in the database. This chapter describes the necessary preparations for ingestion, and the ingestion process itself.

### 7.5.1 Preparations for the ingest

The images to ingest should first be sorted based on their intended use (i.e. their *purpose*, type at the unix prompt

```
awe $AWEPIPE/astro/toolbox/ingest/Ingest.py
```

to get information on the possible purposes. For locally stored, *un*compressed copies of each image proceed as follows (if the images are compressed, decompress them first):

**1)** Identify the files by collecting relevant header items. This can be done by using something like `gethead <header items> *.fits`. However, if there are too many files, the shell will refuse to expand the command. In this case, use **foreach** instead, and append the output to a file:

```
> foreach i ( *.fits )
foreach? gethead "HIERARCH ESO TPL ID" "HIERARCH ESO DPR TYPE" IMAGETYP
OBJECT "HIERARCH ESO INS FILT1 ID" "HIERARCH ESO INS FILT ID" EXPTIME $i
>> headers.txt
foreach? end
```

> Explanation: **foreach** is a c-shell looping construct. **gethead** is a **wcstools** program to get header items from FITS files. Hence for each FITS file a few relevant header items are read and appended to the file "headers.txt". Wcstools is most likely installed on your system. Website: http://tdc-www.harvard.edu/software/wcstools/.

**2)** Group the images by the *purpose* for which these have been observed. This grouping is based on the header information retrieved in the previous step. For example:

```
> grep "BIAS, READNOISE" headers.txt > readnoise.txt
> grep "FLAT, DOME" headers.txt > domes.txt
> grep "FLAT, SKY" headers.txt > twilight.txt
> grep "STD, ZEROPOINT" headers.txt > photom.txt
> grep "NGC" headers.txt > science.txt
```

This will be easy if the guidelines for scheduling observations given in chapter 7.4 have been followed.

**3)** Use, for example, the editor **vim** to remove anything but the file names from the text files produced in the previous step:

```
> vim bias.txt
```

Then type ":" to enter command mode (Esc cancels):

```
": %s/fits.*/fits"
```

This is a regular expression that will search and replace each occurrance of "fits<something>" with "fits". "%" means for all lines, "s" is for subsitute, and the "/"'s are to separate the search and replace expressions, ".*" matches one or more characters of any kind.

**4)** You now have files containing a list of FITS filenames (one per line) named after the *purpose* for which the data was obtained. Now move the FITS files (or links) to subdirectories named after this *purpose*, for example:

```
> mkdir READNOISE
> foreach i (`cat readnoise.txt`)
foreach? mv $i READNOISE
foreach? end
```

That is it for the preparation. There are, of course, many ways to do this preparation, but this way is quite fast for any number of files.

**Tips and possible complications**

It may be helpful, especially when trying to ingest many files, to place links to the location of the raw MEF files in your current working directory:

```
> foreach i ( /ome03/users/data/WFI/2004/10/*.fits )
foreach? ln -s $i
foreach? end
```

In case the files are compressed with the common Unix compression programs gzip, zcat or bzip2 just make the links to the compressed files in the same way:

```
> foreach i ( /ome03/users/data/WFI/2004/10/*.fits.Z )
foreach? ln -s $i
foreach? end
```

Now we have links to all the files you want to ingest in your current working directory.

In case the images are compressed with common compression algorithms, you could work as follows:

```
> foreach i ( *.fits.bz2 )
foreach? dd bs=500k count=1 if=$i | bzip2 -qdc > hdr.txt
foreach? echo -n "$i " >> headers.txt
foreach? gethead "HIERARCH ESO TPL ID" "HIERARCH ESO DPR TYPE" IMAGETYP OBJECT
"HIERARCH ESO INS FILT1 ID" "HIERARCH ESO INS FILT ID" EXPTIME hdr.txt
>> headers.txt
foreach? end
```

(Explanation: dd (disk-dump(?)) reads one block of size 500k from the input file $i. The ouput is decompressed by bzip2 and redirected to an ascii file. You can use gethead on this file again to get the header items. Output is appended to the same file "headers.txt".)

Other commands that may be of use:

```
> fgrep [-v] -f <file1> <file2>  -- Print difference between files (diff works
much slower on large files).
> wc <file>                      -- Word count
```

### 7.5.2 Ingesting data

The actual ingestion of the data is handled by a **Recipe** called `Ingest.py`, which can be found in `$AWEPIPE/astro/toolbox/ingest`. If your username is AWJSMITH the **Recipe** is invoked from the Unix command line with the following command:

```
env project='AWJSMITH' awe $AWEPIPE/astro/toolbox/ingest/Ingest.py -i <raw data> -t <type> [-com
```

where `<raw data>` is one or more file names (for example WFI*.fits), and `<type>` of the data to be ingested. Setting the environment variable project ensures that the data is ingested into your personal context. See the Context HOW-TO for a description of the notion of context. To get a list of all possible values for the `-t` parameter, just type:

```
awe $AWEPIPE/astro/toolbox/ingest/Ingest.py,
```

and an on-screen 'manual' will show up.

Running the `Ingest.py` recipe, making good use of the preparations described in the previous section, is done thus (the read noise is taken as an example):

```
> cd READNOISE
> env project='AWJSMITH' awe $AWEPIPE/astro/toolbox/ingest/Ingest.py -i *.fits -t readnoise -commit
```

An alternative command, using science data as an example, is:

```
> cd SCIENCE
> foreach i (*.fits)
foreach? env project='AWJSMITH' awe $AWEPIPE/astro/toolbox/ingest/Ingest.py -i $i -t science -commi
foreach? end
```

*Important note*: due to the nature of the ingestion script, this last command can *only* be used for lists of individual science images.

The input data of the ingest script should be in the form of Multi-Extension FITS files (MEFs); most wide-field cameras write the data from their multi-CCD detector block in this form. The ingestion step splits an MEF file into its extensions, creates objects (a software construct) for each extension, stores each extension separately on a dataserver, and then commits the object, with relevant header items connected to it, to the database. Note that each extension is still saved locally, so make sure there is enough free space in the location you are running the ingest script. After ingesting, the local copies of the FITS files can be removed. The `commit` switch is necessary to actually store/commit data; if it is not specified, nothing is written to the dataserver or committed in the database. Note that a log is generated of the ingest process. The log file is called something like `<datetime>.log`.

Each file that is ingested needs to be named according to our filenaming convention. This means that the MEF file is named as follows:

```
<instrument>.<date_obs>.fits
```

Example: WFI.2001-02-13T01:02:03.123.fits

If the file to be ingested is not named according to this convention, a symbolic link with the correct name is created, and the image is ingested with that filename. Hence **the ingested image may not retain its filename**.

## 7.6 HOW-TO: Work with Dates and Times in AWE

Observing nights and timestamps (valid ranges) are important concepts in AWE. These concepts are inherently confusing and some understanding of our conventions is necessary. This HOW-TO describes how to work with dates and times.

### 7.6.1 Observing nights

When an astronomer does observations with a telescope this happens during the night at that telescope. We define a **night** for a particular telescope as **the period between noon and noon the next day at that telescope**. This concept is only valid in terms of **local time**. Since all relevant times that are stored in the database are stored in UTC, depending on the telescope, a conversion from local time to UTC and vice versa is necessary.

The following raw science images observed by WFC@INT are observed in the same night. This is the night of 30 March 2005 in our terminology.

```
WFC.2005-03-30T19:18:55.8.fits
WFC.2005-03-30T20:00:29.6.fits
WFC.2005-03-30T21:11:19.7.fits
WFC.2005-03-30T22:32:10.8.fits
WFC.2005-03-30T23:12:01.1.fits
WFC.2005-03-31T00:00:41.9.fits
WFC.2005-03-31T01:06:57.2.fits
```

In fact all data with observing dates (filenames) between `WFC.2005-03-30T12:00:00.fits` and `WFC.2005-03-31T12:00:00.fits` are considered part of the night of 30 March 2005 for WFC (see table 7.3).

### 7.6.2 Input from the user

Based on input from the user, namely the night for which to process data, both science images and calibration images are selected in the database and used/created. This input is always called "date" and is an argument for command line driven interfaces as well as the web services.

Examples:

```
awe> task = ReadNoiseTask(date='2000-04-28', instrument='WFI', chip='ccd50')

awe> dpu.run('ReadNoise', d='2000-04-28, i='WFI')

awe> query = RawScienceFrame.select(date='2000-04-28', instrument='WFI',
...                                 object='CDF4_B_1')
awe> for f in query: print f.DATE_OBS
...
2000-04-29 00:04:29.00
2000-04-29 00:04:29.00
2000-04-29 01:05:58.00
2000-04-29 00:04:29.00
2000-04-29 00:04:29.00
2000-04-29 00:04:29.00
2000-04-29 01:05:58.00
2000-04-29 01:05:58.00
2000-04-29 01:05:58.00
```

| **raw images and science images** | | |
|---|---|---|
| *description* | *name (header item)* | *type of value* |
| observing date | DATE-OBS | datetime type (UTC) |
| observing date | UTC | float |
| observing date (modified julian date) | MJD-OBS | float |
| observing date (local siderial time) | LST | float |
| FITS file write time | DATE | datetime type (UTC) |
| (if available) start of observing block | (...)OBS START | datetime type (UTC) |
| (if available) start of template | (...)TPL START | datetime type (UTC) |
| **calibration images** | | |
| *description* | *name* | *type of value* |
| start of valid range | timestamp_start | datetime type (UTC) |
| end of valid range | timestamp_end | datetime type (UTC) |
| creation date | creation_date | datetime type (UTC) |
| start of observing block | (observing_block.)date_obs | datetime type (UTC) |
| start of template | (template.)date_obs | datetime type (UTC) |

Table 7.2: Dates as stored in the database

```
2000-04-29 01:05:58.00
2000-04-29 00:04:29.00
2000-04-29 01:05:58.00
2000-04-29 00:04:29.00
2000-04-29 00:04:29.00
2000-04-29 01:05:58.00
2000-04-29 01:05:58.00
```

In fact all data with observing dates (filenames) between `WFI.2000-04-29T16:00:00.fits` and `WFI.2000-04-30T16:00:00.fits` are considered part of the night of 29 April 2000 for WFI (see table 7.3).

### 7.6.3 Time stamps

To be able to automatically select appropriate calibration files (bias, flat etc.) from the database for a given science image or as a result of the "date" input by the user, timestamps are assigned to each calibration file that define a period for which this calibration file is valid. All dates that are stored in the database and used for this purpose are in UTC. See also table 7.2. Calibration files are valid for a certain multiple of observing nights, depending on the type of file.

### 7.6.4 Dates in the database

The dates in table 7.2 are stored in the database.

### 7.6.5 Conversions between local time and UTC

A night in terms of UTC is determined by converting "noon" to UTC. No correction is made for any daylight saving time. The shift in the timestamps is exactly equal to the timezone of the instrument.

| *instrument* | *timezone* | *typical timestamp start (UTC)* | *typical timestamp end (UTC)* |
|---|---|---|---|
| WFC @ INT | 0 | 2005-01-05T12:00:00 | 2005-01-06T12:00:00 |
| WFI @ 2.2m | 4 | 2005-01-05T16:00:00 | 2005-01-06T16:00:00 |
| OCAM @ VST | 4 | 2005-01-05T16:00:00 | 2005-01-06T16:00:00 |
| MDM8K @ 2.4m | 7 | 2005-01-05T19:00:00 | 2005-01-06T19:00:00 |

Table 7.3: Instrument time zones

## 7.7 HOW-TO Process Data in a Distributed (Parallel) Way

### 7.7.1 Summary

Here are some of the actions that one may want to do when Processing using the Distributed Processing Unit (DPU). These actions are described in this HOW-TO.

- Submit you own jobs to the queue (either from AWE or by using a webservice)
- Inspect the status of your jobs or a DPUin general
- Request which tasks (sequence of tasks) the dpu recognizes
- Request which options a task (sequence of tasks) recognizes.
- Using your local (changed) code when processing remotely
- Cancel jobs
- Obtain the logs of your jobs

At the AWE prompt an object is available which is used to run jobs on the DPU. Public methods for this object are:

- *dpu.get_jobids()*: Return the list of identifiers of your jobs known to the DPU
- *dpu.run(<arguments>)*: Submit jobs
- *dpu.get_dpu_identifiers()*: Returns a list of all known DPU's.
- *dpu.show_dpu_identifiers()*: Print the above, more verbose
- *dpu.select_dpu()*: Select a different DPU
- *dpu.get_sequence_identifiers()*: Returns a list of known task sequences
- *dpu.show_sequence_identifiers()*: Print the above, more verbose
- *dpu.get_sequence_options(<sequence identifier>)*: Returns a list of all arguments that are recognized for the given sequence identifier
- *dpu.show_sequence_identifiers(<sequence identifier>)*: Print the above, more verbose
- *dpu.get_job_result(<jobid>)*: If results have been committed, returns a list of the main created objects
- *dpu.get_logs([<job identifier(s)>])*: Return all (no argument) logs or the logs of the specified job(s)
- *dpu.cancel_job(<jobid>)*: Cancels a job (as long as it is not yet running)

In the Environment with key *dpu_name* it is defined which DPUto use. This can be changed to select a different DPUas default. To change the DPUin an AWEsession use the method *dpu.select_dpu()*.

### 7.7.2 Viewing the queue

For each Distributed Processing Unit (DPU) there is a webpage available which displays its queue. These pages can be found from the homepage, under *Astro-WISE Information System -> Processing Grid -> Cluster Queues*.

On the page you can inspect the status of your jobs. Shown are among others, user information, job status and running time. Here is the webpage for the DPU in Groningen:

(click on the links in the lower left part of the screen to view the queue)

http://dpu.hpc.rug.astro-wise.org/

### 7.7.3 Processing in AWE

From the `awe`-prompt it is possible to process your data remotely and in a distributed fashion. When you start up the interpreter an instance of the class used for this (the `Processor` class), is automatically generated and assigned to the variable "dpu". So, when you start `AWE` you will see something like this:

```
Python 2.5.1 (r251:54863, Jul 25 2007, 11:52:36)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.


                Welcome to the Astro-WISE Environment


Importing Astro-WISE packages. Please wait...

Initializing Distributed Processing Unit...

Current profile:
- username : <username>
- database : db.astro.rug.astro-wise.org
- project  : <project>

awe>
```

The message about the Distributed Processing Unit indicates that a `Processor` instance (called "dpu") has been created.

The dpu can be asked which sequence identifiers it recognises:

```
awe> dpu.get_sequence_identifiers()
['HotPixels', 'ReadNoise', 'Bias', 'MasterFlat', 'NightSkyFlat', 'DomeFlat',
'TwilightFlat', 'Gain', 'ColdPixels', 'QuickCheck', 'FringeFlat', 'Photcat',
'Photom', 'Reduce>GAstrometry>Coadd', 'Regrid>Coadd', 'GAstrom', 'SourceList',
'Reduce', 'Astrometry', 'Regrid', 'GAstrometry', 'Reduce>GAstrometry>Regrid',
'GAstromSL', 'Coadd', 'Reduce>Astrometry', 'Reduce>Regrid', 'Reduce>Coadd',
'AssociateList', 'GalFit', 'GalPhot']
```

The dpu can also be asked which options a particular task takes:

```
awe> dpu.get_sequence_options('HotPixels')
['i', 'd', 'c', 'p', 'C', 'instrument', 'date', 'chip', 'pars', 'commit']
```

More verbose:

```
awe> dpu.show_sequence_options('HotPixels')
Recognized options for the astro.recipes.HotPixels OptionsParser are:
"i" or "instrument" (default: )
"d" or "date" (default: )
"c" or "chip" (default: )
"p" or "pars" (default: {})
"C" or "commit" (default: 0)
```

You can start tasks as follows:

```
awe> dpu.run('DomeFlat', i='WFI', d='2000-04-28', f='#842') # all CCDs
awe> dpu.run('DomeFlat', i='WFI', d='2000-04-28', f='#842', C=1) # commit results
awe> dpu.run('DomeFlat', i='WFI', d='2000-04-28', f='#842', c='ccd50') # one CCD
awe> dpu.run('Reduce', i='WFI', o='CDF4_B_?', d='2000-04-29', f='#845')
```

Further options can be specified for dpu.run:

- `dpu_time`: override any dpu_time (expected processing time) derived by the tasks themselves
- `dpu_mem`: specifying the memory required may influence which nodes are used
- `dpu_aweversion`: version (AWBASE or current) of binaries to use on the compute cluster
- `send_code`: send your own version of the Python code (located in directory AWEPIPE) to use on the DPU
- `return_jobid`: return jobid for accounting purposes

The "run" method will print a job identifier for every job submitted to the queue of the Distributed Processing Unit (DPU):

```
[schmidt] 09:55:55 - Calling: Processor.run(ReadNoise, instrument= , i=WFI ,
 c=ccd50 , d=2000-04-28)
[schmidt] 09:55:55 - Estimated process time : 10 seconds
[schmidt] 09:55:55 - Sending job with identifier 82382 to DPU
```

These jobids can be used to retrieve logs of finished jobs (see the next section). Jobids of jobs you submitted can be requested with this command:

```
awe> dpu.get_jobids()
[84820L]
```

A concise status overview for your jobs can be requested as well:

```
awe> dpu.get_status()
[schmidt] 13:37:11 - Jobid 84820 has status FINISHED N/E/A/S/U 75/0/0/0/0
```

### 7.7.4 Using your local (changed) code when processing remotely

When you run a job using the DPU and you have a local checkout of the code, this code is shipped to the computing cluster and used there, except when the option `send_code=False` is given to the dpu.run method.

### 7.7.5 Options

Several options can be given to the dpu.run method:

- dpu_time: override any dpu_time (expected processing time) derived by the tasks themselves
- dpu_mem: specifying the memory required may influence which nodes are used
- dpu_aweversion: version (AWBASE or current) of binaries to use on the compute cluster.
- return_jobid: return jobid for accounting purposes
- send_code: send your own version of the Python code (located in directory AWEPIPE) to use on the DPU)

### 7.7.6 Logs and job identifiers

In an active `AWE` session it is possible to ask the Processor to return logs:

```
awe> dpu.get_logs()
12:22:16 - ++++++++++++++++++++++++++++++++++++++++++++
12:22:16 - job_status = 0
12:22:16 - ++++++++++++++++++++++++++++++++++++++++++++
12:22:16 - 12:10:00 - Querying database for instances of class RawBiasFrame
12:10:22 - Running : imcopy 'OCAM.2005-09-07T07:43:44.905_3.fits[1]' 'tmpdTqMx
N.fits'
12:10:22 - Running : imcopy 'OCAM.2005-09-07T07:44:36.165_3.fits[1]' 'tmpz6bkO
r.fits'
12:10:23 - Making ReadNoise object
12:10:23 - Using RawBiasFrame OCAM.2005-09-07T07:43:44.905_3.fits
12:10:23 - Using RawBiasFrame OCAM.2005-09-07T07:44:36.165_3.fits
12:10:23 - Computing difference...
12:10:23 - Estimating rms iteratively...
12:10:23 - Maximum number of iterations : 5
12:10:23 - Rejection threshold :    5.0 (sigma)
12:10:24 - The read noise (ADU) for ccd ESO_CCD_#67 is :   2.34
12:10:24 - Difference between biases is :  -0.077 (mean),   0.00 (median)
```

The returned lines are also written to a single ("date+time".log) file in your local directory per `awe`-prompt session.

### 7.7.7 Cancelling jobs

Jobs can be cancelled from the `awe`-prompt using the following command:

```
awe> dpu.cancel_job(313L)
```

or

```
awe> for jobid in dpu.get_jobids():
         dpu.cancel_job(jobid)
```

I.e. the argument is the job identifier of the job you want to cancel.

## 7.8 HOW-TO USE DARMA

DARMA is a Data Acquisition, Representation, and MAnipulation interface for use with FITS[2] images and headers, and eventually FITS tables. The current interface with the Astro-WISE Environment uses only the FITS header functionality of DARMA, and that is all that will be discussed in this HOW-TO.

### 7.8.1 DARMA Header Interface

DARMA headers are effectively PyFITS[3] headers with alot of fancy extra functionality and optimizations to make working with FITS headers in Astro-WISE very simple and powerful. As with PyFITS headers, DARMA headers include header verification, but on-demand instead of upon output to a file, so you always know your header meets the current FITS standard as implemented in the version of PyFITS being used. DARMA's header interface has at minimum, the same functionality Eclipse headers have. This means, that if you are already familiar with Eclipse headers, you need not read further to use it. If you are not or want to know more, both the basic and the more advanced functionality will be described below and in the following sections.

The basic access to DARMA headers is a Python dictionary-like interface:

```
awe> header = darma.header.header('filename.fits')
awe> header['BITPIX']
-32
awe> header['OBJECT']
'Standard'
awe> header['OBJECT'] = 'Standard (SA101)'
awe> header['OBJECT']
'Standard (SA101)'
awe> del header['OBJECT']
awe> header['OBJECT']
None
awe> header['OBJECT'] = 'Standard (SA101)'
awe> header['OBJECT']
'Standard (SA101)'
awe> header['OBJECT'] = ('Standard (SA101)', 'Object name from telescope')
awe> header.OBJECT
OBJECT  = 'Standard (SA101)'   / Object name from telescope
awe>
```

The example above shows the basic interface to a FITS header and basic manipulations of the header values. A header is loaded upon instantiation with a valid filename and an optional index indicating the extension number (0=primary, 1=first extension, 2=second extension, etc.). Access to a keyword or numeric index of the header returns the value of the keyword, and allows modification of the value or deletion of the keyword entry, or addition of a new one (the old one is recreated in the example above.). Lastly, keyword comments can be added using a (value, comment) tuple. The card displayed by the header attribute shows the added comment. Header attribute access is described next.

[2] FITS is currently the only data packaging format supported. Different formats can and will be included in the future.

[3] http://www.stsci.edu/resources/software_hardware/pyfits/

NOTE: Keywords dealing with the data (e.g., BITPIX, NAXIS, etc.) can be modified, but any changes will be lost when the header is paired with data, as their values will be take directly from the data.

In addition to the primary dictionary interface, there is an informational, attribute-based interface that can show the value of the entire header card for a given keyword. There is also a special attribute that shows a listing of all header cards:

```
awe> header.BITPIX
BITPIX  =                  -32 / number of bits per data pixel
awe> header.BITPIX.key
'BITPIX'
awe> header.BITPIX.value
-32
awe> header.BITPIX.comment
'number of bits per data pixel'
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -32 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
EXTEND  =                    T / FITS dataset may contain extensions
.
.
.
awe>
```

The special attribute 'card_list' returns the representation of a PyFITS CardList (pyfits.Header.ascardlist()) and shows exactly what the actual header looks like in the FITS file. The dump() method simply prints the string representation of this same card list. Use the dump() method to show the header contents with newline characters included.

NOTE: The attribute-based interface goes only one-way. It shows the keyword values, but cannot set them. You must use the dictionary interface to modify keyword values or comments!

### 7.8.2 On-demand Header Verification

DARMA headers are verified using PyFITS output verification on demand at every change or access to the header (e.g., modify, add, or delete a keyword, value, and/or comment). The verification is robust with a default verification option of 'silentfix'. Use the 'option' keyword in the constructor to override this default value.

ADVANCED: If you need to modify or add several keywords at once that would normally raise a verification error at each one, you can set the _IS_VERIFIED flag of the header object to True between changes to suppress automatic verification. After all the changes are complete, the next access to the header will run the verification to ensure that all is well.

### 7.8.3 Special Keywords

FITS standard allows for several keywords with special meaning or behavior. These keywords generally cannot be accessed or modified in the normal ways. Their unique properties are discussed below.

### COMMENT, HISTORY, and 'BLANK' keywords

The COMMENT keyword allows arbitrary lines of text to be entered into the header to give general commentary capability at any location in the header. The string will be folded at a length of 72 characters and put into as many comment cards as needed (i.e., a string of length 160 will be folded into 3 COMMENT cards of string length 72, 72, and 16 characters).

The HISTORY keyword has a similar purpose and is processed in the same way as the comment card. Its primary purpose, as its name implies, is to give a record of the history the associated FITS image has seen.

The 'BLANK' keyword provides the facility to add spacers, or line-holders in the header. It is unlike the previous two because no keyword for it appears in the header, and because it can take an optional string value where the others have a mandatory string value.

All three of these keywords have special methods to write and to read them because they are not unique in occurrence like all other FITS keywords. The usage of the COMMENT, HISTORY, and 'BLANK' keyword cards is illustrated below:

```
awe> header = darma.header.header().default()
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
awe> header.add_comment('This is a comment.')
awe> header.add_comment('This          is          a          very \
...         long        comment          indeed.')
awe> header.add_history('This is a history statement.')
awe> header.add_blank(after='EXTEND')
awe> header.add_blank('         Begin comments and histories.', after='EXTEND')
awe> header.add_blank(after='EXTEND')
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T

                  Begin comments and histories.

COMMENT   This is a comment.
COMMENT   This          is          a          very        long        comment
COMMENT            indeed.
HISTORY   This is a history statement.
awe>
```

### HIERARCH

PyFITS includes native support for ESO's HIERARCH keywords usage. With HIERARCH keywords, you can extend the 8-character limit of the standard FITS convention and use ESO's strict hierarchical keyword structure or a free-form structure. The keyword/value/comment length cannot exceed the length of the card (i.e., 80 characters). See the current Registry of FITS Conventions for more details.

DARMA handles HIERARCH keywords in effectively the same way PyFITS does. HIERARCH keyword cards can be accessed or set with or without the 'HIERARCH ' string preceding

the keyword. Keywords that are automatically interpreted as HIERARCH keywords are those that contain spaces, exceed 8 characters, or contain non-alphanumeric characters other than '-' or '_'. Keywords beginning with 'HIERARCH ' are always interpreted as HIERARCH keywords.

```
awe> header = darma.header.header().default()
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
awe> header['HIERARCH ESO TEL TEMP'] = 300
awe> header['this keyword is non-standard'] = 'non-standard'
awe> header['ThisKeywordIsAlsoNonStandard'] = 'also non-standard'
awe> header['strange$keyword'] = 'strange'
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
HIERARCH ESO TEL TEMP = 300
HIERARCH this keyword is non-standard = 'non-standard'
HIERARCH ThisKeywordIsAlsoNonStandard = 'also non-standard'
HIERARCH strange$keyword = 'strange '
awe> header['ESO TEL TEMP']
300
awe>
```

### 7.8.4 Saving and Advanced Creation

Unlike Eclipse headers, DARMA headers can be saved and loaded from text files or lists of header cards. In this section, that functionality will be illustrated.

#### Saving Headers

Saving a header is as simple as calling its save method:

```
awe> header = darma.header.header().default()
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
awe> header.save('header.fits')
awe> header.save('header.txt', raw=False)
awe> os.system('cat header.txt')
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
END
0
awe>
```

There are two ways to save a header as shown above: as raw FITS header as one line in 2880 byte blocks (the default), or as standard text with newlines at the end of each line in only enough file space to contain it. As you may have already noticed, the card_list does not contain an 'END' header card. This is because PyFITS headers only receive the 'END' card when written to a FITS file with data. This is also done upon saving a the header as you can see in the above example. The '0' after the 'END' card is just the return value from the shell.

**Creating Headers**

DARMA headers can be loaded from FITS files as expected, but they can also be created in three other ways:

```
awe> header = darma.header.header(card_list='header.txt')
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
awe> card_header = darma.header.header(card_list=header.card_list)
awe> card_header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
awe> fd = file('header.txt')
awe> lines = fd.readlines()
awe> fd.close()
awe> str_header = darma.header.header(card_list=lines)
awe> str_header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
awe>
```

In the above example, three headers were made in three different ways from the same source. The first header was loaded from a text file that had one header card of the form 'key = value / comment' per line. This was the text file saved in the previous example with option raw=False. The second header used the PyFITS.CardList instance as its source. The last header used a list of strings loaded from the same file that was the source for the first header.

> NOTE: When using a text file directly for the header source, you MUST use the card_list option of the constructor. The filename option only works for real FITS file or headers that were saved with the default option raw=True. When you load a header from this last type of file, you may get a warning about data size inconsistency. In this case only, it can be safely ignored.

### 7.8.5 Information

Lastly, you can display information about a header object:

```
awe> header = darma.header.header().default()
```

```
awe> header.card_list
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
awe> header.info()
         class: <class 'astro.util.darma.header.header'>
  total length: 4 cards
     (comment): 0 cards
     (history): 0 cards
      itemsize: 80 bytes
     data size: 320 bytes
  size on disk: 2880 bytes
awe>
```

Here you see the class, the total number of cards, the number of comment cards, the number of history cards, and the data sizes of each card, of all the cards, and of the header as stored in a FITS file.

> NOTE: The astute reader should note that slack space of raw FITS-like header files can be considerable. If saving a large number of header files, the most optimal storage size will be achieved using the raw=False option (i.e., plain text output with newline characters).

# Chapter 8

# Astro-WISE Environment

## 8.1 HOW-TO Use the awe-prompt (Python Interpreter)

### 8.1.1 Introduction

Working with the Astro-WISE software involves working with a command line interface. The interface is in fact the standard interactive Python interpreter, customized to facilitate our needs. It is assumed that you have followed the steps in §7.1 (Getting started). You may need to set a number of environment variables again with the following command though:
now start the interpreter:

```
> awe
```

This will print a message such as the following:

```
Python 2.7.6 (default, Jan  8 2014, 14:02:13)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.


                Welcome to the Astro-WISE Environment

|       You are running the AWBASE version

Importing Astro-WISE packages. Please wait...

Distributed Processing Unit: dpu.hpc.rug.astro-wise.org
Dataserver: ds.astro.rug.astro-wise.org

Current profile:
- username : <your database username>
- database : db.astro.rug.astro-wise.org
- project  : <your active project>
- current privileges : 1 (MyDB)

awe>
```

At this point most (all?) of the classes/modules that you need are automatically imported. As this happens, the consistency of those parts of the code that are relevant to the database and

that you may have in your personal version is checked. This takes several seconds. You can see all defined variables (including the classes and modules) in the current so-called namespace by typing:

```
awe> dir()
```

In fact, to get insight in classes, arguments etc., always rely on the combination of:

```
awe> dir(<module, class, attribute or method>)
```

and

```
awe> help(<module, class, attribute or method>)
```

and

```
awe> <module,class,attribute>.__doc__>
```

For example:

```
awe> PhotometricParameters.zeropnt.__doc__
'The response of the instrumental setup [mag]'
```

or if you have an instantation pp of the class PhotometricParameters already:

```
awe> pp.__class__.zeropnt.__doc__
'The response of the instrumental setup [mag]'
```

The `type(<object>)` command returns the type of the object. For example

```
awe> a=1
awe> type(a)
<type 'int'>
awe> photcat=PhotSrcCatalog()
awe> type(photcat)
<class 'astro.main.PhotSrcCatalog.PhotSrcCatalog'>
```

### 8.1.2 Key combinations

The awe-prompt includes functionality from the "readline" library. This library is used in Linux shells and Emacs. Here is a non-exhaustive list of its functionality:

- Tab: Command completion
- Ctrl-a: Go to beginning of line
- Ctrl-e: Go to end of line
- Ctrl-k: Delete in front of cursor to end of line
- Ctrl-u: Delete behind cursor to beginning of line
- Ctrl-p or Up: History search backward (and History up)
- Ctrl-n or Down: History search forward (and History down)

### 8.1.3 Imported package: pylab (plotting)

Along with Astro-WISE packages, **pylab** (matplotlib) is automatically imported. Matplotlib is a powerful Python plotting tool, with close ties to MatLab. Plots can be made for example as follows:

```
awe> x = range(10)
awe> y = [3*i**2 + 10 for i in x]
awe> pylab.scatter(x,y)
```

or

```
awe> pylab.plot(x,y)
```

again, use

```
awe> help(pylab.scatter)
```

and

```
awe> help(pylab.plot)
```

to get help with the syntax.

For more information on matplotlib, see the manual at http://matplotlib.sourceforge.net/.

### 8.1.4 Imported package: numpy (numerical Python)

Operations on large arrays and matrices is the speciality of the **numpy** package. For example:

```
awe> a = numpy.arange(1000000, dtype='float').reshape((1000,1000))
awe> b = numpy.arange(1000000, dtype='float').reshape((1000,1000))
awe> b = b.transpose()
awe> c = a/b
```

### 8.1.5 Imported package: eclipse

See §2.5.2 for more information on this package.

### 8.1.6 Imported packages: os, sys, glob (standard Python)

Along with the above application oriented packages, several standard Python modules are also imported. Two major functionalities of these modules are:

#### System commands

With the **os** module, system commands can be executed for example as follows:

```
awe> os.system('pwd')
awe> os.system('ls')
```

#### Filename lists

With the **glob** module, lists of (existing) files can be easily created using the standard Linux shell wildcards:

```
awe> filenames = glob.glob('OMEGACAM.2014*.fits')
awe> filenames = glob.glob('OMEGACAM.2014-0[3-9]*_?.fits')
```

### 8.1.7 Started: Distributed Processing Unit interface

When the awe-prompt starts, an instance of the class "Processor" is created. Using this class, you can start jobs (tasks) on a remote cluster. See §7.7 and §2.3.3 for information on how to use this class.

## 8.2 Images and catalogs in Astro-WISE

This section gives an overview of all database table names (and Python class names) representing images and catalogs. Where possible a reference/link is provided to the appropriate HOW-TO.

### 8.2.1 Images

| Raw images | Calibration images | Science images |
|---|---|---|
| RawBiasFrame | BiasFrame 12.1 | (RawScienceFrame) |
| RawDarkFrame | DomeFlatFrame 16.1 | ReducedScienceFrame 21.2 |
| RawDomeFlatFrame | TwilightFlatFrame 16.1 | RegriddedFrame 21.4 |
| RawScienceFrame | NightSkyFlatFrame 16.1 | CoaddedRegriddedFrame 21.5 |
| RawTwilightFlatFrame | FringeFrame 17.1 | |
| | IlluminationCorrectionFrame 19.5 | |
| | HotPixelMap 13.1 | |
| | ColdPixelMap 14.1 | |
| | SaturatedPixelMap 20.4 | |
| | SatelliteMap 20.4 | |
| | CosmicMap 20.4 | |
| | WeightFrame 20.4 | |

Table 8.1: Overview of naming scheme for images

### 8.2.2 Catalogs

Below are several tables describing the handling of catalogs in Astro-WISE.

| Catalogs | Description |
|---|---|
| SourceList 21.6 | E.g. SExtractor catalog |
| AssociateList 21.6 | Contains lists of SourceList ID and Source ID pairs |

Table 8.2: Overview of naming scheme for catalogs

| Classes pointing to SourceLists | Description |
|---|---|
| GalFitModel 22.1 | Galfit model of a single source in a SourceList |
| GalPhotModel 22.3 | Galphot model of a single source in a SourceList |

Table 8.3: Overview of naming scheme for classes pointing to SourceLists

| **Classes creating new SourceList(s)** | **Description** |
|---|---|
| CombinedList 21.6 | A SourceList which is the combination of two other SourceLists |
| PhotRedCatalog 22.4 | Links a new SourceList with photometric redshifts to all associated input SourceLists and an AssociateList |
| LightCurve 22.5 | (MDIA) Creates several SourceLists to store result of variability determination |

Table 8.4: Overview of naming scheme for classes creating new SourceLists

| **Other catalogs** | **Description** |
|---|---|
| VariabilityFrame 22.7 | (VODIA) Stores a catalog in file format with light curves |

Table 8.5: Overview of naming scheme for other classes related to catalogs

## 8.3 HOW-TO Query the Database from Python

In order to query data in the database or *commit* data to it, an interface to SQL (Structured Query Language, the standard command driven query interface for databases) was written. Using the interface it is possible to query the database from Python scripts or the Python interpreter and obtain complete Python objects, with their entire history in the form of their normal hierarchy intact. Conversely Python objects with their entire history intact can be committed to the database for later retrieval. The interface supports a number of often used SQL constructs that are described in the following subsections.

### 8.3.1 General syntax, comparison operators, AND and OR

A database query from Python generally has this structure:

```
awe> query = <class>.<attribute> <comparison operator> <value>
```

Where class can be any DBObject (objects that are stored in the database) and attribute can be any attribute of any DBObject and the hierarchy of a DBObject can be followed as deep as it goes. Comparison operators can be all the usual: ==, ! =, >, >=, <, <= (equal to, not equal to, greater than, greater than or equal, smaller than, smaller than or equal respectively). Note that a list of persistent properties (those properties that can be queried on in the database) can be obtained for all ProcessTargets (use the class NOT an instance of the class) as follows:

```
awe> BiasFrame.get_persistent_properties()
['chip', 'creation_date', 'filename', 'globalname', 'imstat', 'instrument',
'is_valid', 'object_id', 'observing_block', 'prev', 'process_params',
'process_status', 'quality_flags', 'raw_bias_frames', 'read_noise',
'timestamp_end', 'timestamp_start']
```

and

```
awe> RawScienceFrame.get_persistent_properties()
['AIRMEND', 'AIRMSTRT', 'DATE', 'DATE_OBS', 'EXPTIME', 'LST', 'MJD_OBS',
'OBJECT', 'OBSERVER', 'UTC', 'astrom', 'chip', 'extension', 'filename',
'filter', 'globalname', 'imstat', 'instrument', 'is_valid', 'object_id',
'observing_block', 'overscan_x_stat', 'overscan_y_stat', 'prescan_x_stat',
'prescan_y_stat', 'process_status', 'quality_flags', 'raw_fits_data',
'template']
```

So, an example of a query is:

```
awe> query = RawScienceFrame.EXPTIME >= 300.0
```

or

```
awe> query = RawTwilightFlatFrame.imstat.median < 30000.0
```

Queries can be comprised of multiple parts separated by AND (&) or OR (|) operators, where the different parts are between parentheses:

```
awe> q = (RawScienceFrame.OBJECT == 'ngc6822')
awe> q = (RawScienceFrame.OBJECT == 'ngc6822') | \
         (RawScienceFrame.OBJECT == 'ngc 6752 - Field')
awe> q = (RawScienceFrame.OBJECT == 'ngc6822') & \
         (RawScienceFrame.chip.name == 'ESO_CCD_#65')
```

Note that the backslashes at the end of the lines only indicate that the command continues on the next line. Lengths of queries (number of results) can be obtained using the Python *len* function:

```
awe> len(q)
110
```

Attributes of the obtained objects can be printed as follows:

```
awe> for f in q: print f.filename, f.OBJECT, f.filter.name, f.chip.name, f.EXPTIME
...
OMEGACAM.2012-06-16T05:51:25.429_1.fits ngc6822 OCAM_u_SDSS ESO_CCD_#65 580.0
OMEGACAM.2012-06-16T06:46:22.166_1.fits ngc6822 OCAM_u_SDSS ESO_CCD_#65 580.0
OMEGACAM.2012-07-26T02:30:04.947_1.fits ngc6822 OCAM_H_ALPHA ESO_CCD_#65 580.0
OMEGACAM.2012-07-19T03:17:24.651_1.fits ngc6822 OCAM_g_SDSS ESO_CCD_#65 580.0
OMEGACAM.2012-06-16T06:25:32.663_1.fits ngc6822 OCAM_u_SDSS ESO_CCD_#65 580.0
OMEGACAM.2012-06-16T05:30:37.246_1.fits ngc6822 OCAM_u_SDSS ESO_CCD_#65 580.0
OMEGACAM.2012-06-16T07:07:10.329_1.fits ngc6822 OCAM_u_SDSS ESO_CCD_#65 580.0
OMEGACAM.2012-06-15T07:23:26.839_1.fits ngc6822 OCAM_H_ALPHA ESO_CCD_#65 580.0
OMEGACAM.2012-06-15T07:33:51.550_1.fits ngc6822 OCAM_H_ALPHA ESO_CCD_#65 580.0
OMEGACAM.2012-07-26T02:19:39.575_1.fits ngc6822 OCAM_H_ALPHA ESO_CCD_#65 580.0
etc.
```

It is also possible to construct a query object without a query clause. This query can be used to iterate through all objects of the specific class.

```
awe> query = <class>.select_all()
```

### 8.3.2 Using wildcards (like)

It is possible to use wildcards in particular when selecting using strings. Wildcards are implemented as they are in the common UNIX shells, (? for any character, * for any number of characters).

```
awe> q = (RawScienceFrame.instrument.name == 'OMEGACAM') & \
         (RawScienceFrame.OBJECT.like('ngc*'))

awe> q = RawScienceFrame.filename.like('OMEGACAM.2013-06-02T06:51:04.629_?.fits')
awe> for f in q: print f.filename, f.OBSERVER, f.DATE_OBS, f.filter.name, f.EXPTIME
...
OMEGACAM.2013-06-02T06:51:04.629_1.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_2.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_3.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_4.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_5.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_6.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_7.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_8.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
OMEGACAM.2013-06-02T06:51:04.629_9.fits UNKNOWN 2013-06-02 06:51:04 OCAM_r_SDSS 600.0
```

### 8.3.3 Querying list attributes (contains)

If an attribute points to a list the method `contains` can be used to query for elements in this list. The elements in this list can be simple types like int, float and string, or persistent classes. The method `contains` accepts a singe element or a list of elements. In case of a list all elements in this list must be present in the queried attribute. The order of the elements in the list is not taken into account.

```
awe> q1 = CoaddedRegriddedFrame.instrument.name == 'OMEGACAM'
awe> regrid1 = q1[0].regridded_frames[0]
awe> regrid2 = q1[0].regridded_frames[1]
awe> q2 = CoaddedRegriddedFrame.regridded_frames.contains(regrid1)
awe> len(q2)
2
awe> q2 = CoaddedRegriddedFrame.regridded_frames.contains( [regrid1, regrid2] )
awe> len(q2)
2
```

First the CoaddedRegriddedFrames are found which have regrid1 in their regridded_frames attribute. Then all CoaddedRegriddedFrames which have both regrid1 and regrid2.

```
awe> q = AstrometricParameters.FITPARMS.contains(0.0023721800000000002)
awe> len(q)
2
awe> q = AstrometricParameters.FITPARMS.contains([-0.0023634099999999998, -5.41757e-06])
awe> len(q)
1
```

In the above examples the FITPARMS attribute of AstrometricParameter are queried. First for a single value, then a list.

### 8.3.4 Ordering by attribute values (order_by)

It is possible to order a query by one of the attributes of the objects. Note that this alters the returned list.

```
awe> q = (RawScienceFrame.OBJECT == 'ngc6822') & \
         (RawScienceFrame.chip.name == 'ESO_CCD_#88')
awe> for f in q: print f.filename, f.DATE_OBS, f.filter.name, f.EXPTIME
OMEGACAM.2012-06-16T06:46:22.166_28.fits 2012-06-16 06:46:22 OCAM_u_SDSS 580.0
OMEGACAM.2012-06-16T05:51:25.429_28.fits 2012-06-16 05:51:25 OCAM_u_SDSS 580.0
OMEGACAM.2012-07-19T03:17:24.651_28.fits 2012-07-19 03:17:24 OCAM_g_SDSS 580.0
OMEGACAM.2012-06-16T06:25:32.663_28.fits 2012-06-16 06:25:32 OCAM_u_SDSS 580.0
OMEGACAM.2012-07-26T02:30:04.947_28.fits 2012-07-26 02:30:04 OCAM_H_ALPHA 580.0
OMEGACAM.2012-06-16T07:07:10.329_28.fits 2012-06-16 07:07:10 OCAM_u_SDSS 580.0
etc.
etc.
awe> for f in q.order_by('DATE_OBS'): print f.filename, f.DATE_OBS, f.filter.name, f.EXPTIME
OMEGACAM.2012-06-02T06:26:52.601_28.fits 2012-06-02 06:26:52 OCAM_u_SDSS 580.0
OMEGACAM.2012-06-02T06:37:18.057_28.fits 2012-06-02 06:37:18 OCAM_u_SDSS 580.0
OMEGACAM.2012-06-02T06:47:41.774_28.fits 2012-06-02 06:47:41 OCAM_u_SDSS 580.0
OMEGACAM.2012-06-15T07:23:26.839_28.fits 2012-06-15 07:23:26 OCAM_H_ALPHA 580.0
```

```
OMEGACAM.2012-06-15T07:33:51.550_28.fits 2012-06-15 07:33:51 OCAM_H_ALPHA 580.0
OMEGACAM.2012-06-15T07:44:16.252_28.fits 2012-06-15 07:44:16 OCAM_H_ALPHA 580.0
etc.
etc.
```

### 8.3.5 Ordering returning maximum, minimum (max, min)

It is possible to select from a selection of objects the one with the maximum or minimum of a particular attribute:

```
awe> q = (RawScienceFrame.OBJECT == 'ngc6822') & \
         (RawScienceFrame.chip.name == 'ESO_CCD_#88')
awe> ma = q.max('EXPTIME')
awe> print ma.DATE_OBS, ma.filter.name, ma.EXPTIME
2013-05-07 08:59:37 OCAM_g_SDSS 600.0
awe> mi = q.min('EXPTIME')
awe> print mi.DATE_OBS, mi.filter.name, mi.EXPTIME
2012-06-16 06:46:22 OCAM_u_SDSS 580.0
awe> latest = q.max('DATE_OBS')
awe> print latest.filename, latest.DATE_OBS, latest.EXPTIME
OMEGACAM.2013-07-02T04:43:48.792_28.fits 2013-07-02 04:43:48 600.0
```

### 8.3.6 Querying project specific data (project_only)

It is possible to restrict the results of a query to objects of the currently set project or a specific project. The following example first shows the length of a query for all the public data, then for the currently set project and last for a specific project. Note that the project_only method is sticky, it will affect future usage of the query object.

```
awe> context.set_project('ALL')
awe> q = CoaddedRegriddedFrame.instrument.name == 'OMEGACAM'
awe> len(q)
213
awe> len(q.project_only())
0
awe> len(q.project_only('KIDS'))
204
```

First the project ALL is set, then all CoaddedRegriddedFrames are queried which have as instrument name OMEGACAM. Then only those specific to project ALL, and finally those visible from project ALL, but in project KIDS. Instead of the project name the (numerical) project id can also be used to indentify the project.

If you set the environment variable PROJECT_ONLY to a project name or id then all queries will use the project_only method automatically with this project. If set to True then the current project will be used, and if set to False the environment variable will be ignored.

### 8.3.7 Querying user specific data (user_only)

It is possible to restrict the results of a query to objects created by the current or a specific user. The following example first shows the length of a query for all the public data, then for the current user and last for a specific user. Note that the user_only method is sticky, it will affect future usage of the query object.

```
awe> from common.database.Database import database
awe> context.set_project('ALL')
awe> print database.username().upper()
'AWEHELMICH'
awe> q = CoaddedRegriddedFrame.instrument.name == 'OMEGACAM'
awe> len(q)
213
awe> len(q.user_only())
9
awe> len(q.user_only(user='AWJMCFARLAND'))
204
```

First the project ALL is set and the current user is printed. Then all CoaddedRegridded-Frames are queried which have as instrument name OMEGACAM Then only the Regridded-Frames created by the current user are printed. And finally the RegriddedFrames created by AWJMCFARLAND are shown. Instead of the user name the (numerical) user id can also be used to indentify the user.

If you set the environment variable USER_ONLY to an user name or id then all queries will use the user_only method automatically. If set to True then the current user will be used, and if set to False the environment variable will be ignored.

### 8.3.8 Querying privileges specific data (privileges_only)

It is possible to restrict the results of a query to objects having specific privileges. When an argument is omitted the current privileges are used, otherwise the specified. The following example first shows the length of a query for all the visible data, then for the current privileges (1) and last for privileges of 5. Note that the privileges_only method is sticky, it will affect future usage of the query object.

```
awe> context.set_project('ALL')
awe> context.set_privileges(1)
awe> q = CoaddedRegriddedFrame.instrument.name == 'OMEGACAM'
awe> len(q)
213
awe> len(q.privileges_only())
9
awe> len(q.privileges_only(5))
0
```

First the project ALL is set and the current privileges are set to 1. Then all CoaddedRegrid-dedFrames are queried which have as instrument name OMEGACAM. Then only those with privileges of 1 are printed. Finally those with privileges 5 are shown.

If you set the environment variable PRIVILEGES_ONLY to a privileges number then all queries will use the privileges_only method automatically. If set to True then the current privi-leges will be used, and if set to False the environment variable will be ignored.

### 8.3.9 Project favourite (project_favourite)

The project_favourite flag is intended to favor the (calibration) data owned by the project above data from other projects. This is implemented by adjusting the order in which results from a query are returned. The creation_date will still be used to get the newest version, but if data

is present in the current project that will be used instead of (possible) newer data from other projects.

The project_favourite flag can be enabled in two ways; on query level and environment level. To make a query project_favourite call the project_favourite method on the query. To make all queries project_favourite set the Environment setting PROJECT_FAVOURITE to True. The default setting is not to use project_favourite.

The following example shows the usage of the project_favourite on the command line. It shows the maximum creation_date of all BiasFrames in the KIDS project, and then the maximum creation_date of all BiasFrames in the current project (ALL) :

```
awe> context.set_project('KIDS')
awe> q = BiasFrame.instrument.name == 'OMEGACAM'
awe> q.max('creation_date').creation_date
datetime.datetime(2014, 8, 21, 7, 37, 58)
awe> context.set_project('ALL')
awe> q.project_favourite().max('creation_date').creation_date
datetime.datetime(2014, 1, 16, 15, 50, 33)
```

### 8.3.10 Related: retrieving images from the fileserver (retrieve)

The final step of a database query could very well be to retrieve images selected in the database from the fileserver in order to look at them.

```
awe> q = (RawScienceFrame.OBJECT == 'ngc6822') & \
         (RawScienceFrame.chip.name == 'ESO_CCD_#65')
awe> len(q)
110
awe> for f in q: f.retrieve()
...
[smyth] 2014-08-21T11:34:40 - Retrieving OMEGACAM.2012-06-16T05:51:25.429_1.fits
[smyth] 2014-08-21T11:34:40 - Retrieved OMEGACAM.2012-06-16T05:51:25.429_1.fits[5695kB] in 0.28 sec
[smyth] 2014-08-21T11:34:40 - Running: imcopy 'OMEGACAM.2012-06-16T05:51:25.429_1.fits[1]' 'tmpFJLy
[smyth] 2014-08-21T11:34:41 - Retrieving OMEGACAM.2012-06-16T06:46:22.166_1.fits
[smyth] 2014-08-21T11:34:41 - Retrieved OMEGACAM.2012-06-16T06:46:22.166_1.fits[5658kB] in 0.14 sec
[smyth] 2014-08-21T11:34:41 - Running: imcopy 'OMEGACAM.2012-06-16T06:46:22.166_1.fits[1]' 'tmpeQXP
[smyth] 2014-08-21T11:34:41 - Retrieving OMEGACAM.2012-07-26T02:30:04.947_1.fits
[smyth] 2014-08-21T11:34:41 - Retrieved OMEGACAM.2012-07-26T02:30:04.947_1.fits[6015kB] in 0.22 sec
[smyth] 2014-08-21T11:34:41 - Running: imcopy 'OMEGACAM.2012-07-26T02:30:04.947_1.fits[1]' 'tmpdufY
[smyth] 2014-08-21T11:34:42 - Retrieving OMEGACAM.2012-07-19T03:17:24.651_1.fits
etc.
etc.
```

Note that *imcopy* is run to decompress these images, which are stored in compressed format on the fileserver.

### 8.3.11 The select method, quicker queries

*Note that this method automatically ignores invalid data*

Constructing queries as above can be a somewhat verbose affair. To facilitate easier querying for which less input is necessary, the *select* method has been implemented for all ProcessTargets. The above queries can be written for example as follows:

```
awe> q = RawScienceFrame.select(instrument='OMEGACAM', chip='ESO_CCD_#65', object='ngc6822')
```

For a complete list of possible arguments of the select method see its docstring:

```
awe> help(DomeFlatFrame.select)
Help on method select in module astro.main.ProcessTarget:

select(cls, **searchterms) method of astro.database.DBMeta.DBObjectMeta instan
ce
    Class method to select RawFrames, Calfiles and ReducedScienceFrames
    from the database.

    Syntax example:

    s = RawScienceFrame.select(instrument='WFI', filter='#842',
        chip='ccd50', time_from='2000-01-02 04:45:46',
        time_to='2000-01-02 05:03:00')

    Possible search terms:
    ----------------------
    chip        - select of the same CCD ('ccd50', 'ccd51', etc.)
    date        - select of the same date (i.e. date at the start of
                  observing night, in yyyy-mm-dd format)
    exptime     - select frames with similar exposure time
                  (EXPTIME-0.8sec to EXPTIME+0.8 sec)
    extension   - select (raw) frames for a certain extension of its
                  RawFitsData object
    filename    - select a frame(!) by its filename
    filter      - select of the same filter ('#842', '#843', etc.)
    instrument  - select of the same instrument ('WFI', 'WFC', 'OCAM')
    object      - select for OBJECT header keyword, uses "like"
                  functionality, which allows wildcards "*" and "?"
    time_from   - precise form of date, in yyyy-mm-dd hh:mm:ss format
    time_to     - required when using time_from
```

### 8.3.12 More examples

*Question:* How do I query using dates? *Answer:* In general you need to make a datetime object specifying an exact time in UTC for your date to be recognized. All times and dates in the database are in UTC.

```
awe> date = datetime.datetime(2014,7,1)
awe> query = RawDomeFlatFrame.DATE_OBS > date
awe> query = (ReducedScienceFrame.creation_date > date) &\
...  (ReducedScienceFrame.creation_date < date+datetime.timedelta(1))
```

In other cases dates are not datetime objects, in particular when given as arguments to methods or objects. In these cases the dates are meant as the starting date of a **night**. A night is defined as the period between noon on one day and noon the next day. This concept is used to define whether or not calibration files are applicable to a given set of science images.

```
awe> task = ReduceTask(date='2014-06-10', instrument='OMEGACAM', \
```

```
                        filter='OCAM_r_SDSS', chip='ESO_CCD_#77)
awe> bias = BiasFrame.select(date='2014-07-05', instrument='OMEGACAM', \
                             chip='ESO_CCD_#96')
```

The final query, when not using the select method looks like this:

```
awe> midnight = datetime.datetime(2014,7,5) + datetime.timedelta(1)
awe> instrument = (Instrument.name == 'OMEGACAM')[0]
awe> midnight = instrument.convert_local_to_ut(midnight)
awe> query = (BiasFrame.timestamp_start < midnight) & \
             (BiasFrame.timestamp_end > midnight)
awe> query &= (BiasFrame.instrument.name == 'OMEGACAM')
awe> query &= (BiasFrame.chip.name == 'ESO_CCD_#96')
awe> bias = query.max('creation_date')
```

*Question:* When can I use wildcards in queries? *Answer:* When using the "like" method and only for strings, or in Tasks and the select method in the object argument:

```
awe> query = RawDomeFlatFrame.filename.like('OMEGACAM.2014-08-11*')
awe> task = ReduceTask(date='2014-08-10', instrument='OMEGACAM', object='KIDS*')
awe> query = ReducedScienceFrame.select(object='KIDS*', chip='ESO_CCD_#65')
```

Suppose you've just processed a significant amount of data, and are then interested in finding out some properties that you know are stored in the database. How do you get this information?

*Question:* Give me all image statistics (for example median values) of all OMEGACAM raw bias frames of a particular CCD, observed between two dates:

```
awe> q = (RawBiasFrame.instrument.name == 'OMEGACAM') & \
         (RawBiasFrame.chip.name == 'ESO_CCD_#77') & \
         (RawBiasFrame.DATE_OBS > datetime.datetime(2014,7,1)) & \
         (RawBiasFrame.DATE_OBS < datetime.datetime(2014,7,10))
awe> for f in q.order_by('DATE_OBS'): print f.filename, f.imstat.median
etc.
etc.
OMEGACAM.2014-07-02T11:20:14.159_21.fits 256.0
OMEGACAM.2014-07-02T11:20:56.539_21.fits 256.0
OMEGACAM.2014-07-02T11:37:41.991_21.fits 256.0
OMEGACAM.2014-07-02T11:38:24.101_21.fits 256.0
OMEGACAM.2014-07-03T10:54:22.714_21.fits 263.0
OMEGACAM.2014-07-03T10:55:04.814_21.fits 263.0
OMEGACAM.2014-07-03T10:55:48.834_21.fits 263.0
etc.
etc.
```

A very long list of filename, median pixel value pairs will be printed on screen. (You can abort with Ctrl-C.)

*Question:* Give me all the RawScienceFrames for the OMEGACAM instrument, ccd #88, filter r and for object starting with "NGC".

```
awe> query = (RawScienceFrame.instrument.name == 'OMEGACAM') &\
             (RawScienceFrame.chip.name == 'ESO_CCD_#88') &\
```

```
            (RawScienceFrame.filter.name == 'OCAM_r_SDSS') &\
            (RawScienceFrame.OBJECT.like('NGC*'))
awe> for f in query: print f.filename, f.instrument.name, f.chip.name, \
...  f.filter.name, f.OBJECT, f.EXPTIME
OMEGACAM.2011-10-30T07:35:44.043_28.fits OMEGACAM ESO_CCD_#88 OCAM_r_SDSS NGC 1399 280.0
OMEGACAM.2011-10-30T06:50:51.776_28.fits OMEGACAM ESO_CCD_#88 OCAM_r_SDSS NGC 1399 280.0
OMEGACAM.2011-10-30T06:45:26.873_28.fits OMEGACAM ESO_CCD_#88 OCAM_r_SDSS NGC 1399 280.0
OMEGACAM.2011-10-30T06:56:15.459_28.fits OMEGACAM ESO_CCD_#88 OCAM_r_SDSS NGC 1399 280.0
etc.
etc.
```

*Question:* Give me all RawTwilightFlatFrames observed between two points in time.

```
awe> query = RawTwilightFlatFrame.select(time_from='2014-08-18T16:00:00', \
...  time_to='2014-08-19T16:00:00')
```

*Question:* Select the most recent OMEGACAM MasterFlatFrame from the database, that is valid for the specified night.

```
awe> flat = MasterFlatFrame.select(instrument='OMEGACAM', date='2014-07-13', \
                                   filter='OCAM_g_SDSS', chip='ESO_CCD_#90')
```

## 8.4 HOW-TO Configure Process Parameters

### 8.4.1 Overview

In the figure below an overview is presented of how one can configure process parameters. The preferred way to configure process parameters is through the overall user interfaces found in the Target Processor webservice, and/or in the **Pars** class, which can be used in scripts and from the `awe`-prompt.

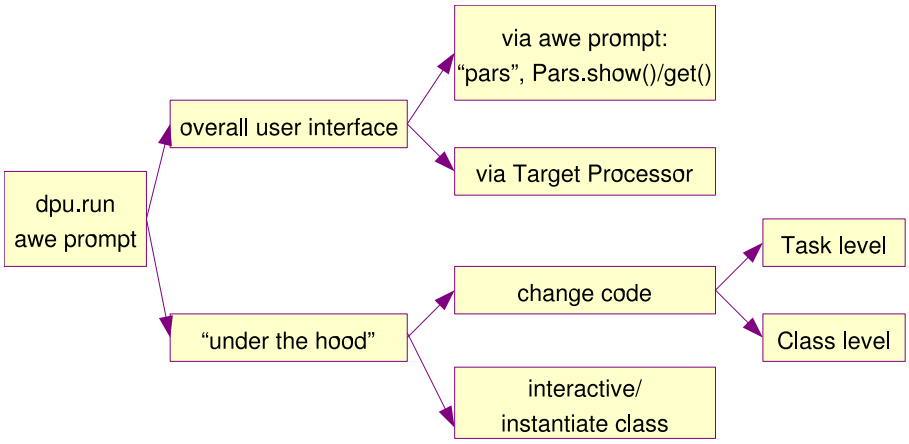



Figure 8.1: There are many ways to configure process parameters

### 8.4.2 Via `awe`-prompt: overall user interface to configure parameters

**via `awe`-prompt:** `pars, pars.show(), pars.get()`

Process parameters can be configured from the `awe`-prompt, both on the task, and on the DPU level. The first question that you may have is: *Which process parameters can I control, and what are their names and default values?* From the `awe`-prompt:

```
awe> pars = Pars(BiasFrame, instrument='OMEGACAM', chip='ESO_CCD_#77')
awe> # or pars=Pars(BiasTask) or pars=Pars('Bias') !!
awe> pars.show()
BiasFrame
 |
 +--process_params
 |   |
 |   +--MAXIMUM_ABS_MEAN: 10.0
 |   +--MAXIMUM_STDEV: 10.0
 |   +--MAXIMUM_STDEV_DIFFERENCE: 10.0
 |   +--MAXIMUM_SUBWIN_FLATNESS: 100000.0
 |   +--MAXIMUM_SUBWIN_STDEV: 100000.0
 |   +--OVERSCAN_CORRECTION: 6
 |   +--SIGMA_CLIP: 3.0
```

In other words, the Pars class can be instantiated with as argument a ProcessTarget class (BiasFrame, SourceList etc.) or a Task (ReduceTask, ReadNoiseTask etc.) or a sequence identifier

as specified in the **dpu.run** method ("Bias", "Reduce>Astrometry"). Note that in the last case a **string** is specified, while in the others a **class** is specified. Additionally an instrument, filter, or chip identifier can be specified in order to generate instrument specific defaults. The **show()** method of Pars displays the configurable parameters.

The parameters can now be changed (tip to reduce typing: press the Tab key get automatic command completion):

```
awe> pars.BiasFrame.process_params.SIGMA_CLIP=5.0
```

In order to use the parameters, you must place them in a dictionary, which is produced by the **get()** method of the Pars class:

```
awe> pars.get()
{'BiasFrame.process_params.SIGMA_CLIP': 5.0}
```

The output can be used as follows:

```
awe> dpu.run('Bias', instrument='OMEGACAM', template='2014-07-04T10:39:28',
             chip='ESO_CCD_#77', p=pars.get())
```

or

```
awe> task = BiasTask(instrument='OMEGACAM', template='2014-07-04T10:39:28',
                     chip='ESO_CCD_#77', pars=pars.get())
```

In other words, a dictionary called "pars", is sent to the Task (or the DPU). You can enter the dictionary directly like this:

```
awe> task = BiasTask(instrument='OMEGACAM', template='2014-07-04T10:39:28',
                     chip='ESO_CCD_#77',
                     pars={'BiasFrame.process_params.SIGMA_CLIP': 5.0})
awe> task.execute()
```

Equivalently using the DPU interface:

```
awe> dpu.run('Bias', i='OMEGACAM', tpl='2014-07-04T10:39:28',
             c='ESO_CCD_#77', p={'BiasFrame.process_params.SIGMA_CLIP' : 5.0})
```

**Using the interface inside a script**

It is possible to use the Pars class and the DPU interface inside scripts as follows:

```
# Example script
from astro.recipes.mods.dpu import Processor
from common.config.Environment import Env
from astro.util.Pars import Pars

dpu = Processor(Env['dpu_name'])
pars = Pars(BiasFrame, instrument='OMEGACAM', chip='ESO_CCD_#77')
pars.BiasFrame.process_params.OVERSCAN_CORRECTION=8
dpu.run('Bias', i='OMEGACAM' tpl='2014-07-04T10:39:28', c='ESO_CCD_#77', p=pars.get())
```

### 8.4.3 Via Target Processor: overall user interface to configure parameters

See the Target Processor web page http://process.astro-wise.org/. In particular note the "Process Parameters" link under "Options" at the bottom left.

## 8.5 HOW-TO use your Astro-WISE `Context` to set data scopes

The `Astro-WISE` concept `Context` allows you to filter from the ocean of data objects in Astro-WISE the subset of data objects that you want to be currently visible to you and your processes. This is the 'data access scope' of your `Context`. At the same time your chosen `Context` also defines the logical subset to which the results of your processes will belong. This is the 'data creation scope' of your current `Context`. In other words, by configuring your `Astro-WISE Context` you define the logical subspaces in the ensemble of data objects in `Astro-WISE` in which you access data and in which you create data.

### 8.5.1 Astro-WISE Context

Your `Astro-WISE Context` is fully defined by choosing the settings of three `Astro-WISE` entities:

1. `Astro-WISE user` identity
2. `project`
3. minimum `privileges` level

#### User

An Astro-WISE user is a person with an Astro-WISE database account. An Astro-WISE `user` has an id number and name (e.g., AWJJOPLIN). Each person has only one account. Thus, each person has a single identity within the `Astro-WISE` system. Each data object is associated with a single Astro-WISE `user`, the data object creator. This is the Astro-WISE `user` that created/ingested the data. Once established the creator of a data object cannot be changed.

At the `awe`-prompt the creator of data object `myobject` can be printed to the screen with

```
awe> myobject._creator # Returns user id
990
awe> from common.database.Database import Database
awe> Database.users[myobject._creator] # Returns user name
'AWJJOPLIN'
```

#### Project

A `project` in `Astro-WISE` is a set of `Astro-WISE users` who collaborate on a common data set. A `project` has a project id, a name, a description, `project` members and optionally an instrument. One or more `Astro-WISE users` can be member of a `project`, and an `Astro-WISE user` can be member of more than one `project` as illustrated in Figure 8.2. Unlike an user each data object belongs to one and only one `project`. The `project` to which a data object belongs is chosen upon the creation/ingestion of the data entity and can not be changed after that.

At the `awe`-prompt the `project` to which data object `myobject` belongs can be printed to the screen with

```
awe> myobject._project #Returns project id
14
awe> from common.database.Database import Database
awe> Database.projects[myobject._project] # Returns project name
'WFI@2.2m'
```

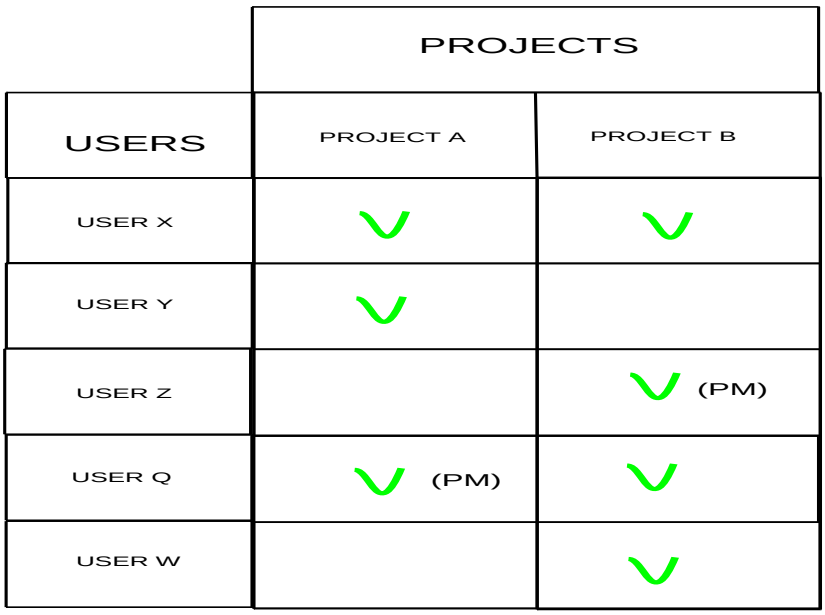



Figure 8.2: This diagram shows which `Astro-WISE users` are `project` members of `Astro-WISE project` A and B. A `project` in `Astro-WISE` is a set of `Astro-WISE users` who collaborate on a common data set. `Project` members are indicated in the diagram by a green tick mark. An `Astro-WISE user` can be member of more than one `project`. Each `project` has one or more `Astro-WISE users` as `project` members. One or more of the `project` members can be `project` manager (indicated with (pm)). A `project` manager has additional abilities and responsibilities to manage project data.

| privileges level | data is shared with |
|---|---|
| **1: MYDB** | only the creator |
| **2: PROJECT** | every member of the project to which the data object belongs |
| **3: ASTRO-WISE** | all Astro-WISE users |
| **4: WORLD** | the world: Astro-WISE users and persons without an Astro-WISE account (latter via webservice DBViewer) |
| **5: VO** | again the whole world and data also available through the Virtual Observatory via Virtual Observatory webservices |

Table 8.6: Privilege levels of Astro-WISE

Some projects have all Astro-WISE users as members: these are called public projects. Some projects have a subset of Astro-WISE users as members: these are called private projects. One or more of the Astro-WISE users in a project can also act as project manager. A project manager has additional abilities and responsibilities to manage data objects that belong to a project. The extra abilities/responsibilities are for data objects at privileges levels of 3 and larger. At these privileges levels the data objects are visible to persons which are not project members. The concept of privileges is explained next.

An overview of all projects in the database, their members, the project's manager etc. can be found on our webpages at the following address:
http://process.astro-wise.org/Projects.

#### Privileges

Privileges of data objects determine which Astro-WISE users have access to the data object. Each data object in Astro-WISE has one of five privileges as listed in Table 8.6.
The initial privileges of a data object are set upon the creation/ingestion of the data entity. They can be changed after that.

At the awe-prompt the privileges of data object myobject can be printed to the screen with

```
awe> myobject._privileges
1
```

### 8.5.2 Using Context

Your Astro-WISE Context is defined by your Astro-WISE user identity, the current project you choose and the current minimum privileges level that you choose. The Astro-WISE user that you are is defined upon login as you login with a certain user name (e.g. AWJJOPLIN). Upon login, also a current project and a current minimum privileges level are set. These two initial selections depend on your login configuration. At the Astro-WISE command-line prompt, the python class context is the interface to view and change the selection of project and privileges level. You can import context at the Astro-WISE prompt via:

```
awe> from common.database.Context import context
```

To see which minimum privileges level is currently active use context.get_current_privileges()

```
awe> context.get_current_privileges()
1
```

To change the minimum privileges level call the context.set_privileges(<privileges>) method :

```
awe> context.set_privileges(2)
```

To see which `project` is currently selected, use `context.get_current_project()`. If the project object is printed, a formatted overview will be given:

```
awe> print context.get_current_project()
   Project:     SONATE  (id = 14)
   Description: 1700 Square degree 5 band survey of the equatorial strip and 2dF
                South region
   Instrument:  OCAM
   Maximum Privileges: 4 (WORLD)
   Current Privileges: 1 (MyDB)
```

`Instrument` lists the instrument associated with the `project`. If it is `None` it means data from any instrument can be associated with the `project`. `Maximum Privileges` is the maximum privileges level which a project member can assign to a data object. To change to a `project` called 'BLUES' at the `awe`-prompt use:

```
awe> context.set_project('BLUES')
```

An user can only set `projects` of which the user is a member. When a project is selected without specifying the minimum `privileges` level they will be set to their default: 1.
To select at the same time the `project` and the minimum `privileges` level at the `awe`-prompt use:

```
awe> context.set_project('BLUES', privileges=2)
```

### Data access scope

The philosophy in `Astro-WISE` is to increase efficiency in survey processing by sharing useful public data (such as calibration data). The philosophy of `Context` is therefore to allow users to select which data they want to see in their data access scope *in addition* to public data from `Astro-WISE projects`.

Whether a data object falls within the data access scope of the `Context` as currently configured depends on the values of 3 attributes which each data object has:

- `_privileges`
- `_project`
- `_creator`

These attributes are compared to the corresponding 3 entities which configure your `Context`: the chosen values for `project`, minimum `privileges` level and your `Astro-WISE user` identity.

The minimum `privileges` level chosen for your `Context` defines from which `privileges` level on you want to have access to data objects. Thus, by lowering the minimum `privileges` level you add data objects to your data access scope. A minimum `privileges` level of $i$ indicates data objects with `privileges` levels $x \geq i$ will fall in your data access scope. For `privileges` levels $\geq 3$ all data objects with that `privileges` level fall into your data scope. For `privileges` levels $\leq 2$ a subset of the data objects with that `privileges` level belong to your data scope. Table 8.7 lists the rules which decide which data objects fall within your data access scope. Figure 8.3 shows a graphical representation of the resulting data access scope for each of the five minimum `privilegels` levels that can be selected for your `Context`.

As an example, assume that you select your `Context` to have `project`='BLUES' and minimum `privileges` level=1. Then your data access scope will include

Table 8.7: This diagram illustrates the data access scope for User X who has selected PROJECT A in his Context. Together with the selected level from the 5 possible minimum `privileges` levels in `Context` it defines the data access scope. The diagram on top shows how sets of data objects are added to your data access scope as the minimum `privileges` level set in the `Context` is lowered. The table below it describes each of these sets which are defined as the data objects which have the values of the attributes `_privileges`, `_project` and `_creator` listed in the columns.

| minimum `privileges` level of `Context` | data access scope |
|---|---|
| 5 | 5 |
| 4 | 5 + 4 |
| 3 | 5 + 4 + 3 |
| 2 | 5 + 4 + 3 + 2 |
| 1 | 5 + 4 + 3 + 2 + 1 |

where the data represented by 5, 4,... meet the following criteria for their attributes `_privileges`, `_project` and `_creator`:

| dataset | `_privileges==` | `_project==` | `_creator==` |
|---|---|---|---|
| 5 = | 5 | ANY `project` | ANY `creator` |
| 4 = | 4 | ANY `project` | ANY `creator` |
| 3 = | 3 | ANY `project` | ANY `creator` |
| 2 = | 2 | PROJECT A | ANY `creator` |
| 1 = | 1 | ANY `project` of which USER X is a member | USER X |

Table 8.8: This table lists to the `privileges` levels to which the `creator` and project manager of a `project` can publish data. The notation format is (possible start levels) ⇒ (possible end levels). The permitted levels are different for public and private `projects`.

| `project` type | `creator` | `project` manager |
|---|---|---|
| private | $(1) \Rightarrow (2)$ | $(2,3,4) \Rightarrow (3,4,5)$ |
| public | $(1,2,3) \Rightarrow (2,3,4)$ | $(2,3,4) \Rightarrow (3,4,5)$ |

- data with `privileges`==1 which are created by you in any `project`
- data with `privileges`==2 which are in `project`='BLUES'
- data with `privileges`≥ 3 in all `projects`, also those of which you are **not** a member

### Data creation scope

Upon creation/ingestion of an object the values of its attributes `_project`, `_privileges` and `_creator` are :

- `_project` set to the id of the `project` of your `Context`
- `_privileges` set to the minimum `privileges` level of your `Context`
- `_creator` set to the id of your `Astro-WISE user` identity

Only `project` members can ingest/create data in a `project`. Data objects can not be created and ingested at all `privileges` levels. In private `projects`, the `project` members can create and ingest data objects only at `privileges` levels 1 and 2. Only `project` managers can then promote data objects to higher `privileges` levels up to `privileges` levels equal 5. This is called `publishing` and is described in the next section. In public `projects` the `Astro-WISE users` can ingest and create data object up to all `privileges` levels up to a maximum of 4.

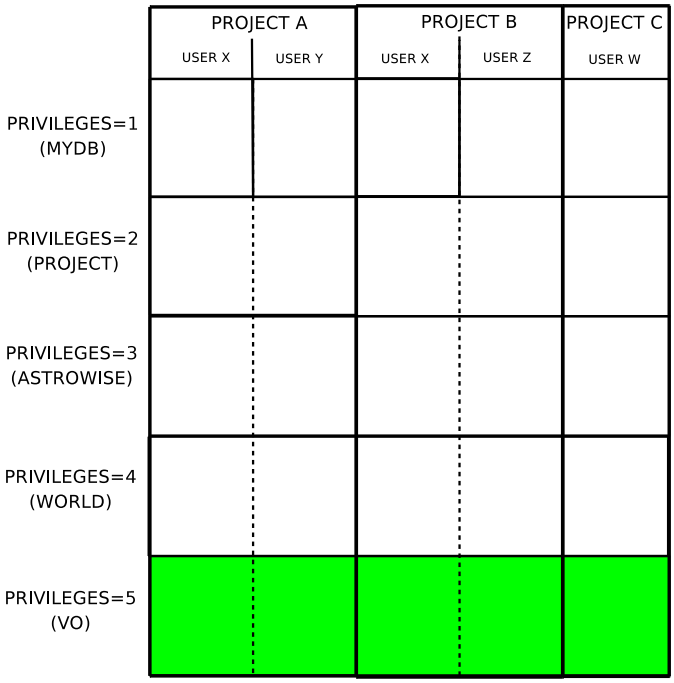
## 8.5.3 Publishing of data objects

The philosophy of `Astro-WISE` is to share results across projects whenever beneficial. An example are shared calibration data. With time calibration scientists of projects will improve their knowledge and methods how to make the best calibration data for that instrument. For example, the improvements could be based on long term trend analysis. In this sense not individual nights but complete instruments become calibrated in `Astro-WISE` . The calibration scientists can share their improved calibration data with the Astro-WISE community by publishing his results to a privileges level of 3 or higher. Astro-WISE users, also in other projects, can then re-process their data using these improved calibration data. Furthermore, to improve their own calibrations of other periods they can inspect how the calibration was derived by the calibration scientist because they can access the data lineage of a data object. In conclusion, the idea is that with time data objects are promoted to higher privileges levels. This promotion to a higher privileges level is called publishing in `Astro-WISE` .
Only the `creators` and `project` managers can publish data objects of a `project`. Table 8.8 shows who can publish data objects to which privileges levels in public and private `projects`. So in private projects the project manager is solely responsible for the `project` data which is accessible to people outside the project team.

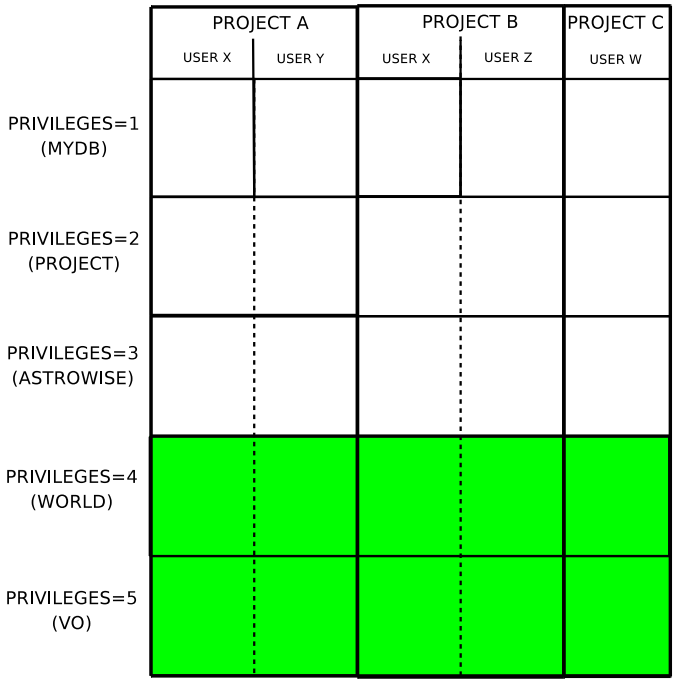
Publishing up to and including `privileges` level 3 is recursive: all dependencies are promoted to the published level as well.

Here is an example how to publish an object to `privileges` level 3:

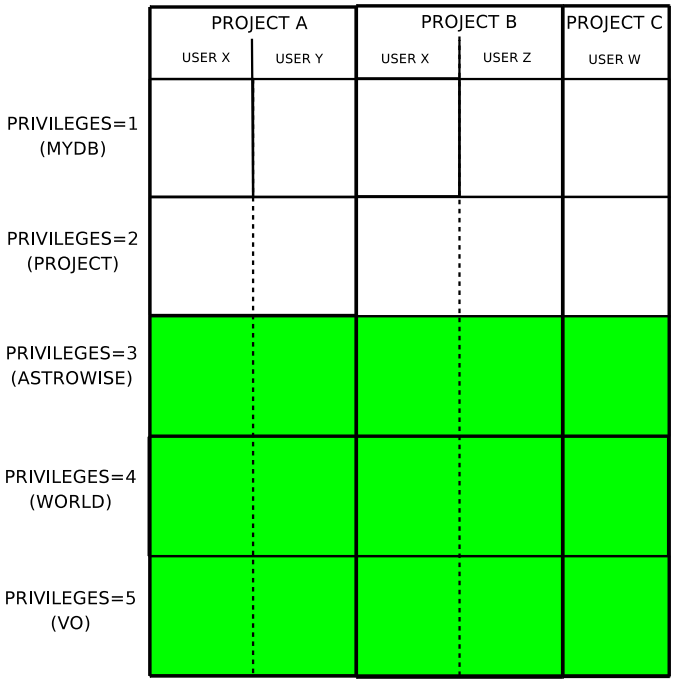
**Context settings:**
**Astro-Wise user == User X**
**Project == Project A**
**Minimum Privileges level == 5**



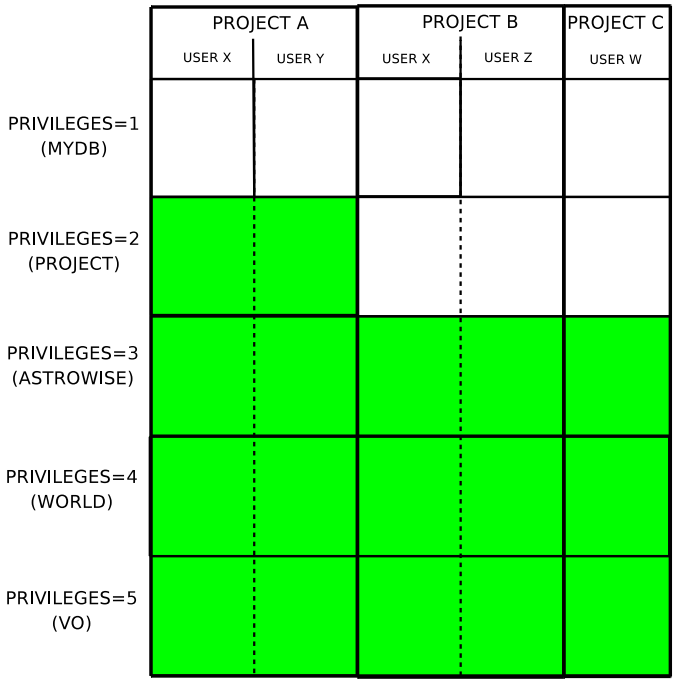

**Context settings:**
**Astro-Wise user == User X**
**Project == Project A**
**Minimum Privileges level == 4**



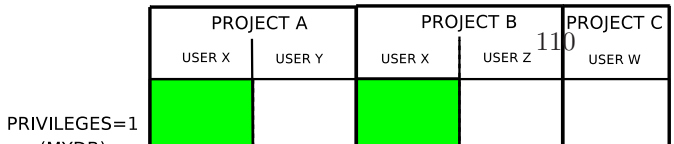

**Context settings:**
**Astro-Wise user == User X**
**Project == Project A**
**Minimum Privileges level == 3**




**Context settings:**
**Astro-Wise user == User X**
**Project == Project A**
**Minimum Privileges level == 2**




**Context settings:**
**Astro-Wise user == User X**
**Project == Project A**
**Minimum Privileges level == 1**



```
awe> context.publish(object, privileges=3, commit=True, verbose=True)
```

The result of publishing is a larger number of people (`Astro-WISE users` and possibly persons without an `Astro-WISE` account) who can view and access the data as listed in Table 8.6

#### Publishing of an object with dependencies in different `projects`

It is possible that an object has dependencies in different projects. If the object has `privileges` level $\geq 2$ it implies that the dependencies which belong to other `projects` have `privileges` level $\geq 3$. If the object has `privileges` level $= 1$ it can be the case that a dependency which belongs to another `project` has `privileges` level$= 1$. Publishing will fail in the latter case. The only way to publish MyDB data which has dependencies in other `projects` is to publish the dependencies first to `privileges` level 3 (or higher). This way the dependencies will always be visible to the parent object, independent of the `project` of the current `Context`. Now the parent object can be published to `privileges` level 2 (or higher).

#### Unpublishing and invalidation

It is possible to demote the `privileges` level of a data object down to privileges level 2. This is called unpublishing. For example, unpublishing an object to `privileges` level 2 is done via:

```
awe> context.publish(object, privileges=2, commit=True, verbose=True)
```

Objects can be depublished to `privileges` level 2 if they are not referenced by data objects at `privileges` level 3 or higher. If they are, it might be desirable to invalidate the object. One can inspect whether a data object `thisobject` is valid as follows:

```
awe> thisobject.is_valid
1
```

1 means valid, 0 means invalid. To invalidate `thisobject` use:

```
awe> context.update_is_valid(thisobject,0)
```

### 8.5.4 Deletion

Objects can be deleted from the database under the following restrictions. Every user can delete the data he created at the `MYDB` privileges level 1. For higher privileges levels only the `project` manager is allowed to delete. To delete a data object `myobject` from the database use:

```
awe> context.delete(myobject)
```

Objects can be deleted if they are not referenced by other data objects. If they are, it might be desirable to invalidate the object. One can inspect whether a data object `thisobject` is valid as follows:

```
awe> thisobject.is_valid
1
```

1 means valid, 0 means invalid. To invalidate `thisobject` use:

```
awe> context.update_is_valid(thisobject,0)
```

# Chapter 9

# AWE Tutorials

## 9.1 Tutorial Introduction

This tutorial will give you a guided introduction to the Astro-WISE environment in about 3 hours. The tutorial starts from the level of a completely new user who has never logged into the Astro-WISE environment but has an Astro-WISE account. This tutorial consists of exercises to be done primarily from the `awe`-prompt. They include the following topics:

- the `awe`-prompt basics,
- basic calibration tasks,
- astrometry,
- photometry,
- working with sourcelists,
- data mining,
- galaxy surface brightness analysis,
- Astro-WISE and Virtual Observatory interoperability,
- links to further documentation.

Topics sometimes depend on the results derived in earlier topics.

The tutorial is by no means comprehensive or exhaustive. The exercises are very basic and many topics are not discussed at all. To get a more in-depth and comprehensive presentation of the Astro-WISE environment we refer to the HOWTOs and the Manual. To get acquainted with the Astro-WISE services: please take the Guided Tour under "Try Astro-WISE" on the home page.

## 9.2 Astro-WISE basics

### 9.2.1 Setting up your environment

In order to start up the command-line interface, it is necessary to modify your shell environment so the AWE software can be found and the system can log you in to the database. The former is done using the *module* framework, the latter by creating a configuration file for AWE:

1. **Edit your shell configuration file**

   On the network of the Kapteyn Astronomical Institute:

   If you're using "csh" or "bash" add the following line to the file ∼/**.cshrc** or ∼/**.bashrc**, respectively:

   ```
   module load awe
   ```

2. **Obtain a database account and setup your ∼/.awe/Environment.cfg file**

   It is necessary to obtain a database username and password in order to login to and write ("commit") to the database. Please ask your local Astro-WISE DBA for an account.

   In your home directory make a directory `.awe` (starting with a dot) and in this subdirectory make a file called "Environment.cfg", readable only to yourself:

   ```
   ...]$ cd
   ...]$ mkdir .awe
   ...]$ cd .awe
   ...]$ touch Environment.cfg
   ...]$ chmod a-rwx,u+rw Environment.cfg
   ```

   Then add the following lines to that file replacing your username (e.g., awjkennedy) and password (e.g., IwonIn1960) in the appropriate places. The line `project` ensures the awe-prompt always starts in context ALL (what context is will be explained later in the tutorial). The line `privileges : 1` ensures that it starts within the private space of that context ('MyDB').

   ```
   [global]
   database_user : awjkennedy
   database_password : IwonIn1960
   project : ALL
   privileges : 1
   ```

   Start the awe-prompt from your shell:

   ```
   ...]$ awe
   ```

   Check that your username and default project are used. In the above case that means the welcome message ends with a message like this:

   ```
   Current profile:
   - username : awjkennedy
   - database : db.astro.rug.astro-wise.org
   - project  : ALL
   - current privileges : 1 (MyDB)

   awe>
   ```

you now have access to the database. Quit awe by hitting `Ctrl-d`.

### 9.2.2 At the awe-prompt: Looking Around

1. Start the `awe`-prompt by typing `awe` at the command line. The `awe`-prompt is the standard Python interpreter plus a few additions. The Python interpreter has built-in functions and classes:

   ```
   awe> dir(__builtins__)
   ```

   If you are not familiar with the Python language and/or object oriented programming it is a worthwile to browse this Python tutorial. It will make the rest of the tutorial much easier to grasp. Many object oriented programming concepts (especially the chapter on classes) are crucial concepts at the `awe`-prompt. The built-in functions of Python contain basic functions such as "range", "len", "int", "float", "dir" and "help" that can be useful at the `awe`-prompt. To illustrate this, the following example calculates element 19 of the Fibonacci sequence using Python syntax:

   ```
   awe> int(round( ( (1.0+5.0**0.5)**19 - (1.0-5.0**0.5)**19 ) / (2.0**19*5.0**0.5) ))

   4181
   ```

2. Conventional subjects in astronomy are represented at the `awe`-prompt by (Python) classes. These may be images (e.g., masterflats, raw science images, coadded frames) and catalogs, or instruments, chips, astrometric solutions, photometric solutions etc. Conventional metadata such as found in FITS headers are associated with these classes as *properties*. Examples are exposure time, observation date, as well as links to other objects that were used to create the object. *Methods* are functions associated with the class. For example, there are methods named `commit`, `make`, `inspect` and `do_statistics`. The most commonly used classes are automatically loaded when starting the `AWE` environment and their names are listed in the main "namespace". You can print the main namespace by entering:

   ```
   awe> dir()
   ```

3. Start the built-in help system in a similar way:

   ```
   awe> help()

   Welcome to Python 2.7!  This is the online help utility.

   If this is your first time using Python, you should definitely check out
   the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.
   .
   .
   help>
   ```

   Follow the instructions to find help on pretty much anything by typing the name of it. Use quit or Ctrl-d to exit. It is usually convenient to get help on a specific module, function, or even object. To see what dir() does exactly, for instance, type:

```
awe> help(dir)
# the screen clears and shows the following:
Help on built-in function dir in module __builtin__:

dir(...)
    dir([object]) -> list of strings

    If called without an argument, return the names in the current scope.
.
.
```

4. One of the classes listed is called Chip. This is a representation of a CCD. It has its own namespace which can be printed by typing:

```
awe> dir(Chip)
['__class__', '__del__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__', '__hash__', '__init__', '__metaclass__',
'__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'_delete', '_inverses', '_publish', 'as_dict', 'commit',
'copy_persistent_properties', 'database', 'get_creator',
'get_inverse_properties', 'get_persistent', 'get_project', 'info',
'inverse_objects', 'inverse_query', 'name', 'object_id', 'persists',
'pickle_id', 'pixelsize', 'recommit', 'select_all', 'update_header']
```

5. Another class is called RawTwilightFlatFrame. It is a representation of a raw sky flat. It also has its own namespace:

```
awe> dir(RawTwilightFlatFrame)
['DATE', 'DATE_OBS', 'EXPTIME', 'LST', 'MEAN_HIGH', 'MEAN_LOW', 'MJD_OBS',
'NAXIS1', 'NAXIS2', 'OBJECT', 'OBSERVER', 'OVSCX', 'OVSCXPRE', 'OVSCXPST',
'OVSCY', 'OVSCYPRE', 'OVSCYPST', 'PROCESS_TIME', 'PRSCX', 'PRSCXPRE',
'PRSCXPST', 'PRSCY', 'PRSCYPRE', 'PRSCYPST', 'STATUS_COMPARE',
'STATUS_INSPECT', 'STATUS_MAKE', 'STATUS_VERIFY', 'UTC', '_IS_ABSTRACT',
'_IS_CAL', '_IS_CONFIG', '_IS_RAW', '_IS_SCIENCE', '_IS_SEQ',
'_IS_SUPPORT',
.
.
```

   The namespace lists both *properties* (e.g., observation date DATE_OBS, exposure time EXPTIME) and *methods* (e.g., inspect, display) of the class.

6. Telescope calibration files such as sky flat-fields have important properties that are stored in the Astro-WISE database. These are called persistent properties and they can be listed as follows:

```
awe> RawTwilightFlatFrame.get_persistent_properties()
['DATE', 'DATE_OBS', 'EXPTIME', 'LST', 'MJD_OBS', 'NAXIS1', 'NAXIS2',
'OBJECT', 'OBSERVER', 'OVSCX', 'OVSCXPRE', 'OVSCXPST', 'OVSCY',
'OVSCYPRE', 'OVSCYPST', 'PRSCX', 'PRSCXPRE', 'PRSCXPST', 'PRSCY',
'PRSCYPRE', 'PRSCYPST', 'UTC', 'chip', 'creation_date', 'extension',
```

```
'filename', 'filter', 'globalname', 'imstat', 'instrument', 'is_valid',
'object_id', 'observing_block', 'overscan_x_stat', 'overscan_y_stat',
'prescan_x_stat', 'prescan_y_stat', 'process_params', 'process_status',
'quality_flags', 'raw_fits_data', 'template']
```

In this list "chip" points to another class: the class Chip we just described. To list its persistent properties type:

```
awe> RawTwilightFlatFrame.chip.get_persistent_properties()
['name', 'object_id']

or

awe> Chip.get_persistent_properties()
['name', 'object_id']
```

7. You can find out what each property means by entering:

```
awe> help(RawTwilightFlatFrame)
Help on class RawTwilightFlatFrame in module astro.main.RawFrame:

class RawTwilightFlatFrame(RawFrame)
 |  Method resolution order:
 |      RawTwilightFlatFrame
 |      RawFrame
 |      astro.main.BaseFrame.BaseFrame
 |      common.database.DataObject.DataObject
 |      common.database.DBMain.DBObject
 |      astro.main.ProcessTarget.ProcessTarget
 |      common.database.DBMeta.DBMixin
 |      common.main.ProcessTarget.ProcessTarget
 |      astro.main.OnTheFly.OnTheFly
 |      common.main.OnTheFly.OnTheFly
 |      __builtin__.object
 |
...
...
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  DATE
 |      UTC date the original data file was saved [None]
 |
 |  DATE_OBS
 |      UTC date at the start of the observation [None]
 |
 |  EXPTIME
 |      Total observation time [sec]
.....


Hit "q" to exit the help page.
```

Here the difference between DATE and DATE_OBS becomes clear. The help tells you much more than just the properties. The first line tells you that the code for this class is stored in module astro.main.RawFrame (how to find the code will be explained later). It is followed by a definition of RawTwilightFlatFrames. The explanation of the other lines is beyond the scope of this tutorial.

8. You can use the persistent properties to query the database for a particular RawTwilightFrame. Lets find all OmegaCAM RawTwilightFlatFrames which were observed in August 2011 (the start of science observations with OmegaCAM):

```
NB: enter the following 2 lines on 1 line of the awe prompt:

awe> query = (RawTwilightFlatFrame.instrument.name == 'OMEGACAM') &
(RawTwilightFlatFrame.DATE_OBS < datetime.datetime(2011,9,1))

NB: you can enter the following command on 2 input lines by continuing
    the line with '\':

awe> query = (RawTwilightFlatFrame.instrument.name == 'OMEGACAM') & \
             (RawTwilightFlatFrame.DATE_OBS < datetime.datetime(2011,9,1))

or by using an extra set of parentheses:

awe> query = ((RawTwilightFlatFrame.instrument.name == 'OMEGACAM') &
(RawTwilightFlatFrame.DATE_OBS < datetime.datetime(2011,9,1)))

awe> len(query)
6240
```

You can select one of these RawTwilightFrames using an index and see when it was observed exactly:

```
awe> raw = query[10]
awe> raw.DATE_OBS
datetime.datetime(2011, 8, 10, 22, 49, 22)
```

or more directly:

```
awe> query[10].DATE_OBS
datetime.datetime(2011, 8, 10, 22, 49, 22)
```

and retrieve the image from the dataserver (download to your current working directory):

```
awe> raw.retrieve()
```

once retrieved, you can display it:

```
awe> raw.display()
```

or inspect it:

```
awe> raw.inspect()
```

The last three commands are method calls. Methods typically perform an operation on a (Python) object. The object "raw" is called an instantation of the class RawTwilight-FlatFrame. The last command brings up a graphical window: enter 'q' when you hover over it to quit the graphical window.

9. One other very useful method on instances of database objects in AWE, is `info()`. This method prints out all the persistent attributes and their values:

```
awe> chip = Chip()
awe> chip.info()
Chip: <astro.main.Chip.Chip object at 0xb03cd0ec>
 |
 +-name:
 +-object_id: '0000000000000000000000000000000000
```

The Chip class has only one useful persistent attribute: `name`. `object_id` is a unique object identifier identifying each object in the database. Try to run the `info` method on all the other objects in these tutorials to get their information. Be aware, however, that not all objects have this method.

Notes:

- For this tutorial it can be helpful to use "Tab" key a lot for automatic completion of names. Tab completion does not work on queries of objects (queries are explained in next section).
- A command-line history is saved. Using the "Up", "Down" or "Ctrl-p", "Ctrl-n" keys you can recall previous commands. If you specify a partial command the history will be searched for matches which start with the same characters.
- Make sure you use the class when using `get_persistent_properties()` (i.e., `RawTwilightFlatFrame.get_persistent_properties()`) and not an instance of the class (i.e., `raw.get_persistent_properties()`).

### 9.2.3 The power of querying

1. **Which instruments are stored in the database?**

This can be done in several ways. One way is to query for all instruments of which the name is not equal to an empty string. Alternatively you could use the "like" functionality to ask for all Instruments for which the name is equal to any string.

```
awe> q = Instrument.name != ''
awe> for i in q: print i.name

awe> q = Instrument.name.like('*')
awe> for i in q: print i.name
```

Reminder: `Instrument.get_persistent_properties()` shows you that `Instrument` has the property `name` stored in the Astro-WISE database. Entering:

```
awe> Instrument.n
```

and hitting the Tab key also reveals that `Instrument` has the property `name`. NB: Tab completion does not work on queries (i.e., `q.n` following above example).

2. **What are the names of the chips (CCDs) present in the OmegaCAM camera?**
   You can ask this of the `context` object, instantiated by default at the `AWE` prompt:

   ```
   awe> context.get_chips_for_instrument('OMEGACAM')
   ```

3. **How many RawScienceFrames for OmegaCAM are stored in the** Astro-WISE **database?**

   ```
   awe> q = (RawScienceFrame.instrument.name == 'OMEGACAM')
   awe> len(q)
   3213677
   ```

   ...and counting. Please note that for each OmegaCAM exposure 32 (one for each CCD) RawScienceFrames are created in the database.

4. **Execute these lines at the prompt to select a particular ReducedScienceFrame (a flat-fielded and de-biased science image, for a single CCD) from our database.**

   ```
   awe> sci = ((ReducedScienceFrame.DATE_OBS == datetime.datetime(2011, 8, 6, 8, 5, 20)) & \
              (ReducedScienceFrame.chip.name == 'ESO_CCD_#65')).min('creation_date')
   ```

   For this ReducedScienceFrame, retrieve from the Astro-WISE dataserver the image itself and the bias image that was used. The FITS images will be saved to the current working directory.

   ```
   awe> sci.retrieve()
   awe> sci.bias.retrieve()
   ```

   The images, such as ReducedScienceFrames, which you and others create in Astro-WISE are stored on the Astro-WISE dataservers. As explained in more detail later, this will only be done if you explicitly *commit* these images to Astro-WISE. Even after committing them, you can invalidate images at any time if you decide they are no good after all.

5. **For the ReducedScienceFrame of the previous exercise print the observing date, the object, instrument, filter and chip (CCD) name.**

   ```
   (NB: enter following 2 lines as one line on the awe prompt.)
   awe> print sci.DATE_OBS, sci.OBJECT, sci.instrument.name, sci.filter.name,
   sci.chip.name
   # Or
   awe> print sci.DATE_OBS
   awe> print sci.OBJECT
   # etc.
   ```

6. **For the ReducedScienceFrame selected, print the observation date DATE_OBS of the RawBiasFrames used in the creation of the master bias that was used in debiasing the ReducedScienceFrame.**

```
awe> for frame in sci.bias.raw_bias_frames: print frame.DATE_OBS
```

And to print the equivalent modified Julian date:

```
awe> for frame in sci.bias.raw_bias_frames: print frame.MJD_OBS
```

```
or by calculating from the DATE_OBS:
```

```
awe> from common.util.utilities import datetime_to_mjd
awe> for frame in sci.bias.raw_bias_frames: print datetime_to_mjd(frame.DATE_OBS)
```

The "from ... import ..." statement loads a method to do the conversion. To find out what can be imported from a module, simply import the module and use the built-in dir() function:

```
awe> from common.util import utilities
awe> dir(utilities)
```

To get help on the entire module, use the built-in help() function:

```
awe> help(utilities)
```

7. **What were the exposure levels of the raw dome flats that were used?**

   Note that the flat-field used in the reduction is a MasterFlatFrame which is this case was created from a master dome (DomeFlatFrame).

```
(NB: make sure to indent the 2nd line below:)
awe> for frame in sci.flat.domeflat.raw_domeflat_frames: print frame.imstat.median
```

### 9.2.4 More Advanced Queries

1. **Find out how many RawBiasFrames there are for the OmegaCAM instrument where the bias level is greater than 1000 ADU.**

   The property of the RawBiasFrame that contains its image statistics is called "imstat". This is an instance of the "Imstat" class, which itself has properties such as "mean", "median" and "stdev". We need to combine this query with a query on the name of the instrument of the RawBiasFrame.

```
NB: print following lines on a single awe prompt line.
awe> q = (RawBiasFrame.instrument.name == 'OMEGACAM') & \
         (RawBiasFrame.imstat.median > 1000.0)
awe> len(q)
```

2. **Find out how many RawScienceFrames observed the first week of November 2011 are present in the database for the OmegaCAM instrument. For these RawScienceFrames, print the observation date, the filter, the R.A. and Dec, and the OBJECT header keyword.**

   Here a lot of things come together. Usually, frame attributes that are all upper-case letters correspond to a FITS header keyword of the same name. This will apply to only some of our attributes. We will need to know the names of the other attributes that we are interested in, and if they can be queried on at all. This is done as follows:

```
awe> RawScienceFrame.get_persistent_properties()
```

In particular note the DATE_OBS, "astrom" and OBJECT properties. The R.A. and Dec can be found in "astrom" (as CRVAL1 and CRVAL2, respectively). In addition, we are directly querying on a datetime object (DATE_OBS) in order to get the data for 2011. This requires that you create a datetime object for your dates, so you can compare the two.

It may be helpful to limit the query to a single CCD, in order to avoid getting 32 times the same information (once for each CCD of the OmegaCAM camera).

```
awe> dat1 = datetime.datetime(2011,11,1)
awe> dat2 = datetime.datetime(2011,12,1)
awe> q = (RawScienceFrame.DATE_OBS>dat1) & \
         (RawScienceFrame.DATE_OBS<dat2) & \
         (RawScienceFrame.chip.name=='ESO_CCD_#65') & \
         (RawScienceFrame.instrument.name=='OMEGACAM')
awe> for s in q: print s.DATE_OBS, s.filter.name, s.OBJECT, s.astrom.CRVAL1,
s.astrom.CRVAL2
```

### 9.2.5 System Calls from the awe-prompt

1. It is possible to do Unix/Linux system calls from the awe-prompt. The "os" module is imported by default and can be used to run commands as follows:

```
awe> os.system('ls')
awe> os.system('pwd')

awe> command = 'skycat file.fits&'
awe> os.system(command)
```

To move one directory up use:

```
awe> os.chdir('..')
```

### 9.2.6 Understanding Python errors/exceptions/backtrace

1. If you type something erroneous at the awe-prompt an error message will be returned. The last sentence of the message is almost always the most useful line to determine the cause of the error.

```
awe> RawScienceFrame.instrument.ThisAttributeDoesNotExist=='bla'
Traceback (most recent call last): File "<stdin>", line 1, in <module>
File
"/net/smyth/data/users/helmich/aweTEST/common/database/DBProperties.py",
line 200, in __getattr__ persistent_property.__getattr__(self, attr) File
"/net/smyth/data/users/helmich/aweTEST/common/database/DBProperties.py",
line 126, in __getattr__ raise AttributeError, 'Persistent attribute "%s"
of class "%s" does not have a persistent attribute "%s"' %
(self.attribute, self.cls.__name__, attr) AttributeError: Persistent
attribute "instrument" of class "RawScienceFrame" does not have a
persistent attribute "ThisAttributeDoesNotExist"
```

## 9.3 Calibrating data

In this tutorial chapter you will process data for the first time. We take the example of reducing raw science frames. This involves trimming, de-biasing, flat-fielding, illumination-correction, as well as the detection of saturated pixels, cosmic-rays and satellite tracks. Together with cold- and hot pixels these affected pixels are assigned a weight of zero in the weight map, which is also created. The results will be stored on the Astro-WISE dataservers and database. We assume that for now you want to store those such that only you can see these results. You can decide later to make the results accessible to other people.

### 9.3.1 Database projects and privileges

1. Data in the database is organized in projects. To see in which project you are currently working:

   ```
   awe> print context.get_current_project()
   ```

   For example the output could be:

   ```
      Project:     OMEGACAM@VST  (id = 36)
      Description: All public data for OmegaCAM.  OmegaCAM is a 1 square
   degree wide field, optical, 16k X 16k pixel camera for the VLT Survey
   Telescope (VST) on Paranal Observatory.
      Instrument:  OMEGACAM
      Maximum Privileges: 4 (World)
      Current Privileges: 1 (MyDB)
   ```

   If you are doing this tutorial as part of a course you have been asked to produce the data in a certain project: `TUTORIAL2014`. You can change to this project by typing:

   ```
   awe> context.set_project('TUTORIAL2014')
   ```

   If you have not been asked to enter a certain project you can use the project you are in now. From now on, everything you create will be stored inside the chosen project. There will be other members of this project. You can set privileges to restrict who can view and access data within a project. A privilege of 1 means only you can view and access the data:

   ```
   awe> context.set_privileges(1)
   ```

   From here on, everything you create will be stored at privilege level 1 in the chosen project: only you can view/access the data. To allow other people in the project to see products, set privileges to 2 before you create them.

### 9.3.2 Processing science frames

1. **Selecting and exploring** `RawScienceFrames` Enter the following query that selects the raw science data objects, i.e., instantiations of the class `RawScienceFrame`, from the database.

```
awe> query = \
     (RawScienceFrame.instrument.name == 'OMEGACAM') & \
     (RawScienceFrame.template.start == datetime.datetime(2012, 6, 1, 8, 51, 44)) & \
     (RawScienceFrame.is_valid > 0)
```

To see how many raw science frames are found type:

```
awe> len(query)
160
```

These are 5 OmegaCAM exposures, each consisting of 32 CCDs (32 RawScienceFrames are created for a single exposure).

We select only the data of ccd#65 for now:

```
awe> query = query & (RawScienceFrame.chip.name == 'ESO_CCD_#65')
awe> len(query)
5
```

To obtain general meta-data of the last RawScienceFrame type:

```
awe> raw = query.max('DATE_OBS')
awe> raw.info()
```

or only the observation dates, filter names, and maximum pixel value in the frame for all of them:

```
awe> for raw in query: print raw.DATE_OBS, raw.filter.name, raw.imstat.max
```

Download the FITS file:

```
awe> raw.retrieve()
```

Now that you have the file locally you can display it using your favorite image viewer.

To visually inspect a frame with the help of a few analysis tools type:

```
awe> raw.inspect()
```

This brings up a graphical window. Hover your mouse over an object in the window and hit 'w': this shows a 3D wire-frame plot of the pixel values around the mouse pointer. Hit 'q' when you hover over it to close the graphical window. For a listing of all hot-keys type:

```
awe> help(raw.inspect)
```

2. **Reduce the `RawScienceFrames` that you found above**.

   *See the calibration* HOW-TO *(section 10) for an overview of how to process data in Astro-WISE.*

   You will use standard recipes to reduce the `RawScienceFrames`. It is straightforward to adapt these or to write your own recipes, but this is beyond the scope of this tutorial.

   In the Astro-WISE environment you may choose the compute cluster (the Distributed Processing Unit or DPU) where the data reduction processes run. In that case no results are stored on your local machine. Created data (FITS files) are stored on the dataserver while meta-data is committed to the database.

   To de-bias and flatfield the above raw science data using the DPU you can use the "Reduce" recipe:

```
awe> filenames = [raw.filename for raw in query]
awe> dpu.run('Reduce', instrument='OMEGACAM', raw_filenames=filenames,
             commit=True)
```

The commit=True switch ensures that your data is committed. Choosing commit=False will perform a "dry run": everything will be done except committing your results at the end. Alternatively, you can do the query and reduction at once via:

```
awe> dpu.run('Reduce', instrument='OMEGACAM', template='2012-06-01T08:51:44',
             chip='ESO_CCD_#65')
```

You can view the status of your DPU job via

```
awe> dpu.get_status()
```

or by browsing the DPU server's webpage. See the links found on the: Processing Grid page page (http://www.astro-wise.org/portal/aw_prompt.shtml).

You can see how the processing went by retrieving the processing logs from the DPU.

```
awe> dpu.get_logs()
```

This command will not return anything until the process has been completed. The returned lines are also written to a single ("date+time".log) file in your local directory per awe-prompt session.

To cancel DPU processing jobs:

```
awe> for jobid in dpu.get_jobids(): dpu.cancel_job(jobid)
```

Although it has advantages to do the processing of `RawScienceFrames` in parallel on a DPU it can sometimes be convenient to use your local machine. In this case the results are left in your current working directory. To de-bias and flatfield your raw science data found above using using local CPU you can use the same Reduce recipe as follows:

```
awe> task = ReduceTask(instrument='OMEGACAM', raw_filenames=filenames,
                       commit=True)
awe> task.execute()
or...
awe> task = ReduceTask(instrument='OMEGACAM', template='2012-06-01T08:51:44',
                       chip='ESO_CCD_#65', commit=True)
awe> task.execute()
```

### 9.3.3 Inspect the results: ReducedScienceFrame

1. **Locating the results.** The results of above the process are de-biased, flatfielded science frames, which is a class of objects called `ReducedScienceFrame` in Astro-WISE. To select one of the ones you just created type:

```
awe> qred = (ReducedScienceFrame.raw==query[0])
awe> qred.project_only()
awe> qred.user_only()
awe> red = qred.max('creation_date')
```

The first command queries for all ReducedScienceFrames which were created from the RawScienceFrame object query[0] in all projects you have access to. The second command narrows the selected ReducedScienceFrames to only those created within the project you are currently in. The third command zooms in on the subset of those that were created by you. The last command selects the ReducedScienceFrame that you created most recently. To do all this in a single command for all RawScienceFrames in query:

```
awe> qred=[(ReducedScienceFrame.raw==raw).project_only().user_only().max('creation_date') for r
```

2. **Inspecting the results.** This can be done in similar fashion as you did for the RawScienceFrames above. For example:

   ```
   awe> qred[0].inspect()
   ```

3. **Determine which calibration frames were used.** As you did not create the bias frames and flatfields, the Astro-WISE environment selected those for you. To get general information on which ones were used:

   ```
   awe> qred[0].bias.info()
   awe> qred[0].flat.info()
   ```

   and to inspect one of the RawBiasFrames that were used to create the bias frame:

   ```
   awe> red=qred[0] #To allow Tab completion in next command....
   awe> red.bias.raw_bias_frames[0].inspect()
   ```

4. **We have not configured anything beyond specifying the input frames in the previous examples. Find out which other parameters can be configured.**

   *See the configuration* HOW-TO *(section 8.4) for more details*

   Calling the `help()` function on the Task that you are going to run gives a description of the arguments to the Task. The arguments are mostly query parameters which determine which input frames (RawScienceFrames in this case) are going to be reduced. In addition an argument "pars" can be specified which contains the processing parameters.

   ```
   awe> help(ReduceTask)
   ```

   Instantiate the *Pars* class with as argument the task, the pipeline identifier used in the dpu call or the class:

   ```
   awe> p = Pars(ReduceTask, instrument='OMEGACAM')
   awe> p = Pars('Reduce', instrument='OMEGACAM')
   awe> p = Pars(ReducedScienceFrame, instrument='OMEGACAM')
   ```

   By setting the instrument argument, instrument specific default processing parameters are used. Now call the `show()` method:

   ```
   awe> p.show()
   ```

   A parameter can be set as follows (use Tab completion):

```
awe> p.ReducedScienceFrame.process_params.FRINGE_THRESHOLD_HIGH = 6.0
```

Call the `get()` method to obtain the dictionary that you can supply to the task:

```
awe> p.get()
{'ReducedScienceFrame.process_params.FRINGE_THRESHOLD_HIGH': 6.0}
```

## 9.4 Astrometric calibration

In this tutorial you will astrometrically calibrate the ReducedScienceFrames (i.e., de-biased and flatfielded science frames) from the previous tutorial.

### 9.4.1 Find ReducedScienceFrames to run astrometry on

1. In a previous exercise, you reduced some RawScienceFrames, resulting in the creation of ReducedScienceFrames. **Construct a list of these frames.**

   Instruction: Use information in the Python database querying HOW-TO at the Astro-WISE portal to construct a query that contains your and only your ReducedScienceFrames.

```
Answer (1):
awe> t = '2012-06-01T08:51:44'
awe> q = ReducedScienceFrame.template.start == dateutil.parser.parse(t)
awe> frames = q.project_only().user_only()
awe> len(frames)
<some number of frames>

Answer (2):
# Alternatively, you can look for ReducedScienceFrames that you made less
# than 20 minutes ago:

# Use current UTC minus datetime.timedelta(days, seconds, microseconds).
awe> date = datetime.datetime.utcnow() - datetime.timedelta(0, 20*60, 0)
awe> q = (ReducedScienceFrame.creation_date > date).user_only()
```

### 9.4.2 Derive astrometric calibration

1. **Derive astrometry** by creating AstrometricParameters objects for the ReducedScienceFrames.

   Instruction: Use information in the Astrometry check HOW-TO at the Astro-WISE portal to re-derive the AstrometricParameters for your ReducedScienceFrames.

```
Answer:
awe> filenames = [f.filename for f in frames]
awe> dpu.run('Astrometry', instrument='OMEGACAM', red_filenames=filenames, C=1)
```

   or (note that this command selects the most recent, valid ReducedScienceFrames for the specified template and chip, which may be frames made by someone other than yourself)

```
awe> dpu.run('Astrometry', instrument='OMEGACAM', template='2012-06-01T08:51:44',
             chip='ESO_CCD_#65', C=1)
```

2. **Examine the attributes** of an AstrometricParameters object.

   Instruction: Use the dir() method to see the attributes of interest (usually in all capital letters).

```
Answer:
awe> ap = (AstrometricParameters.reduced == frames[0])[0]
awe> dir(ap)
```

```
['CD1_1', 'CD1_2', 'CD2_1', 'CD2_2', 'CRPIX1', 'CRPIX2', 'CRVAL1', 'CRVAL2',
'CTYPE1', 'CTYPE2', 'FITERRS', 'FITPARMS', 'Flagged', 'MEAN_DDEC', 'MEAN_DRA',
'NFITPARM', 'NREF', . . .
awe> ap.NREF
...
awe> ap.RMS
...
awe> ap.SEEING
...
```

### 9.4.3 Visually inspect astrometry

1. **Display the quality control plot for the AstrometricParameters object.**

   Instruction: Follow the instructions in the Astrometry Quality Control HOW-TO at the Astro-WISE portal to inspect the astrometric solution for one of your ReducedScience-Frames.

   Answer:

   ```
   awe> ap.inspect()
   ```

## 9.5 Photometric Pipeline

This section gives a quick summary of how photometric calibration can be done in Astro-WISE.

### 9.5.1 Deriving zeropoint and extinction

Photometric calibration in Astro-WISE is done by comparing the counts in standard stars on images to their magnitudes as listed in a reference catalog of standard stars. The final result is a *PhotometricParameters* object which stores the zeropoint and extinction. This object can then be used to photometrically calibrate your ReducedScienceFrames.

Lets again consider the OmegaCAM data used in previous exercises. It was observed the night of 31 May 2012, using the r filter. A Landolt standard star field, SA113, was observed that night. ReducedScienceFrames already exist in the database for that standard field exposure:

1. ```
   awe> q = ReducedScienceFrame.template.start == dateutil.parser.parse('2012-06-01T09:38:56')
   awe> len(q)
   32
   ```

   Lets again only look at CCD#65:

   ```
   awe> q = q & (ReducedScienceFrame.chip.name == 'ESO_CCD_#65')
   ```

2. Create a Photometric Source Catalog, a *PhotSrcCatalog* object, which crossmatches standard stars listed in a standard star catalog, known as a Photometric Reference Catalog, (a *PhotRefCatalog* object) with stars on the ReducedScienceFrame observation. An example command from the `awe`-prompt using a standard recipe is:

   ```
   awe> filenames = [red.filename for red in q]
   awe> task = PhotcatTask(instrument='OMEGACAM', red_filenames=filenames,
        ... transform=1, inspect=1, commit=1)
   awe> task.execute()
   ```

   `red_filenames` is a list of the filenames of the ReducedScienceFrames of the standard star observations. `transform=1` ensures that differences between the passbands of the instrument and the standard photometric system are accounted for. `inspect=1` means a plot will be created to inspect the resulting crossmatch (default `inspect=0`). `commit=1` ensures the result is saved in the database (default `commit=0`). The above command uses default settings for process parameters such as the standard stars to use and the configuration of the source extraction algorithm.

3. Create a Photometric Parameters Catalog, a PhotometricParameters object, which contains photometric parameters, such as zeropoint and exintction. An example command from the `awe`-prompt using a standard recipe is:

   ```
   awe> task = PhotomTask(instrument='OMEGACAM', red_filenames=filenames,
        ... pars={'PhotometricParameters.process_params.SIGCLIP_LEVEL': 3.5},
        ... inspect=1, commit=1)
   ```

   where `red_filenames`, `inspect` and `commit` have the same meaning as for the Photometric Source Catalog. The process parameter for sigma clipping was increased; it removed individual standard star measurements which differ by more than 3.5 standard deviations from the median from the calculation of the zeropoint. The above command uses a default atmospheric extinction correction.

   After following these steps ReducedScienceFrames which contain your scientific targets can be photometrically calibrated using the PhotometricParameters object.
   ```

### 9.5.2 Standard Star Catalog operations

1. To retrieve the standard star catalog from the database:

   ```
   awe> refcat = PhotRefCatalog.get()
   awe> refcat.retrieve()
   ```

   To explore its content, see the possibilities by typing:

   ```
   awe> dir(refcat)
   ```

   For example:

   ```
   awe> mag_dict = refcat.get_dict_of_magnitudes('SloanR')
   ```

2. To overlay the content of the catalog over the frame of a standard field which we found in above exercises, retrieve the frame:

   ```
   awe> q[0].retrieve()
   ```

   Display it in skycat:

   ```
   awe> q[0].display()
   ```

   Retrieve the standard star catalog in skycat format to local disk:

   ```
   awe> refcat.make_skycat()
   ```

   To find its filename:

   ```
   awe> os.system('ls *scat')
   ```

   Use the "File" menu in skycat to load and display the image and the "Data-servers>Local Catalogs" menu to load and overplot the standard catalog and overplot it.

## 9.6 SourceList and AssociateList Exercises

1. **For the OmegaCAM data used in earlier execises, make a SourceList each for 2 ReducedScienceFrames (they overlap). Set the SExtractor detection threshold to 6.0**

   Lets find the ReducedScienceFrames we made earlier and pick two:

   ```
   awe> reds = (ReducedScienceFrame.chip.name == 'ESO_CCD_#65').user_only().project_only()
   awe> filename1 = reds[0].filename
   awe> filename2 = reds[1].filename
   awe> p = Pars(SourceList, instrument='OMEGACAM')
   awe> p.SourceList.sexconf.DETECT_THRESH = 6.0
   awe> pars = p.get()
   awe> task = SourceListTask(filenames=[filename1], pars=pars, commit=1)
   awe> task.execute()
   awe> task = SourceListTask(filenames=[filename2], pars=pars, commit=1)
   awe> task.execute()
   ```

2. **For one of the SourceLists you just made, obtain the number of sources in the sourcelist, find the brightest source, and obtain its half-light radius.**

   The final line printed by the task should have display the SourceList identifier (SLID), which can be used to query directly (use your own SLID instead of the one below):

   ```
   awe> sl = (SourceList.SLID == 11212841)[0]
   ```

   Alternatively, select the most recent SourceList created by yourself:

   ```
   awe> sl = (SourceList.SLID > 100000).user_only().max('SLID')
   ```

   Get the information from the SourceList:

   ```
   awe> print sl.number_of_sources
   awe> magsid = [(src['MAG_ISO'], src['SID']) for src in sl.sources]
   awe> magsid.sort()
   awe> print magsid[0]
   awe> (-16.573373794555664, 230L)
   awe> sl.sources[230]['SID'], sl.sources[230]['FLUX_RADIUS']
   (230L, 6.32753324508667)
   ```

3. **Find out the piece of sky covered by the USNO catalogue in the database.**

   ```
   awe> usno = (SourceList.name == 'USNO-A2.0')[0]
   awe> RA, DEC = usno.sources.RA, usno.sources.DEC
   awe> print max(RA), min(DEC), min(RA), max(DEC)
   ```

   This is the current coverage as of 7/7/2014 of the USNO-A2.0 catalogue in the Astro-WISE database.

4. **Find the B and R mags of the sources in the USNO catalog which are inside a distance of 0.1 degrees of position 12.0$^o$ -29.0$^o$.**

```
awe> print usno.info()
awe> attrs = { 'RA': [], 'DEC': [], 'USNO_RMAG': [], 'USNO_BMAG': [] }
awe> area = (12.0, -29.0, 0.1)
awe> r = usno.sources.area_search(Area=area, attr_dict=attrs)
awe> nos = len(r[usno.SLID])
awe> for k in range(nos):
...   print attrs['USNO_BMAG'][k], attrs['USNO_RMAG'][k]
...
```

5. AssociateLists are (among others) used for the Global Astrometric Solution. We are going to inspect some of these lists in the database. These are easily found since their names all start with GAS:

   ```
   awe> task = AssociateListTask(ids=[(11212841, 's'), (11212851, 's')])
   awe> task.execute()
   ```

   As before, the final line printed by the task shows the AssociateList identifier (ALID). Get it:

   ```
   awe> al = (AssociateList.ALID == 838511)[0]
   ```

   and **confirm the sourcelists it has been made from and find out the pointing and corresponding CCD's.**

   ```
   awe> for sl in al.sourcelists:
   ...    print sl.frame.chip.name, sl.frame.astrom.CRVAL1, sl.frame.astrom.CRVAL2
   ```

6. An AssociateList contains only references to sources in the associated SourceLists or to other AssociateLists. **How to make a new SourceList from an AssociateList?** This can be done with *CombinedList*.

   Initialize a CombinedList with the selected AssociateList:

   ```
   awe> cl = CombinedList(al)
   ```

   We have to select the method which we will use to combine data. There are three of them: `combined_method=1` will include in the resulting SourceList all sources (associated by AssociateList and non-associated), `combined_method=2` will include only sources which are presented in AssociateList, and `combined_method=3` will include only sources which are presented in SourceLists (used to create AssociateList) but not in AssociateList itself. By default `combined_method=2`.

   ```
   awe> cl.set_combined_method(1)
   ```

   We have also to specify which attributes of the associated SourceLists we would like to see in the output SourceList. There are two modes: to treat attributes as a magnitude attribute (4 new attribute will be created - `MAG_1` to store average value, `MAGFLAG_1` to store flag for the magnitude, `MAGERR_1` to store rms of the value, `MAGN_1` to store a number of input values, 2 or 1 in the example) or to specify the attribute as user-defined with the user-selected aggregate function.

   For example, we want to see in the output SourceList `MAG_ISO` which is a magnitude in the OmegaCAM filter `'OCAM_r_SDSS'`.

```
awe> cl.set_user_defined_magnitudes({'MAG_ISO':'OCAM_r_SDSS'})
```

At the same time we want to see in the output SourceList FLUX_RADIUS and YM2, and we want a maximum value for the first attribute not an average.

```
awe> cl.set_user_defined_attributes(['FLUX_RADIUS','YM2'])
awe> cl.set_aggregate_functions({'FLUX_RADIUS':'MAX'})
```

Next we make and commit a new SourceList

```
awe> cl.make()
awe> cl.commit()
SourceList: Name of SourceList : SL-EHELMICH-0011213541
SourceList ID      : 11213541
Sources in list    : 541
Parameters in list : 15

 |
 +-COMBINE_METHOD: 1
 +-OBJECT:
 +-SLID: 11213541
 +-associatelist: 838511
 +-astrom_params: None
 +-chip: None
 +-creation_date: 2014-07-07 11:24:46.104790
 +-detection_frame: None
 +-filename:
 +-filter: None
 +-filters: MAG_1:OCAM_r_SDSS
 +-frame: None
 +-globalname:
 +-instrument: None
 +-is_valid: 1
 +-llDEC: -33.666714532
 +-llRA: 338.173542005
 +-lrDEC: -33.6667144939
 +-lrRA: 338.001660691
 +-name: SL-EHELMICH-0011213541
 +-number_of_sources: 541
 +-object_id: 'FD99B6D971BD7548E043C016A9C3FD7B'
 +-process_params: <astro.main.SourceList.SourceListParameters object at 0x71952c10>
 +-sexconf: <astro.main.Config.SextractorConfig object at 0x71952c90>
 +-sexparam: <class 'common.database.typed_list.typed_list'>(<type 'str'>, [])
 +-sources: {'MAGFLAG_1': <type 'long'>, 'YM2': <type 'float'>, 'MAG_1': <class 'common.util.ty
 +-ulDEC: -33.3532431068
 +-ulRA: 338.173231335
 +-urDEC: -33.3532430691
 +-urRA: 338.00197156
None
```

As we can see, new SourceList with SLID=11213541 has attributes MAG_1 (contains MAG_ISO from input SourceLists), FLUX_RADIUS (maximum FLUX_RADIUS from input SourceLists) and YM2 (average YM2 from input SourceLists) and contains 541 sources.

## 9.7 Data Mining Exercises

### 9.7.1 Investigating Twilight Behavior from RawTwilightFlatFrames

1. Find all OmegaCAM raw twilight flat frames for the chip with name `ESO_CCD_#92` and the filter with `OCAM_g_SDSS`, which have image statistics such that the median is smaller than 5000 ADU. Only select those that have their `quality_flags` set to zero. Assign the *query* to a Python variable.

   How many raw twilight flat frames satisfy the above criteria?

   ```
   awe> qinstr  = RawTwilightFlatFrame.instrument.name == 'OMEGACAM'
   awe> qfilter = RawTwilightFlatFrame.filter.name   == 'OCAM_g_SDSS'
   awe> qchip   = RawTwilightFlatFrame.chip.name     == 'ESO_CCD_#92'
   awe> qim     = RawTwilightFlatFrame.imstat.median >  1000
   awe> qqf     = RawTwilightFlatFrame.quality_flags == 0
   awe> query  = qinstr & qfilter & qchip & qim & qqf
   awe> print len(query)
   1033
   ```

2. Using the result from the previous question, assign the MJD_OBS (the modified Julian date) of each RawTwilightFlatFrame to variable `t` and the median of the image statistics divided by the EXPTIME of each RawTwilightFlatFrame to variable `y`. Make a scatter plot of `t` versus `y`.

   ```
   awe> t = [rtf.MJD_OBS for rtf in query]
   awe> y = [rtf.imstat.median/rtf.EXPTIME for rtf in query]
   or, quicker,
   awe> result = [(rtf.MJD_OBS, rtf.imstat.median/rtf.EXPTIME) for rtf in query]
   awe> t, y = zip(*result)
   awe> pylab.scatter(t, y)
   ```

3. Since there seems to be a recurring pattern in the previous plot, we'll have a closer look. Take the fractional part of the `t` variable, which contains the modified Julian date for each RawTwilightFlatFrame and assign it to the variable `tfrac`. Clear the figure and make a scatter plot of `tfrac` versus `y`.

   ```
   awe> tfrac = [k - math.floor(k) for k in t]
   awe> pylab.clf()
   awe> pylab.scatter(tfrac, y)
   ```

4. Determine the time difference between Greenwich and the place where the RawTwilight-FlatFrames were observed - La Silla. Does this correspond to the longitude of La Silla?

### 9.7.2 Bias level for OmegaCAM

1. Display the bias level as a function of time for chip `ESO_CCD_#96` of the OmegaCAM camera (raw biases are represented in Astro-WISE by the class `RawBiasFrame`).

   ```
   awe> query = (RawBiasFrame.instrument.name == 'OMEGACAM') &\
                (RawBiasFrame.DATE_OBS > datetime.datetime(2011,8,1)) &\
                (RawBiasFrame.chip.name == 'ESO_CCD_#96') &\
   ```

```
              (RawBiasFrame.quality_flags == 0) &\
              (RawBiasFrame.is_valid > 0)
awe> res = [(b.DATE_OBS, b.imstat.median) for b in query]
awe> dat, val = zip(*res)
awe> pylab.clf()
awe> pylab.scatter(dat, val)
```

## 9.8 Galaxy surface brightness analysis

### 9.8.1 Selecting your source

1. **Identify a source in a SourceList which we want to analyse.** Actually you will probably be thinking of an image rather than a SourceList. So lets start with that. Retrieve the image with the following filename:

   ```
   Sci-EHELMICH-WFI-------#842---Coadd---Sci-54552.5317447-4d2189f46deed79e536
   cdbfb0af72ae24187ffd8.fits
   ```

   Presumably you don't want to type in the full name, so use a wildcard:

   ```
   awe> c = CoaddedRegriddedFrame.filename.like('*ffd8.fits')[0]
   ```

   Lets have a look at this image, and select a galaxy from it.

   ```
   awe> c.retrieve()
   awe> c.display()
   ```

   A SourceList exists for this image:

   ```
   awe> sl = (SourceList.SLID == 423431)[0]
   ```

   Make a *skycat* catalog for the image and find the SID of an interesting source.

   ```
   awe> sl.make_skycat_on_sources()
   ```

   Now overplot the catalog in skycat. Open the file and select *Data-Servers -> Local Catalogs -> Load from file....* As filter fill in `*.scat`. You may need to increase the number of sources to get all of them to display. If you click on a symbol the corresponding entry in the catalog is highlighted. Here you can determine the SID of the source.

   Take the source with SID 16246 in the SourceList with SLID 423431.

### 9.8.2 GalPhot: Isophotal analysis: GalPhot

It is instructive to first read the Galphot HOW-TO, in particular to get an idea of the names of classes and methods defined for Galphot.

Isophotes can be fit to sources using the Galphot package. The main classes used to store the information of the fit are *GalPhotModel* and *GalPhotEllipse*. Lets follow an example here.

1. Use the `GalPhotTask` task to do the isophotal fits.

   ```
   awe> task = GalPhotTask(instrument='WFI', slid=423431, sids=[16246], commit=1)
   awe> task.execute()

   # or
   awe> dpu.run('GalPhot', i='WFI', slid=423431, sids=[16246], C=1)
   ```

2. **Inspect the model.** We must start by finding the model we made in the previous step in the database. This can be done by selecting the most recent GalPhotModel created by yourself.

```
awe> m = (GalPhotModel.SLID == 423431).user_only().max('GPID')
```

The methods *get_model()*, *get_residual()*, *get_science()* can be used on a GalPhotModel to visually inspect its quality.

```
awe> s = m.get_science()
awe> s.display()
awe> r = m.get_residual()
awe> r.display()
```

The fitted ellipses are stored in the model, and they can also be obtained:

```
awe> m.ellipses[0].r
```

For a description of the ellipse parameters see the help page of one of the ellipses:

```
awe> help(m.ellipses[0])
```

You can obtain/visualize the ellipse parameters:

```
awe> ellipses = m.get_model_parameters()
awe> rad = [e['r'] for e in ellipses]
awe> pos = [e['pos'] for e in ellipses]
awe> pylab.scatter(rad, pos)
```

For more information see the Galphot HOW-TO.

### 9.8.3 GalFit: 2D Parametric fits to a galaxy surface brightness distribution

The main classes used to store Galfit models are *GalFitModel* and a number of classes named e.g. *GalFitSeric*, each of which is a function that can be fit to the data. See the first section of the Galfit HOW-TO for an overview of the important classes defined for using Galfit in Astro-WISE.

1. Here too, **we first need to identify a source on which we want to run Galfit**. We can take the same source as in the previous example, the source from the SourceList with SLID 423431 and SID 16246.

2. **Create a Sersic model for the galaxy where the index of the Sersic model is fixed at 3.** The model components are specified as a list of dictionaries. Each dictionary describes one component. (Physical) component parameters come in 3 variants e.g.: *ix, x, dx*, respectively the initial, final and error values. Additionally the parameter is set to be fixed or free with the *free_x* parameter.

```
awe> task = GalFitTask(instrument='WFI', slid=423431, sids=[16246],
                       models=[{'name': 'sersic', 'iN': 3, 'free_N': 0}],
                       commit=1)
awe> task.execute()

# or
awe> dpu.run('GalFit', i='WFI', slid=423431, sids=[16246],
             m=[{'name': 'sersic', 'iN': 3, 'free_N': 0}], C=1)
```

3. Again we need to find the result in the database:

```
awe> m = (GalFitModel.SLID == 423431).user_only().max('GFID')
```

4. Now we can inspect the model: retrieve the residual image of the science and the model.

```
awe> sci = m.get_science()
awe> res = m.get_residual()
awe> mod = m.get_model()
awe> sci.display()
etc.
```

5. Have a look at the parameters of the model:

```
awe> m.show_model_parameters()

# or
awe> m.info()

awe> m.components[0].info()
```

For more information see the Galfit HOW-TO.

## 9.9 Interoperability between Astro-WISE and Virtual Observatory software

In this tutorial we will use SAMP connectivity to find bright blue galaxies in a specific WFI field in a graphical, interactive way. You will need to have javaws (java webstart) installed on your system.

### 9.9.1 SAMP

SAMP (Simple Application Messaging Protocol) is a communication protocol for astronomical tools. Several pieces of software have SAMP support, we will use TOPCAT (a table viewer) and Aladin (an image viewer).

1. **Select one of your own SourceLists**, or use SourceList with SLID 135591 (from a WFI V band image.) We will assume the SourceList is called `sl`.

   ```
   awe> sl = (SourceList.SLID == 135591)[0]
   ```

2. **Starting other SAMP applications.**
   Start the following programs:

   - Aladin (this will start a SAMP HUB)
   - TOPCAT

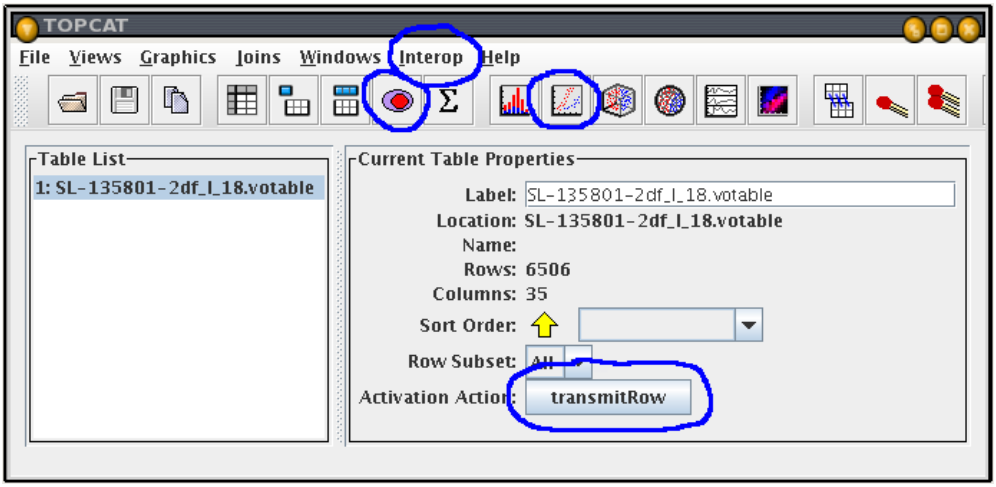



Figure 9.1: The TOPCAT main window. The buttons used is this tutorial are circled. From top to bottom, from left to right: the 'Interop' menu, the 'Row Subset' button, the 'Scatter Plot' button and the 'Activation Action' button.

**Solution:**
Starting these programs requires java, people in Leiden can try the computer `para1` if there is no (or an incorrect) java version installed on their machine. TOPCAT can be started from the terminal by typing

```
awe> os.system('topcat &')
```

Alternatively you could use `javaws` (if available) to start both TOPCAT and Aladin.

```
awe> os.system('javaws http://aladin.u-strasbg.fr/java/Aladin-proto.jnlp &')
awe> os.system('javaws http://andromeda.star.bris.ac.uk/~mbt/topcat/topcat-full.jnlp &')
```

3. **Connect Astro-WISE to the hub and transmit your SourceList (and its frame) through SAMP.**

```
awe> from astro.services.samp.Samp import Samp
awe> samp = Samp()
awe> samp.broadcast(sl.frame)
awe> samp.broadcast(sl)
```

4. **Select some sources in TOPCAT and send their SIDs to the awe-prompt.** You should see your SourceList loaded in the TOPCAT 'Table List' in its main window. Use TOPCAT to select specific sources of your interest and send their SIDs to the `awe`-prompt.

   **Solution A ('highlight' sources):**

   - In the main TOPCAT window, press the (empty) button next to 'Activation Action' and choose 'Transmit Row' from the 'Set Activation Action' window that pops up, and close it.
   - Create a 'Scatter Plot' and display your favorite attributes.
   - Click on an interesting source in the plot.
   - Retrieve the SID in the `awe`-prompt with

     ```
     awe> samp.highlighted(sl)
     837
     ```

   **Solution B ('select' sources):**

   - From the main TOPCAT window create a 'Scatter Plot' and display your favorite attributes.
   - Use the 'select region' tool to select multiple interesting sources at once.
   - Click the 'select region' button again to create a subset containing the sources (you need to name the subset).
   - Open the 'Row Subsets' window by clicking its button in the main TOPCAT window.
   - On the 'Row Subsets' window, select your subset and press the 'broadcast subset' button.
   - Retrieve the SIDs in the `awe`-prompt with

     ```
     awe> samp.selected(sl)
     [0, 2, 820, 983, 1037, 1090, 1093, 1139, 1175, 1197, 1292, 1392]
     ```

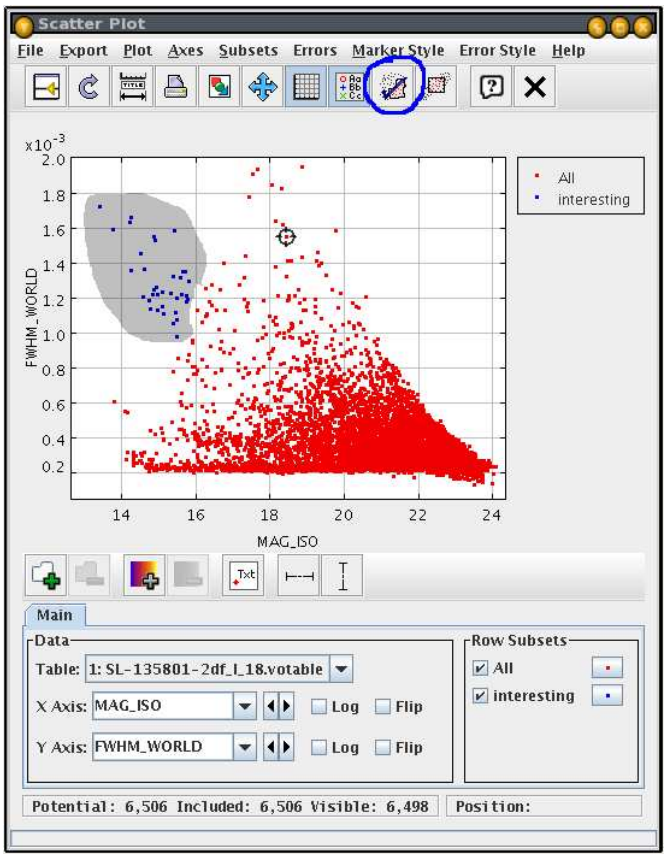



Figure 9.2: The TOPCAT Scatter Plot window. In blue the 'Select Region' button has been circled. With this tool the blue points have been selected and labeled 'interesting' (Solution B). One of the points is selected with the black cross hair and send over SAMP (Solution A)

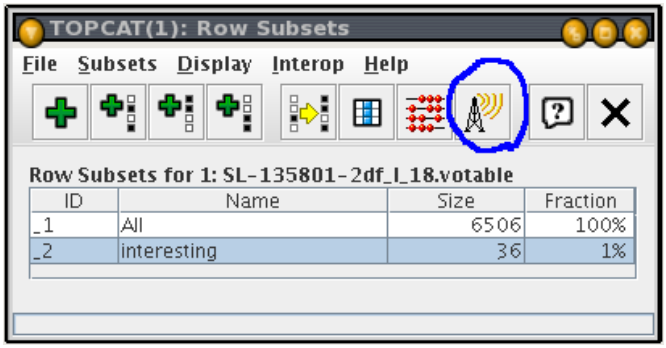



Figure 9.3: The TOPCAT Row Subset window. The 'Transmit Subset' button is encircled, it is grayed out when no subset is selected yet.

## 9.10 Where to go next after this tutorial

The tutorials written here discuss a number of key parts of Astro-WISE, but they do not intend to be exhaustive. Here are collected some things that may be of further interest.

### 9.10.1 Manual, HOW-TO's and other documentation

The Astro-WISE manual can be found here: http://www.astro-wise.org/docs/Manual.pdf.

Most of the manual (the HOW-TO's) can be read online. See here (http://www.astro-wise.org/portal/aw_howtos.shtml). Use the menu on the left to navigate the HOW-TO's. Note that the suggested reading order is more or less top-to-bottom in that menu.

A number of existing number-crunching programs are used within Astro-WISE, such as SExtractor (source extraction), SWarp (regridding/coaddition), NumPy (various), Eclipse (image arithmetic, statistics etc.). Their manuals, as well as other documentation can be found on this page (http://www.astro-wise.org/portal/aw_documents.shtml)

### 9.10.2 Web-services

This tutorial does not cover the web-services that are part of Astro-WISE. In short there are web-services for viewing the contents of the database, for processing data, and for (in-)validating existing data products. There is a Guided Tour that familiarizes one with them. You can find it at the Astro-WISE homepage (http://www.astro-wise.org) in the upper-left box.

### 9.10.3 Source code

Our source code is maintained using the CVS (Concurrent Versions System) software. CVS allows multiple people to work on a set of files, stores these files in a central place and keeps track of all changes to the files. The code is viewable on request for Astro-WISE members on our CVS website (http://cvs.astro-wise.org). See also the CVS HOW-TO (http://www.astro-wise.org/portal/howtos/man_howto_cvs/man_howto_cvs.shtml).

Here is a short overview of the directory structure:

```
Top of directory structure:
awe:

awe/astro:

Package      | Provided functionality
-------------------------------------------------------------------------------
config       | Startup and environment/configuration files
database     | Astronomy specific database modules
experimental | Experimental modules
external     | Python wrappers for tools such as LDAC, Sextractor and Swarp
filerecipes  | Data reduction recipes for use without database
instrument   | Instrument specific modules
main         | Modules for the pipeline processing of images
plot         | Various plotting modules
recipes      | Data reduction recipes (to create classes in "main")
services     | (Web-) services for astronomy
test         | Unittests
toolbox      | Scripts for ingestion, installation, various maintenance
```

```
util          | Utility modules usable throughout the code
-------------------------------------------------------------------------------

awe/common:

Package     | Provided functionality
-------------------------------------------------------------------------------
config      | Startup and environment/configuration files
database    | Persistency mechanism and implementations of its interface
log         | Logging and messaging
math        | Mathematical Python routines such as statistics & least squares
net         | General network-related python modules
services    | Common (web-) services
toolbox     | Scripts for e.g. maintenance of the database, installation, etc.
util        | Routines for compression, checksumming, datetime manipulation.
-------------------------------------------------------------------------------
```

### 9.10.4 Links

The Howto page at our homepage is found under *Astro-WISE information system − > Howtos & Manual*. Use the menu on the left to navigate the HOW-TO's. Here are some interesting ones:

- Observing guidelines (*General − > Observing Guidelines*)
- Context (*AW Environment − > Context*)
- Inspect method of BaseFrames (*Visualization − > Inspections*)
- Inspect method of PhotSrcCatalog (*Visualization − > Photometric Catalog*)
- Combining individual CCDs (*Visualization − > Multi-extension FITS*)

See also the glossary of Astro-WISE concepts etc. (http://www.astro-wise.org/portal/glossary.shtml).

# Chapter 10

# Calibration Pipeline: overview

The calibration part of the Astro-WISE Environment is divided into **three** major pipelines : a bias pipeline, a flat field pipeline, and a photometric pipeline. All of these are composed of several fine-grained, atomic processing steps known as tasks. In the following sections, these pipelines are discussed. A summary is given of the atomic tasks that make up the various pipelines, and their place therein is described. Also given are a few examples of how the different tasks can be steered through the DPU interface. For the interface and use of every individual task, please read the corresponding HOW-TO.

## 10.1 The atomic tasks and their context

The atomic tasks that make up the calibration pipelines are summarized in Table 10.1, together with their role in the system and the identifier under which these are known to the DPU interface.

The sequence of tasks that make up the bias and flat field pipelines is shown in Figure 10.1. The sequence of tasks that should be run for the photometric calibration is shown in Figure 10.2. In these figures, each one of the individual tasks is represented by one box. The arrows indicate the flow of the pipeline, and the shaded parts show particular (optional) branches therein.

### 10.1.1 The bias and flatfield pipelines

The bias and flat field pipelines are pretty straightforward : no difficulties or surprises here. Note, however, that after having derived the masterflat, the processing continues in the photometric pipeline (hence the arrow at the end of the line in Fig. 10.1).

### 10.1.2 The photometric pipeline

The photometric pipeline is more tricky than the bias or flatfield pipeline for two reasons : (1) a piece of the image pipeline must be run to process the photometric standard fields (hence the `Reduce` and `Astrometry` boxes in Fig. 10.2), (2) the optional branch of the illumination correction is an iterative loop that in Fig. 10.2 actually runs backwards (characterising the illumination variation is more of an interactive process).

## 10.2 Examples of running the atomic tasks with the DPU

Make master biases for all CCDs of OmegaCAM for a certain bias template:

Table 10.1: The atomic processing steps that make up the various calibration pipelines and the identifiers through which these can be selected by the user from the DPU interface. Note that not all processing steps are available through the DPU interface.

| Pipeline | Processing step | Purpose | DPU identifier |
|---|---|---|---|
| **Bias** | `ReadNoise` | Deriving the read noise | `ReadNoise` |
| | `Bias` | Creating a master bias | `Bias` |
| | `HotPixels` | Creating a hotpixel map | `HotPixels` |
| | `Gain` | Derive the gain | `Gain` |
| **Flat field** | `DomeFlat` | Creating a domeflat | `DomeFlat` |
| | `ColdPixels` | Creating a coldpixel map | `ColdPixels` |
| | `TwilightFlat` | Creating a twilightflat | `TwilightFlat` |
| | `MasterFlat` | Creating a masterflat | `MasterFlat` |
| | `FringeFlat` | Creating a fringeflat | `FringeFlat` |
| **Photometric** | `PhotCalExtractResulttable` | Measuring fluxes of standard stars | `Photcat` |
| | `PhotCalExtractZeropoint` | Deriving the zeropoint | `Photom` |
| | `PhotCalMonitoring`[1] | Monitoring the atmosphere | - |
| | `IlluminationCorrectionVerify`[1] | Characterizing illumination variations | - |
| | `IlluminationCorrection` | Creating an illumination correction frame | - |

[1]This processing step is performed on a single node

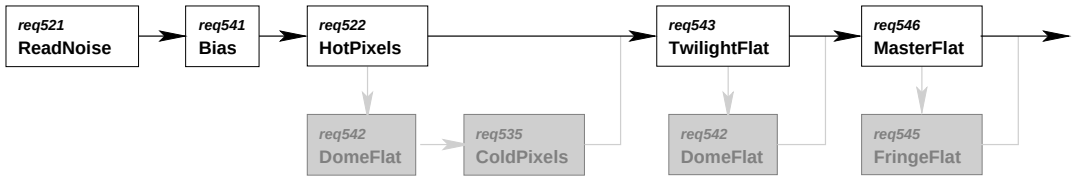



Figure 10.1: The order and flow of the atomic processing steps in the bias and flat field pipelines. The lightly shaded parts show the (optional) branches.

```
awe> dpu.run('Bias', instrument='OMEGACAM', template='2014-07-04T10:39:28')
```

Make a twilightflat for the specified night and filter:

```
awe> dpu.run('TwilightFlat', i='OMEGACAM', d='2014-06-29', f='OCAM_r_SDSS')
```

Make photometric catalogs from a particular standard field exposure:

```
awe> dpu.run('Photcat', i='OMEGACAM', raw=['OMEGACAM.2014-04-26T23:55:32.384_32.fits'])
```

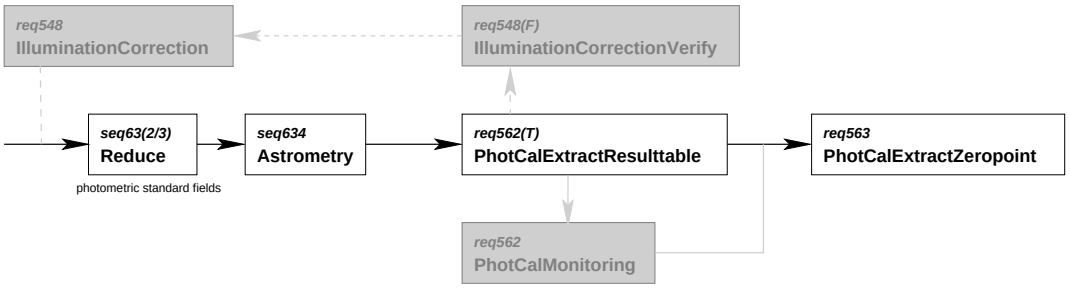



Figure 10.2: The order and flow of the atomic processing steps needed to do the photometric calibration. The lightly shaded parts show the (optional) branches.

# Chapter 11

# Calibration: Read noise

## 11.1 HOW-TO Derive the read noise

### 11.1.1 What is the read noise?

The read-out noise is the noise introduced in the data by the read-out process of CCDs. It is measured from pairs of bias exposures. The rms scatter of the differences between two bias exposures is computed and divided by $\sqrt{2}$; this is the read noise in ADU.

### 11.1.2 Querying

The read noise value is stored in the database using the *ReadNoise* class. A query using the *select* method such as the following will select the most recently created, valid ReadNoise object available for the given instrument, date and chip:

```
awe> rn = ReadNoise.select(instrument='OMEGACAM', date='2000-04-28',
                           chip='ESO_CCD_#65')
awe> print rn.read_noise
```

### 11.1.3 Deriving the read noise

To derive the read noise using the DPU (for all 32 CCDs of OmegaCAM simultaneously in this example) you can do the following (again fill in instrument and date as appropriate):

```
awe> # Example using distributed processing
awe> dpu.run('ReadNoise', i='OMEGACAM', d='2014-04-28', C=1)
```

To derive the read noise on your own machine you can do the following:

```
awe> # Example using a Task (single CCD)
awe> task = ReadNoiseTask(instrument='OMEGACAM', date='2014-04-28'
                          chip='ESO_CCD_#65', commit=1)
awe> task.execute()
```

# Chapter 12

# Calibration: Bias

## 12.1 HOW-TO Create a Bias

### 12.1.1 Bias correction using a bias image

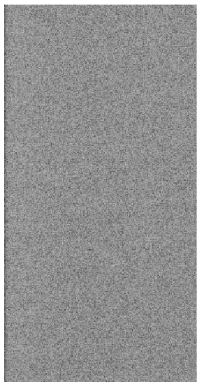


Figure 12.1: A typical bias exposure

One basic step in calibrating raw science images is debiassing. Every exposure by the camera contains a non-zero "bias" level, introduced by the AD converter on the FIERA. Because the bias is literally added by the instrument it needs to be **subtracted** from every exposure. A bias image is a zero-second exposure and typically for each night multiple bias images (5-10) are averaged to create a "master" bias image, which is subtracted from the science images. By using a bias image to correct for bias, any structure in the bias is corrected for, especially also any structure orthogonal to the CCD readout direction. A disadvantage of using bias images is that there may be a significant time difference between the determination of the bias and the (science) images that you want to correct. Therefore a different method (overscan correction) is often used when the bias level of a camera is variable on relatively short timescales.

*In* `AWE` *the bias image used to correct raw science images is called the BiasFrame.*

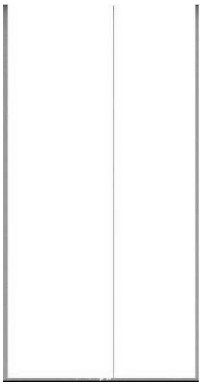


Figure 12.2: Pre- and overscan regions in an image. The central part of the image has much higher intensity than the overscan regions, the color scale is such that it is white. A bad column is visible in the central part of the image.

### 12.1.2 Bias correction using pre- or overscan regions

Creating bias images is not the only way to correct images for bias. Pre- and overscan regions can be used for this purpose as well. These regions are strips of ∼100 pixels at the edges of each image. Usually these regions are not physical pixels that exist on the CCD. Instead the regions are produced by letting the CCD read its readout register about 100 times without having it move charge from the pixels into the readout register ("y direction") or within the readout register to the read port ("x direction"). The advantage of using this method is that pre- and overscan regions are obtained simultaneously with each (science) image, so any short timescale variations in the bias level are accounted for. The disadvantage is that a bias value is not determined per pixel (as in an image), but at best per line or per column.

### 12.1.3 `AWE`: combining both methods

In `AWE`, both methods can be used together. It is possible to subtract the overscan values from the bias images themselves, and then create a master bias image. This master bias image will have an average level of 0, approximately. If you then use the overscan regions in the science images as well as the master bias image, you will correct for both structure in x direction, as well as variations in the bias level. Obviously, if the bias changes in level as well as structure on short time scales, there is little that can be done.
In `AWE`, bias images are assumed to be available for your data. Bias images are always required. However, it is possible to use a number of different overscan correction methods. When running tasks, the method of overscan correction can be specified with the option "overscan". Possible values are:

- 0 – No overscan correction
- 1 – Use median of the (entire) prescan region in X direction
- 2 – Use median of the (entire) overscan region in X direction
- 3 – Use median of the (entire) prescan region in Y direction
- 4 – Use median of the (entire) overscan region in Y direction
- 5 – Use per-row average value of the prescan region in X direction

- 6 – Use per-row average value of the overscan region in X direction (default)
- 7 – Use per-row average value of the prescan region in X direction, smoothing the averages over 50 rows
- 8 – Use per-row average value of the overscan region in X direction, smoothing the averages over 50 rows
- 9 – Use per-row average value of the prescan region in X direction, smoothing the averages over 10 rows
- 10 – Use per-row average value of the overscan region in X direction, smoothing the averages over 10 rows

### 12.1.4 Syntax, examples

To derive a master bias image, it is necessary to derive ReadNoise objects first (if they are not already present). See the ReadNoise HOW-TO (11.1) for more information.
Now derive the master bias as follows:

```
awe> dpu.run('Bias', instrument='OMEGACAM', date='2014-04-28', overscan=6, commit=1)
```

Or using short options:

```
awe> dpu.run('Bias', i='OMEGACAM', d='2014-04-28', oc=6, C=1)
```

where "oc", or "overscan" is one of the values described in the previous section.

# Chapter 13

# Calibration: Hot pixels

## 13.1 HOW-TO Create a HotPixelMap

### 13.1.1 What is a hot pixel map?

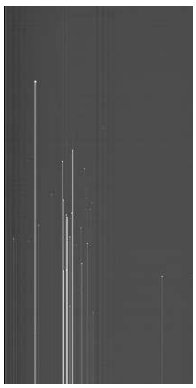


Figure 13.1: BiasFrame containing several bad columns of hot pixels.

A hot pixel map is an image of so-called hot pixels in the CCD detector. Hot pixels are pixels which indicate high values despite not being illuminated. These pixels are detected in bias images because biases have an exposure time of 0 seconds; they are not illuminated. In Astro-WISE the **HotPixelMap** is derived from the **BiasFrame**. Hot pixels destroy the value of all pixels behind the broken pixel as charge is moved through it during the read-out process of the CCD. The result is a **bad column** (see figure 13.1).

The HotPixelMap is a **mask image**: good pixels have a value of 1 and bad pixels a value of 0.

### 13.1.2 Making a hot pixel map

To derive a HotPixelMap, type in the following at the `AWE` prompt:

```
awe> # Example using distributed processing (all CCDs simultaneously)
awe> dpu.run('HotPixels', i='OMEGACAM', d='2014-04-28', C=1)
```

or

```
awe> # Example using a Task (single CCD)
awe> task = HotPixelsTask(instrument='OMEGACAM', date='2014-04-28',
                          chip='ESO_CCD_#77', commit=1)
awe> task.execute()
```

# Chapter 14

# Calibration: Cold pixels

## 14.1 HOW-TO Create a ColdPixelMap

### 14.1.1 What is a cold pixel map?

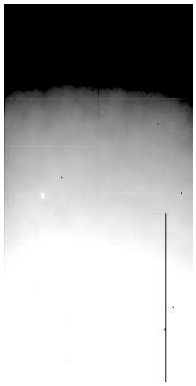


Figure 14.1: DomeFlatFrame containing a bad column of cold pixels.

A cold pixel map is an image of so-called cold pixels in the CCD detector. Cold pixels are broken pixels which report low or zero counts even when illuminated. These pixels are determined from flat-field exposures because those have high counts (relative to the night sky background) across the image. In Astro-WISE the **ColdPixelMap** is derived from the **DomeFlatFrame**. Cold pixels destroy the value of all pixels behind the broken pixel as charge is moved through it during the read-out process of the CCD. The result is a **bad column** (see figure 14.1).

The ColdPixelMap is a **mask image**: good pixels have a value of 1 and bad pixels a value of 0.

In Astro-WISE all pixels that deviate substantially from the other pixels in the (dome) flat-field are considered "cold" even though brighter pixels are also detected.

### 14.1.2 Making a ColdPixelMap

To derive a ColdPixelMap, type in the following at the `AWE` prompt:

```
awe> # Example using distributed processing
awe> dpu.run('ColdPixels', i='WFI', d='2000-04-28', f='#842', C=1)
```

or

```
awe> # Example using a Task (single CCD)
awe> task = ColdPixelsTask(instrument='WFI', date='2000-04-28', chip='ccd50',
                           filter='#842', commit=1)
awe> task.execute()
```

# Chapter 15

# Calibration: Gain

## 15.1 HOW-TO Derive a gain

### 15.1.1 Definition

The gain is the conversion factor between the signal in ADU's supplied by the readout electronics and the detected number of photons (in units $e^-$/ADU). The gain factors are needed to convert ADU's in raw bias-corrected frames to the number of electrons, i.e. detected photons. For OmegaCAM a procedure (template) to determine the gain is defined, which involves taking two series of 10 dome flatfield exposures with a wide range of exposure times. Derive the rms of the differences of two exposures taken with similar exposure (integration time). Exposure differences of pairs should not exceed 4%. The regression of the square of these values with the median level yields the conversion factor in $e^-$/ADU (assuming noise dominated by photon shot noise).

### 15.1.2 Deriving the gain

For most instruments default gain values have been determined and are in the system, so it is usually not necessary to make your own values. The class used to store the gain in the database is called *GainLinearity*. Existing gain values can be obtained for example with this query from the awe prompt (fill in the appropriate instrument, observing date and chip):

```
awe> q = (GainLinearity.instrument.name == 'OMEGACAM') & \
         (GainLinearity.template.start > datetime.datetime(2014,7,2)) & \
         (GainLinearity.template.start < datetime.datetime(2014,7,4))
awe> print q[0].template.start, q[0].gain
2014-07-03 12:35:56 2.49576806277
```

There is a recipe to use the 20 domeflats to determine the gain. The recipe can be used like this:

```
awe> task = GainTask(instrument='OMEGACAM', template='2014-07-03 12:35:56',
                     chip='ESO_CCD_#65')
awe> task.execute()
```

# Chapter 16

# Calibration: Flat-field

## 16.1 HOW-TO Create a Flat-field

### 16.1.1 Flat-fielding

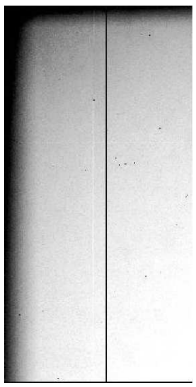


Figure 16.1: A typical flat-field exposure

A basic step in calibrating raw science images is flat-fielding. A flat-field is the response of the telescope-camera system to a source of uniform radiation. Typically, a flat-field does **not** look flat. The flat-field is a multiplicative effect; the differences in attenuation of light over the field of view depend on exposure level. Science images are therefore **divided** by a flat-field normalized to 1. There are several ways of determining a flat-field. The attempt in the case of both dome and twilight flats (see the next sections) is to take approximately 5-10 images with an exposure level of around 20000-30000 ADU, combine and then normalize them. Flat-fields usually contain a large scale gradient, out-of-focus images of dust present on the filter and dewar window, as well as in focus features of dust on the CCDs or defects in the CCD. In other words, flat-fields contain anything that obstructs the light as it falls on the CCD(s).

*In* `AWE`*, the flat-field image used to correct raw science images is called the MasterFlatFrame.*

### 16.1.2 Dome flat fields

A flat-field can be obtained by observing a screen on the inside of the dome of the telescope, which is illuminated by lights. This is called a domeflat in `AWE`. The advantage of dome flat-

fields is that it is easy to repeatedly obtain a good signal to noise image of around 20000 ADU. Disadvantages are that the direction of entry of light in the telescope is different from that during the night, and that it is very difficult to illuminate a screen in such a way that it is a source of uniform radiation.

*In* AWE *a flat-field made from a set of dome flats is called the DomeFlatFrame*

### 16.1.3 Twilight flat fields

Usually flat-fields are obtained by observing the sky during evening and/or morning twilight. During this time the sky better approximates uniform illumination than easily possible with dome flats, and light enters the telescope in much the same way as during the night. Disadvantages are that the brightness of the sky changes rapidly during twilight, and it can be difficult to obtain at least 5 good flat-fields of around 20000-30000 ADU. Time is always an issue. In addition, twilight flats taken in near-darkness can contain stars.

*In* AWE *a flat-field made from a set of twilight flats is called the TwilightFlatFrame*

### 16.1.4 Night-sky ("super") flats

Raw science images have a non-flat background, attributed to flat-field effects. Information about how to flat-field science images therefore is present in the science images themselves. It is possible to use a combination of approximately 10 or more **science** images as a flat-field. Any common structure can be attributed to flat-field effects and is stored in the NightSkyFlatFrame. In the Astro-WISE system, such flats are used as a correction on the other flats, and a new MasterFlatFrame must be derived incorporating the NightSkyFlatFrame.

*In* AWE *a flat-field made from a set of (reduced) science frames is called the NightSkyFlatFrame*

### 16.1.5 Combining flats into a master flat

The procedure to go from raw dome flat fields and raw twilight flat fields to the master flat that is used to flat-field science images is as follows. First the raw dome flats are combined into a master dome flat. This is done by normalizing the raw flats to 1 and stacking them in a cube. Then, for the same pixel in the different input images, the average value is calculated, while rejecting any outliers. This value is the value of the master (dome/twilight) flat for that pixel. This procedure is applied when making the master dome, as well as the master twilight flat.

In the Astro-WISE system, master dome and master twilight flats are combined. Master twilight flats are used to obtain the large scale structure, while master dome flats are used to obtain the small scale structure. In addition a night-sky flat may be incorporated into the final master flat.

### 16.1.6 Syntax, examples

To derive a master flat image, it is necessary to make a master dome flat as well as a master twilight flat.

Make a master dome flat. Note that it is necessary to specify the type of overscan correction (see the Howto for bias correction) that you want to use, which should be the same as that was used in creating the master bias image:

```
awe> # Example using distributed processing
awe> dpu.run('DomeFlat', i='WFI', d='2000-04-28', f='#842', C=1)
```

```
awe> # Or specifying a different overscan correction method
awe> dpu.run('DomeFlat', i='WFI', d='2000-04-28', f='#842', oc=0, C=1)
```

Now make the master twilight flat:

```
awe> dpu.run('TwilightFlat', i='WFI', d='2000-04-28', f='#842', C=1)
awe> # Or specifying a different overscan correction method
awe> dpu.run('TwilightFlat', i='WFI', d='2000-04-28', f='#842', oc=0, C=1)
```

Finally, make the master flat:

```
awe> dpu.run('MasterFlat', i='WFI', d='2000-04-28', f='#842', C=1)
```

### 16.1.7 Using the master dome or master twilight directly

When you explicitly do not want to combine the master dome flat and master twilight flat, for example because there simply are no raw dome flats available this is also possible. A parameter "ct" or "combine" needs to be specified in order to do this. The value of the "combine" parameter can be one of:

- 1: combine the master dome and master twilight flats (default)
- 2: use only the master dome flat
- 3: use only the master twilight flat

Syntax example:

```
awe> dpu.run('MasterFlat', i='WFI', d='2000-04-28', f='#842', ct=3, C=1)
awe> # or using the long option name
e> dpu.run('MasterFlat', i='WFI', d='2000-04-28', f='#842', combine=3, C=1)
```

### 16.1.8 Using night sky flats

In AWE, night sky flats are applied as a correction to the master flat, thereby creating an new master flat as the end-product. To use night sky flats, one must follow these steps:

- 1: Derive the master flat
- 2: Derive the night sky flat
- 3: Rederive the master flat, specifying that a query for night sky flats should be performed

Syntax example for step 2:

```
awe> dpu.run('NightSkyFlat', i='WFI', d='2000-04-28', f='#842', o='CDF4_?', C=1)
```

Syntax example for step 3:

```
awe> dpu.run('MasterFlat', i='WFI', d='2000-04-28', f='#842', nightsky=1,
             combine=1, C=1)
# or using shorter options
awe> dpu.run('MasterFlat', i='WFI', d='2000-04-28', f='#842', n=1,
              ct=1, C=1)
```

# Chapter 17

# Calibration: De-fringing

## 17.1 HOW-TO Correct fringes in science images

Thinned CCD detectors may display an effect called "fringes" when filters towards the red end of the visible spectrum are used (e.g. Johnson/Cousins R, I and Z filters). Fringes are a thinned-film interference effect invoked by the presence of distinct emission lines produced in the atmosphere. The structure of the fringes in a CCD is determined by variations in the thickness of the CCD's silicon layer. In addition, the amplitude of fringes depends on atmospheric conditions and exposure time. Fringes are an additive effect; correction is achieved by creating a *FringeFrame* calibration image, which is scaled and subtracted from science images after the bias and flat-field are applied.

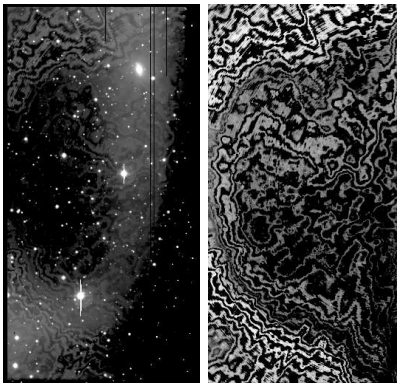


Figure 17.1: A science image in the I-band, displaying fringes (left) and a corresponding *FringeFrame* calibration image (right).

OmegaCAM i-band observations can contain large-scale background fluctuations even after flat-fielding which can cause problems with defringing. Background subtracted images are therefore used in the creation of the fringe frames, and the scale factor is derived from background subtracted science frames.

### 17.1.1 Creating a FringeFrame

*FringeFrame*s are created from a subset of all science images taken during a night, by calculating a median image of the cube of the selected science images. It is best to avoid exposures with nearly identical pointings or extended sources to prevent artifacts due to stars or galaxies in the median image.

In the following example all science images for a particular night, filter and CCD are selected.

```
awe> task = FringeTask(instrument='WFI', date='1999-06-16', filter='#845',
chip='ccd54')
awe> task.execute()
```

Using the DPU for all CCDs:

```
awe> dpu.run('FringeFlat', i='WFI', d='1999-06-16', f='#845')
```

Alternatively, a subset of raw science images can be specified manually, by giving their filenames:

```
awe> task = FringeTask(raw=['science1.fits', 'science2.fits', 'science3.fits'])
awe> task.execute()
```

Using the DPU:

```
awe> dpu.run('FringeFlat', raw=['science1.fits', 'science2.fits', 'science3.fits'])
```

### 17.1.2 De-fringing science images

If a FringeFrame is available for the appropriate CCD and filter, it will automatically be used when science images are reduced (see section 21.2).

# Chapter 18

# Calibration: Astrometry

## 18.1 HOW-TO Derive Astrometry

To derive an astrometric calibration for a given reduced science frame, a `AstrometricParametersTask` object is used. A global astrometric solution can be derived using the `GAstromTask` (see §18.3).

### 18.1.1 AstrometricParametersTask Example

If an astrometric solution is suspect in any way, or if the methods or calibration catalogs are improved, a new astrometric solution can be derived without having to recalibrate the `RawScienceFrame` from scratch. This is accomplished with the `AstrometricParametersTask`. The most ideal way to do this is with the DPU and is illustrated below:

```
awe> dpu.run('Astrometry', i='WFI', red_filenames=['Sci-USER-WFI-#877-red-53664
.5.fits', ...],
             C=0)
```

where `i` is the instrument name (mandatory argument), `red_filenames` is a file list of `ReducedScienceFrames`, and `C` is the commit switch used to commit the results to the database. Without the DPU, the task would be run like:

```
awe> task = AstrometricParametersTask(red_filenames=['Sci-USER-WFI-#877-red-536
64.5.fits'],
...                                   commit=0)
awe> task.execute()
```

where `red_filenames` is a list of filenames of `ReducedScienceFrames` and `commit` is the commit switch. The advantage of using the DPU over this method is that it automatically loops over a list of `ReducedScienceFrame` objects and can run the task on more than one CPU simultaneously.

### 18.1.2 Astrometric calibration - a detailed description

Under the hood of the pre-cooked recipes described in §18.1, a set of LDAC routines is called. This section describes these routines and their role in deriving the astrometric calibration. All steps are performed on catalogs of extracted objects and a reference catalog of astrometric stars. For single frame and global frame astrometric processing, the following steps are made:

1. Preastrometric pairing and approximate determination of the affine transformation parameters (`preastrom`)

2. Application of the affine transformation to the extraction catalog (`aplastrom`)
3. Association of extracted and reference catalogs (`associate`)
4. Filter extranous pairs
5. Building of a pairs catalog based on the derived associations (`make_ssc`)
6. Derivation of the full astrometric solution (`astrom`)
7. Conversion of the astrometric solution parameters to PV parameters (`make_distort`)

**preastrom**

Preastrometric pairing is done on the basis of the WCS information in the FITS header of the image. For the image area the reference objects are extracted from the reference catalog and their (alpha, delta) values are converted to the (x, y) coordinates of the image. Then the x and y distances between all extracted and reference objects are derived. In this distances space a concentration is found through boxcar smoothing. The boxcar has a width of 2×POS_ERROR and the maximum distance area used for searching is 2×MAX_OFFSET. The found peak defines the offset between the WCS information from the FITS header and the actual pointing of the instrument. If the scaling and rotation of the image is well represented by the WCS the peak in the distance plane will be tight. The peak will be smeared, however, if scaling and/or rotation are incorrectly stated in the WCS header information. For example, a scaling error of 1% in a 4000 pixel high CCD will cause the distances to vary over 4000/100 = 40 pixels, so a POS_ERROR of 20 is appropriate in that case.

This offset thus found is applied to the extraction catalog and then a triangulation method is used to derive the remaining affine parameters. For a detailed description of the triangulation method see the LDAC pipeline documentation. In short, vectors connecting triplet of objects are used and a peak is sought in the length/position-angle plane of the vectors. The resulting affine parameters are applied and the number and rms of the fit are reported. To accept the derived affine transformation the reported rms should be less than RMS_TOL.

**aplastrom**

Application of the derive affine transformation to the extracted objects catalogs results in adding a Ra and Dec column to this catalog that represent a good approximation of the true object position. In addition to the position the shape parameters (A_WCS, B_WCS, E_WCS, THETA_WCS) are also transformed using the affine transformation.

**associate**

Association of the extracted objects catalog with the reference catalog is done on the basis of the positional coincidence of extracted and reference objects based on the overlap of their shape parameters. Each object is represented by the 5-tuplet (Ra, Dec, A_WCS, B_WCS, THETA_WCS). Whenever an extracted object and a reference object overlap with eachother so that for one object the (Ra, Dec) falls within the (Ra, Dec, S×A_WCS, S×B_WCS, THETA_WCS) area a pair is created, where S = INTER_COLOR_TOL. For good seeing conditions stellar images have A_WCS values in the order of 0.5 arcsec. Given the inaccuracies of the preastrometric affine transformation one wants to seek a larger area for the associated reference object then just inside the stellar image area. This is done by multiplying the shape parameters by a factor S.

Note that when A_WCS and B_WCS are large, because their is a very bright object in the field, the S multiplication size may cause all reference stars in that large area to be associated with the one bright star. For this reason the next step is run.

### Filtering

Filter extranous pairing removes all very large associations. A good filtering criterion is a pairing distance less than 3 arcseconds.

#### make_ssc

Given the output from the filtered association result for administrative reasons a shuffling of information is necessary and a pairs catalog is build.

#### astrom

Deriving an astrometric solutions is what we are all after here. For a detailed description of how the astrometric solution is determined see the LDAC pipeline documentation. Here a short description of this process is given. The affine transformation is extended to a multidimensional polynomial of degree AstrometricParameters.astromconf.PDEG. In general a 2 dimensional polynomial is good enough to represent the deformation of the light path through the telescope and the imperfections of the plate mount in the focal plane of the telescope. The least square fit of the estimated astrometric solution using the pair information is performed in an iterative process where 3 sigma outliers (pairs) are discarded for the next iterative step. Outliers are removed because propper motion or image artifacts may have created erroneous pairs. The number of iterations steps NITER should be enough, 5 seems a good choice for many applications.

For multi frame processing, Global astrometry, pairs among extracted objects were also created in the previous steps. These pairs participate in the solution as well and because they generally are in over abundance with respect to the extracted objects vs. reference objects pairs they dominate the solution accuracy. Presuming the pointing differences among overlappin exposures are small, the astrometric solution for each pointing could be identical (FDEG = 1 0 0). This may not be the case. In extreme conditions the astrometric solution among the different pointing could be independent of eachother, however, their might be a gradual change of each of the astrometric parameters with pointing. This is represented by a Chebychev polynomial of a certain degree. For each astrometric solution polynomial parameter one can choose a Chebychev polynomial degree. FDEG = 1 0 0 means a Chebychev polynomial of degree 1 for the astrometric distortion polynomial of degree 1, a Chebychev polynomial of degree 0 for the astrometric distortion polynomial of degree 2, etc.

#### make_distort

The final result of the astrometric solution is represented in an internal format that has to be converted to standard WCS patameters.

## 18.2 HOW-TO Create Global Astrometric SourceLists

To derive a global astrometric solution, special `SourceLists` need to be made. For this purpose `GAstromSourceListTask` objects are used. The global astrometric solution can be derived using the `GAstromTask` (see §18.3).

### `GAstromSourceListTask` Example

The `GAstromSourceLists` required for the `GAstromTask` are made with the `GAstromSourceListTask`. Using the DPU this can be done for example like this:

```
awe> dpu.run('GAstromSL', i='WFI', o='M31_?', d='2000-04-28', c='ccd50',
f='#842', C=0)

or

awe> dpu.run('GAstromSL', red=['Sci-#842-ccd50-Red.fits'], C=0)
```

When running locally the equivalent commands are:

```
awe> task = GAstromSourceListTask(instrument='WFI', object='M31_?',
                                  date='2000-04-28',
                                  chip='ccd50', filter='#842', commit=0)
awe> task.execute()

or

awe> task = GAstromSourceListTask(red_filenames=['Sci-#842-ccd50-Red.fits'],
                                  commit=0)
awe> task.execute()
```

where 'instrument' is the instrument name (mandatory argument), 'object' is the object name or wildcard pattern, 'date' is the date of the observations, 'chip' is the CCD name, 'filter' is the filter name, 'red_filenames' is a list of filenames of `ReducedScienceFrame`s, and 'commit' is the commit switch used to commit the results to the database. In the first example, a sourcelist is created from a `CoaddedRegriddedFrame`, while in the second example the same is done from a single `ReducedScienceFrame`.

## 18.3 HOW-TO Create a Global Astrometric Solution

To derive a global astrometric solution, a `GAstromTask` object is used. Before this can be done, GAstromSourceLists *must* be made (see §18.2).

### 18.3.1 GAstromTask Example

An astrometric solution can be improved by using overlapping sources of multiple chips and multiple pointings. This is known as a global astrometric solution and is derived using the `GAstromTask`. The most ideal way to run this task is with the DPU and is illustrated below:

```
awe> dpu.run('GAstrom', i='WFI', o='2df_I_5, f='#879', C=0)
```

where `i` is the instrument name (mandatory argument), `o` is the object name or wildcard pattern, `f` is the filter name, and `C` is the commit switch used to commit the results to the database. Remember, the `GAstromSourceLists` required for the global astrometric solution (GAS) should be created before this task is run. Without the DPU, the task would be run like:

```
awe> task = GAstromTask(instrument='WFI', object='2df_I_5', filter='#879',
...                     commit=0)
awe> task.execute()
```

where `instrument` is the instrument name, `object` is the object name or wildcard pattern, `filter` is the filter name and `commit` is the commit switch.

### 18.3.2 Finding your GAstrometric object

After the `GAstrometric` object is created, it may not be easy to find. This is because of the nature of the `GAstrometric` object. It creates the astrometric solution, but does not store it. The results are stored individually in the `AstrometricParameters` objects, one per input `ReducedSceinceFrame`. There are also limitations on the number and type of attributes it contains.

The simplest way to find a `GAstrometric` object is to start with the `ReducedScienceFrames` that went into the solution. You can use the same parameters used in the `GAstromTask` to do this:

```
awe> query = ReducedScienceFrame.select(instrument='WFI', object='2df_I_5',
...  filter='#879')
awe> len(query)
40
```

Once you have the `ReducedScienceFrames` that went into the `GAstrometric` solution, you can query for one of the `AstrometricParameters` objects and use this to locate the GAstrometric solution and all the associated `AstrometricParameters` objects.

```
awe> red = query[0]
awe> ap = (AstrometricParameters.reduced == red).max('creation_date')
awe> gas = GAstrometric.residuals == ap.residuals
awe> len(gas)
0
```

Here, we choose the most recent `AstrometricParameters` object for one of the `ReducedScienceFrames` in the query, and use the `residuals` attribute to query for the `GAstrometric` object (the individual solutions all use the same global residuals file). In this example, no `GAstrometric` object exists because the above task was run with `commit=0`.

### 18.3.3 Getting the best GAstrometric solution

Blindly combining exposures taken over large spans of time or greatly varying observing conditions will likely result in a poor `GAstrometric` solution. Understanding the concept of fixed focal-plane geometry is vital for high-quality solutions in Astro-WISE.

Global astrometry in Astro-WISE takes advantage of fixed focal-plane geometry applicable under certain conditions for certain telescopes. This means that any difference in the focal-plane geometry from pointing to pointing is assumed to change in a linear fashion only, with higher order distortions remaining constant (e.g., only relative translations of the entire focal plane in RA and Dec are corrected for). When valid, this assumption of fixed focal-plane geometry adds information to the system, benefitting the astrometric solution. Generally, only closely matched sets of exposures taken within strict temporal limits[1] (generally less than one hour between first and last exposure) will demonstrate a fixed focal-plane geometry. This condition minimizes differences in telescope flexure caused by different altitude and azimuth locations.

The standard method in Astro-WISE of combining multiple sets of closely matched exposures is to obtain global solution for each set independently instead of blindly combining them all at once. The independently derived solutions can be applied to the source frames to create frames regridded to the same grid target (spatial reference point on a fixed grid). These regridded frames can then be coadded together to create the final combination (e.g., using SWarp, Eclipse, or PyFITS/NumPy for the image combination).

[1] Strict spatial limits also aid in the global solution, but are not a condition for a fixed focal-plane geometry. The most optimal results are derived from sets of exposures with greater than 90% overlap due to the larger number of sources common to all exposures.

## 18.4 HOW-TO Inspect an Astrometric Solution

Astrometric solutions are created as a result of the `AstrometricParametersTask` and the `GAstromTask`. To determine the quality of the astrometric solution, several methods can be used. The primary method is to inspect the astrometric solution with the `AstrometricParameters inspect()` method. An alternate qualitative method is to view a calibrated catalog overlayed on the image.

> **Please Note**, not all inspection methods are currently available in the `AWBASE` checkout, but *are* in the **current** checkout. These methods have been noted. To use them, see the Getting Started section in the FAQ (Chapter 25) for details on using a different checkout.

### 18.4.1 AstrometricParameters and GAstrometric inspect() methods

#### The Plots

An `AstrometricParameters` object can be inspected by plotting the residuals of the solution versus themselves and versus position. This is exactly what the `inspect()` method does. For a single chip (`AstrometricParameters`), one figure is created with 5 panels. These five panels plot (from top down):

- DDEC versus DRA with line-connected histograms of their distributions
- DRA versus RA
- DDEC versus DEC
- DRA versus XPOS
- DDEC versus YPOS

where DRA is $RA_{reference} - RA_{extracted}$ in arc-seconds, DDEC is $Dec_{reference} - Dec_{extracted}$ in arc-seconds, RA is in degrees, DEC is in degrees, XPOS is the X pixel position of the extracted source, and YPOS is the Y pixel position of the extracted source.

Also included at the top of the figure is the `DATE_OBS` of the source `ReducedScienceFrame`; the mean RA (`<RA>`) and mean Dec (`<DEC>`), both calculated from the distribution plotted; the number of pairs plotted–the same as the number of pairs used in the astrometric solution (`N`); the chip name of the source `ReducedScienceFrame` (`CHIP:`); the mean RA residual (`<DRA>`), mean Dec residual (`<DDEC>`), and sample standard deviation of each distribution (values following the `+-`), all based on the distribution plotted; the `RMS` (root-mean-square) value of the distance of the residual pairs with respect to the DDEC/DRA origin (0,0); and the maximum distance of any residual pair from the DDEC/DRA origin (`Max`). There are also RMS and N values within the first panel. Their significance will be clear when seen in the context of the multi-chip solution.

The multi-chip case (`GAstrometric`) plots exactly the same information, but with multiple pointings per chip, one figure per chip. Each pointing has a different color to distinguish it from the other pointings. Also, the first panel includes the RMS and N values for each pointing individually. The values above the first panel are all calculated with respect to $ALL$ the data plotted.

In addition to the multi-chip reference residuals, an entire set of overlap residuals figures is created. Instead of DRA being $RA_{reference} - RA_{extracted}$, it is $RA_1 - RA_2$, both extracted from their respective frames. $RA_1$ and $RA_2$ will *never* be from the same chip and pointing. The Dec values are similar. There is no other difference between the previous multi-reference figures and these multi-overlap figures.

Lastly, two more figures are created: all reference residuals and all overlap residuls. These figures simply show all the data from all the chips of their respective data set. Each pointing is color-coded identically to the individual chip figures. The reference figure in this multi-chip, multi-pointing case is directly equivalent to the single-chip figure.

**The Methods**

There are currently two ways to run the inspect method for either case. The most straightforward of these is to simply set the `inspect` switch in either the `AstrometericParametersTask` or in the `GAstromTask` when either is run:

```
awe> task = AstrometricParametersTask(red_filenames=['Sci-USER-WFI-#877-red-536
64.5.fits'],
...                                    inspect=1, commit=0)
awe> task.execute()

or

awe> task = GAstromTask(instrument='WFI', object='2df_I_5, filter='#879',
...                       inspect=1, commit=0)
awe> task.execute()
```

In the other method, an `AstrometricParameters` or `GAstrometric` object is instantiated from the database, and its `inspect()` invoked:

```
awe> ap = (AstrometricParameters.reduced.filename == 'Sci-USER-WFI-#877-red-536
64.5.fits')[0]
awe> DataObject(pathname=ap.residuals).retrieve()
awe> ap.inspect()

or

awe> gas = (GAstrometric.gasslist.filename == 'GAS-2df_I_5-53760.5784035')[0]
awe> DataObject(pathname=gas.residuals).retrieve()
awe> gas.inspect()
```

**Modifying the Default Output**

The inspection figures described above are displayed to the screen and written to PNG files. This behavior can be modified, but explaination of these techniques is beyond the scope of this HOW-TO. For the latest documentation for attempting these modifications, simply view the online help (docstrings) of the inspect method(s) and plot class(es) used to create these figures:

```
awe> help(AstrometricParameters.inspect)
awe> from astro.plot.AstrometryPlot import AstromResidualsPlot
awe> help(AstromResidualsPlot)
```

### 18.4.2 Applied inspection methods

The previous sections described the built-in inspection methods showing *predicted* results of the derived solutions in `AstrometricParameters` and `GAstrometric` objects. This section describes extended inspection methods of the derived solutions *applied* to `RegriddedFrames` and

CoaddedRegriddedFrames. All these inspection methods deliver a plot in the same 5-panel form as the built-in inspect() methods, but using different source catalogs for the residuals. Also, the details and latest usage information can be found in the methods' docstrings accessed via the help() command.

#### AstrometricParameters plot_residuals_to_usno() method

- Temporarily in current checkout only.

This plot displays source position residuals between the corrected catalog positions performed by LDAC or sources positions extracted from a RegriddedFrame corrected with the same parameters and the USNO-A2.0 reference catalog. Setting the source parameter to 'solution' or 'applied' selects either the catalog used in the *solution* or a catalog extracted from a RegriddedFrame to which the solution parameters have been *applied*, respectively.

#### AstrometricParameters plot_residuals_to_regrid() method

- Temporarily in current checkout only.

This plot displays source position residuals between the corrected catalog positions performed by either LDAC or SExtractor and sources extracted from a RegriddedFrame corrected with the same parameters. Setting the derived_type parameter to 'solution' or 'sextractor' selects either the catalog used in the *solution* or a catalog extracted from a ReducedScienceFrame by *SExtractor* to which the solution parameters have been applied to the header, respectively.

#### CoaddedRegriddedFrame plot_regrid_residuals() method

- Temporarily in current checkout only.

This plot displays source position residuals between a given RegriddedFrame and all other overlapping frames, all that participate in a CoaddedRegriddedFrame. Setting the use_coadd switch (use_coadd=True) displays source position residuals between the CoaddedRegriddedFrame and all RegriddedFrames that went into its creation. It plots a given RegriddedFrame source position against the average source position from the CoaddedRegriddedFrame.

### 18.4.3 Image inspection method

- Temporarily in current checkout only.

The multi-purpose inspect() method used by all frames inheriting from BaseFrame can create a plot that can be used to display the qualitative residuals on the pixel level by using either difference images or multi-color images using the same mechanism for inspecting individual frames. The detailed usage of this method can be found in section 23.1. The general idea for this purpose is to inspect one RegriddedFrame, setting the compare parameter (compare=True) and specifying the other RegriddedFrame with the other parameter. The routine automatically compares only the overlapping region of the two frames.

```
awe> reg0 = RegriddedFrame(pathname=filename0)
awe> reg1 = RegriddedFrame(pathname=filename1)
awe> reg0.inspect(compare=True, other=reg1)  # common region of reg0-reg1
```

### 18.4.4 Overlaying a calibrated catalog

This method requires a RegriddedFrame obtained from the RegridTask. It needs to be first loaded into SkyCat:

```
awe> q = RegriddedFrame.reduced.filename == 'filename.reduced.fits'
awe> os.system('skycat %s' % (q[0].filename))
```

First, set the desired cut level via the "Auto Set Cut Levels" button, or with "View:Cut Levels...". Next, overlay the catalog by choosing "Data-Servers", then "Catalogs", then "USNO at ESO"[2]. In the dialog that comes up, choose "Search" and all the sources known to the USNO survey will be plotted in circled cross-hairs. They can now be compared directly with the sources on the underlying frame. When inspecting the corellation, remember that the USNO catalog is accurate only to about 0.3 arc-sec.

If the RegriddedFrame was not created from the ReducedScienceFrame, it will need to be regridded with the RegridTask before the inspection above can be carried out. This is because there exist projection effects (distortions) in the ReducedScienceFrame. The RegridTask can be run via the DPU or locally as shown in the example below:

```
awe> dpu.run('Regrid', i='WFI', d='2001-01-01', f='#845', o='Science2', C=0)
```

```
awe> regrid = RegridTask(date='2000-01-01', chip='ccd50', filter='#845',
...                      object='Science2', commit=0)
awe> regrid.execute()
```

### 18.4.5 Examine the AstrometricParameters values

To look at the AstrometricParameters for a given ReducedScienceFrame, the AstrometricParameters objects of interest must first be located in the database:

```
awe> q = AstrometricParameters.reduced.filename == 'WFI.2001-02-16T01:42:31.289
_1.calibrated.fits'
awe> len(q)
1
```

```
awe> dt = datetime.datetime(2005,1,1)
awe> q  = (AstrometricParameters.instrument.name == 'WFI')
awe> q &= (AstrometricParameters.filter.name == '#845')
awe> q &= (AstrometricParameters.chip.name == 'ccd50')
awe> q &= (AstrometricParameters.creation_date > dt)
awe> len(q)
1199
```

The first example shows the a query for an AstromtricParameters object by its source ReducedScienceFrame's filename. The second shows a more general search based on instrument, filter, chip, and date.

> NOTE: Dates and times in the Astro-WISE database environment are generally in the form of datetime objects. Therefore, when querying for them, a datetime object must be used. The main exception is the `select()` method, but this method is not universally implemented at this time.

[2] There are other catalogs available, but the USNO catalog was used in the astrometric solution and should fit well.

## 18.5 HOW-TO Troubleshoot Astrometry Quality Problems and Improve Solutions

Astrometry, like photometry, requires good data to get a good result. The process to derive an astrometric solution is a complex one and is described in brief in HOW-TO Derive Astrometry (see §18.1), and in detail in the LDAC pipeline documentation and many other astrometry references.

The scope of this document is to describe specific things that can go wrong in an astrometric solution and approaches to improve the astrometric solution, both in the context of the Astro-WISE Environment.

If a correction method below refers to a parameter to be set, it can be set through the process parameters interface (see §8.4). In short:

```
> pars = Pars(AstrometricPatameters)
> pars.process_params.SOME_VALUE = 100
> ...
> AstrometricParametersTask(..., pars=pars.get()).execute()
```

As with any aspect of data reduction, a good rule of thumb if you have any problem with astrometry, inspect the data if no obvious error presents itself. Please take a look at HOW-TO Inspect Astrometry (see §18.4) to see how this is done in the Astro-WISE Environment.

### 18.5.1 Errors in LDAC

LDAC is the software suite used to create the astrometric solution, whose steps are given in HOW-TO Derive Astrometry (see §18.1). Despite (or because of) the best efforts of programmers, software can always have problems or can exit for very specific reasons. To help make the error messages less cryptic, LDAC's Python wrapper has been upgraded to output an ordered list of debugging information in case of a software failure. All individual programs linked by a common instance of the LDAC class (e.g., preastrom, associate, aplastrom, astrom, etc.) will have their output logged where it normally would have been supressed so that any warnings preceeding the failure can be seen.

The normal output of an individual LDAC program will show only the command-line, the program version, and program description. If an error occurs, the configuration file, warnings, and errors are also added to this output. If there are programs run from the same LDAC instance, those that have run will have their full output preceeding the output of the failed program. The bottom line is that errors should be easier to diagnose for both the user and the programmer.

Viewing the log output is the best way to try and troubleshoot any astrometry error. The LDAC class instance has verbosity set to NORMAL by default which gives the minimum of information when all is well, and everything when there is a problem. By setting the individual routine settings to VERBOSE (or even DEBUG) instead:

```
> pars = Pars(AstrometricPatameters)
> pars.preastromconf.VERBOSE = 'VERBOSE'
> ...
> AstrometricParametersTask(..., pars=pars).execute()
```

will increase the verbosity in the event of an error. If you need even more and want as much information as LDAC can produce, you can set the LDAC class verbosity in the code to VERBOSE or DEBUG:

```
ldac = LDAC.LDAC(verbose='DEBUG')
...
ldac.preastrom(...)
...
```

This latter method is recommended *only if it is absolutly necessary*, as it involves code changes and a very large amount of program output to go through.

#### Specific conditions from LDAC log output

If the solution fails with a software exception, deciphering the output is the only way to figure out what happened. The terminal Exception (the very last error message) should contain some indication of what happened. If there is no help there, a perusal of the previous log messages can be helpful. If they are not, posting the output to the Issues list is the quickest way to get expert help.

Below are included some very common log messages that may indicate specific problems. Some terms are defined for this section only for clarity:

- *num_ext* is the number of extracted objects, found in the line containing the text: "objects of field"
- *num_ref* is the number of reference objects from the extracted region, found in the line containing the text: "objects from reference catalog"

The first term is the program where a specific error occurs. This is followed by criteria, a possible cause, and a course of action:

- `preastrom` – if *num_ext* is low (on the order of 10) and *num_ref* is high (on the order of 100), there are still more objects to be extracted, lower extraction threshold: `AstrometricParameters.sexconf.DETECT_THRESH`
- `preastrom` – if *num_ext* and *num_ref* are higher (on the order of 100, 1000, respectively, or both of the order of 100), it is possible the AstrometricCorrection is bad or was not run, check the alignment with the reference catalog, see HOW-TO Inspect Astrometry (see §18.4)
- `preastrom` – if *num_ext* and *num_ref* are swapped in magnitude (on the order of 1000, 100, respectively), there may be a cluster where the reference catalog is necessarily incomplete, try using the `GAstromTask` (see §18.3).
- `preastrom` – if *num_ext* is about a factor of 2 larger than *num_ref*, there may have been an error in the pointing where the sources are doubled, inspect the image as described in HOW-TO Inspect Astrometry (see above)

### 18.5.2 Quality Control (QC) Values Exceeded

Astrometric solutions in Astro-WISE, like all other calfiles, have QC limits to determine if the calfile has the required quality. If these limits are exceeded, the solution is marked as bad and there may be methods to help improve the quality. This section addresses specific cases where quality may be improved.

### NREF too low

Low NREF (number of output reference pairs) can be caused by a number of situations, from low SNR data (low EXPTIME), to very sparce fields, to large differences in filter band between the extracted and reference catalogs. There are two ways that too low an NREF can harm a solution:

1. `AstrometricParameters.process_params.MIN_NREF` sets a limit in `preastrom` (MIN_OBJ) where it refuses to process catalogs with fewer than this number of extracted or reference objects in the defined region. An LDAC error will result from this.

2. `AstrometricParameters.process_params.MIN_NREF` also sets a limit for successful solutions from `astrom` (i.e., solutions not affected by the previous point) to be flagged as poor. In this case, it is the final number of reference pairs used in the solution. Only a flag will be set in this case.

To improve a solution with an NREF too low where either of these situations occurred, more reference pairs are necessary. There are a few ways to accomplish this. One method would involve decreasing the extraction threshold:

- `AstrometricParameters.sexconf.DETECT_THRESH`

below the default of 10.0 to increase the number of extracted sources that can be considered for pairing. Another method involves the use of a global solution and requires a dithered set of observations via the `GAstromTask` (see §18.3). One last method involves adding more chips from the same exposure, but it is not formalized in Astro-WISE except during ingestion (the method is called `AstrometricCorrection`).

### NREF too high

Usually, more reference pairs are better, but only to a certain limit. If the density of sources is too high (e.g., greater than approximately 1200 in a 2k×4k CCD chip at 0.238 arcsec/pixel, i.e., the case with WFI) and the association radius too high (e.g., greater than approximately 3 arcsec), false solutions due to random associations can occur. Also, attempting to solve astrometry using data taken in a significantly different filter from reference catalog can add confusion if density is high. For these reasons, a QC limit has been established for this condition.

Very little can be done to help this kind of data other than trying to decrease the number of reference pairs by decreasing the number of extracted sources. This can be done by increasing the extraction threshold:

- `AstrometricParameters.sexconf.DETECT_THRESH`

Please note, this does nothing to decrease the number of reference sources! As such, the helpfullness off this method is limited. In addition to reducing the number of extracted objects, the association radius between the extracted and reference catalogs:

- `AstrometricParameters.associateconf.INTER_COLOR_TOL`

should be decreased to reduce the chance of false associations. An associated parameter, ISO_COLOR_TOL, has no effect for local solutions involving only onechip. Changing it in this case is unnecessary.

#### SIG_DRA or SIG_DDEC, or RMS too high

If the standard deviation of ΔRA or ΔDEC, or the RMS of the solution is outside of the QC limits, the solution is considered poor. To improve this, decreasing the number of reference pairs (NREF) might help, assuming those removed were of poorer quality (e.g., fainter). A better method is to decrease the radius over which associations can occur. Adjusting:

- `AstrometricParameters.associateconf.ISO_COLOR_TOL`

to lower values will do just this with respect to sources in a reference catalog, and adjusting:

- `AstrometricParameters.associateconf.INTER_COLOR_TOL`

will affect only overlapping sources in the same catalog (e.g., global solutions).

What is being adjusted here is the factor of the object's *size* (actually an ellipse as set by the source extraction parameters RA, DEC, a, b, Θ). Only objects whose position overlaps the ellipse of another object will be associated with it. These settings should be used with care as setting these values too low can result in too few reference pairs very quickly.

### 18.5.3 Problems with the Solution

In addition to software exceptions and QC limit issues, the data can be of such a type to prevent a solution from converging. This section describes some specific configurations that can be adjusted to fix certain problems.

#### preastrom

The first step of solving astrometry in Astro-WISE is running `preastrom` to solve the translation, rotation, and scaling of the input data.

The recommendations below have been incorporated into the `AstrometricParameters` routines as a fallback in case `preastrom` fails with the basic settings. If there are still `preastrom` failures at the more conservative settings, inspect the data to make sure there is nothing unusual, and proceed carfully.

#### MAX_OFFSET

The first step of `preastrom` is measuring distances between all objects and examining the (ΔX, ΔY) parameter space. The total area that goes into the distance determination is the area covered by the input catalog plus a border of MAX_OFFSET pixels around the catalog area. Thus, if there is a pointing offset greater than MAX_OFFSET pixels, preastrom will never find the proper solution.

If inspection of the logs of failed `preastrom` runs reveals the number of extracted and reference objects are similar and number on the order of 100, there may be a shift beyond the default value of MAX_OFFSET. Inspect the frame as shown in HOW-TO Inspect Astrometry (see §18.4) to see if this is the case. If so, increase the value of MAX_OFFSET to larger than the offset observed. Be aware, however, that if the object density increases in parts of the border area (e.g., presence of a clutser), increasing MAX_OFFSET may have unpredictable results.

#### POS_ERROR

After the (ΔX, ΔY) measuremnts have been made, `preastrom` runs a boxcar smoothing of the (ΔX, ΔY) parameter space with a boxcar size of POS_ERROR pixels, using the peak of that distribution to define the set of data to triangulate on.

If excessive scaling or moderate rotation ($\widetilde{>}$2 degrees) might be causing `preastrom` to fail, increasing POS_ERROR can help if the density is not too high.

### RMS_TOL & SEL_MIN

Once the subset of data is chosen, a triangulation method using angle and distance is used to determine the final rotation and scaling (affine transformation), and final offsets. If the RMS of the fit is above RMS_TOL, SEL_MIN will be increased to choose a larger number of sources and triangulation will proceed iteratively to try to reduce the RMS below RMS_TOL. In practice, this iteration seldom improves things due to the nature of the method and the quality of the input data. Setting a higher RMS_TOL is usually the only way to overcome this affine transformation determination error.

Increasing both RMS_TOL and SEL_MIN will allow a poorer quality preastrometric fit to continue through the pipeline, where the final solution may or may not be of good quality.

#### associate

After the gross, linear corrections have been made, the reference and extracted sources can be associated using `associate`. `associate` has really only one parameter to help improve the precision of the solution. This is the distance within which objects will be associated, and are represented by the two parameters ISO_COLOR_TOL and INTER_COLOR_TOL.

### ISO_COLOR_TOL & INTER_COLOR_TOL

The definitions and optimizations of these parameters was described in §18.5.2 above. Repeating, INTER_COLOR_TOL is for associations between sources in different catalogs (e.g., extracted and reference catalogs), ISO_COLOR_TOL is for associations between sources in the same catalog (e.g., a joined extracted catalog). Setting these to lower values can result in a more precise fit, but should be used with care.

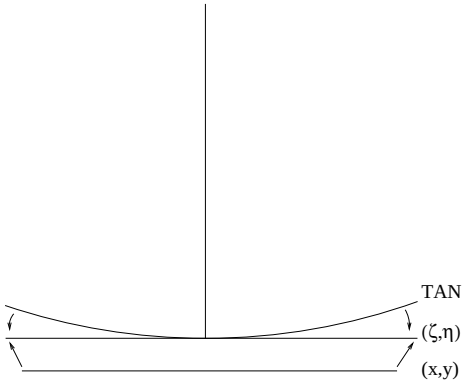



Figure 18.1: A 2-dimensional illustration of mapping of observed sky to idealized sky. The lowest plane (x,y) is the observed sky, and the middle plane ($\zeta$,$\eta$) is the representation of the ideal sky mapped from a tangential (Gnomonic) projection denoted by TAN, the upper-most curved plane. The vertical line represents the radial direction with respect to the detector.

#### astrom

The PDEG parameter describes the polynomial degree used to map the observed sky, the pixel image, denoted by (x,y) to the idealized sky denoted by ($\zeta$,$\eta$), see Fig. 18.1. The required number of associations increases exponentially with degree. Also, the spatial uniformity of the associations becomes more critical with degree. PDEG is applicable for both single-chip and

overlap solutions. When in overlap, however, the FDEG parameter will affect the solution as well.

The FDEG parameter is a list that describes how the mapping of the PDEG degree polynomial occurs from pointing to pointing. The first value in the list maps linear terms (order 1), the second, quadratic terms (order 2), etc. Only the first 3 can ever be changed. Based on the value of each list element, the polynomial coefficents of different pointings can be mapped to each other linearly (FDEG index 1), quadratically (FDEG index 2), etc., or held constant (FDEG index 0). For example, in the default case:

```
FDEG = [1, 0, 0]
```

indicates all first order terms will be mapped linearly between pointings and all higher order terms will be held constant. Similar to PDEG, as FDEG's total value increases, so increases the number of associations and spatial uniformity required to converge.

For single chip fields, it is unlikely that a PDEG $> 3$ will result in a proper fit[3]. To enhance the number of associations over an area of one chip, multiple dithered observations should be used. This enhances both the number and spatial uniformity and allows greater PDEG and FDEG values to be used successfully. See `GAstromTask` (see §18.3) for how to solve astrometry for dithered observations.

[3]Indeed, the number of values stored in the database (PV matrix coefficients) are currently only enough to properly store a solution of PDEG = 3.

# Chapter 19

# Calibration: Photometry

## 19.1 Photometric Reference Catalog and Standard Extinction Curve

The Photometric Reference Catalog is a catalog which contains magnitudes in common bands, coordinates and names for standard stars. The Photometric Reference Catalog and the standard extinction curve are two calibration files that are provided by the system. Both calibration files are directly accessible for the user from the `awe`-prompt.

### 19.1.1 The Photometric Reference Catalog

**Its present contents**

The Photometric Reference Catalog present in the Astro-WISE system is the result of an ongoing project. This section describes the contents of the most recent version of this Photometric Reference Catalog wich has the filename `cal569E_v11.cat`. The catalog *file* can be found in the CVS checkout : `cvsroot/catalog`. This catalog contains stars from four sources: the Landolt catalog, the Stetson catalog, the Sloan Digital Sky Survey (SDSS) Data Release 5 (DR5) catalog and the Preliminary Catalog for the OmegaCAM Secondary Standards Programme which is based on our own observations. For all four sources stellar magnitudes are given in both the Johnson-Cousins $UBVRI$ photometric system and in the Sloan $ugriz$ photometric system (i.e., unprimed).

- **Landolt stars:** the Photometric Reference Catalog contains all 544 stars from the Landolt catalog. The Johnson-Cousins $UBVRI$ magnitudes are taken from Landolt 1992. The Sloan $ugriz$ magnitudes are computed from the Johnson-Cousins $UBVRI$ magnitudes using the transformation equations of Jester et al. 2005 which apply only to stars with $R-I<1.15$. The $ugriz$ magnitudes are not computed for stars which have $R-I>1.15$. Improved coordinates are used for the 526 Landolt stars available at this ESO webpage.
- **Stetson stars:** the Photometric Reference Catalog contains all 39109 Stetson stars (see Table 19.1). The Johnson-Cousins $BVRI$ magnitudes are taken from the on-line catalog of Stetson ($U$-band magnitudes are not available). The Sloan $griz$ magnitudes are computed from the Johnson-Cousins $BVRI$ magnitudes using the transformation equations of Jester et al. 2005 which apply only to stars with $R-I<1.15$. The $ugriz$ magnitudes are not computed for stars which have $R-I>1.15$.

- **SDSS DR5 stars:** the Photometric Reference Catalog contains only SDSS DR5 stars which are located inside 1 square degree fields centered on 22 SA fields (see Table 19.1). The Sloan *ugriz* magnitudes are the psfMags for objects classified as stars available in the SDSS DR5 database. SDSS DR5 quality control flags were taken into account to exclude stars with saturated pixels, being blended with neighboring stars or being too close to the edge of a frame for a given band. This means that stars were excluded if any of the following SDSS DR5 quality flags are raised for any band: COSMIC_RAY, SUBTRACTED, SATURATED, BLENDED, BRIGHT and EDGE. The SDSS DR5 data was retrieved using the python code sdss2refcat.py. The Johnson-Cousins $UBVRI$ magnitudes are computed from the Sloan *ugriz* magnitudes using the transformation equations of Jester et al. 2005 which apply only to stars with $R-I<1.15$. The $UBVRI$ magnitudes are not computed for stars which turn out to have $R-I>1.15$.

- **Preliminary Catalog stars:** we observed 7 of the 22 SA fields listed in Table 19.1 with the WideField-Camera (WFC) on the 2.5m Isaac Newton Telescope at La Palma using Sloan *ugriz* filters. The derived catalog is called the Preliminary Catalog. The SDSS DR5 stars were used as calibrators. The Johnson-Cousins $UBVRI$ magnitudes are computed from the Sloan *ugriz* magnitudes using the transformation equations of Jester et al. 2005 which apply only to stars with $R-I<1.15$. The $UBVRI$ magnitudes are not computed for stars which turn out to have $R-I>1.15$. Further details of the data reduction, photometric calibration and other properties of the Preliminary Catalog are given in Verdoes Kleijn et al. 2006. The PC data have much larger measurement errors than the DR5 data. Additional plots with $u-g$ vs $g-r$ for PC only and $u-g$ vs $g-r$ for PC plus DR5 show the same data as in Fig 4 of Verdoes Kleijn et al. 2006 but as density plots to better show the consistency in stellar locus between the DR5 and PC data. Similar plots for $g-r$ vs $r-i$ for PC only and $g-r$ vs $r-i$ for PC plus DR5 and for $r-i$ vs $i-z$ for PC only and $r-i$ vs $i-z$ for PC plus DR5 are also available. The Preliminary Catalog will be used as starting point to derive the OmegaCAM Secondary Standards with OmegaCAM itself in the first year of operations.

The catalog is in FITS table format and its columns are described in Table 19.2. Whenever a magnitude or its associated error has value 0.0 it means that no value has been determined.

### Retrieving the Photometric Reference Catalog

The Photometric Reference Catalog can be retrieved from the database to the local directory as follows:

```
awe> from astro.main.PhotRefCatalog import PhotRefCatalog
awe> refcat = PhotRefCatalog.get()
awe> refcat.retrieve()
```

which will automatically give the most recent Photometric Reference Catalog in the system. Note the *retrieve* operation in the last line; this is important. The contents of the catalog thus retrieved to the local directory can then be queried using the methods described in §19.1.1.

### Query methods

The Photometric Reference Catalog has six methods for querying/accessing its content. The first four of these are simple methods without parameters. These are the following:

1. `refcat.get_number_of_sources()`,
   which returns the number of sources in he catalog.

Table 19.1: The stars present in the 22 SA fields contained in the Photometric Reference Catalog in `AWE`. The fields cover an area of 1.1×1.1 degree centered on the tabulated coordinates. The number of Landolt standard stars, Stetson standard stars, SDSS DR5 stars and stars from the Preliminary Catalog (PC) for the OmegaCAM Secondary Standard Programme are listed.

| Field | $\alpha$ (J2000) (deg) | $\delta$ (J2000) (deg) | #Landolt | #Stetson | #SDSS DR5 | #PC |
|---|---|---|---|---|---|---|
| SA 51 | 112.663 | +29.828 | 0 | 0 | 214 | 0 |
| SA 57 | 197.171 | +29.384 | 0 | 0 | 952 | 0 |
| SA 68 | 4.146 | +15.844 | 0 | 0 | 1302 | 0 |
| SA 92 | 13.946 | +0.949 | 41 | 213 | 1094 | 6475 |
| SA 93 | 28.783 | +0.824 | 4 | 0 | 1128 | 0 |
| SA 94 | 44.033 | +0.571 | 7 | 0 | 1099 | 0 |
| SA 95 | 58.500 | +0.000 | 45 | 426 | 1093 | 0 |
| SA 98 | 103.021 | −0.328 | 46 | 1116 | 0 | 23840 |
| SA 100 | 133.529 | +0.546 | 6 | 1 | 3343 | 0 |
| SA 101 | 149.112 | −0.386 | 35 | 117 | 1776 | 5591 |
| SA 102 | 163.779 | +0.866 | 5 | 66 | 1517 | 0 |
| SA 103 | 178.779 | +0.556 | 2 | 0 | 1507 | 0 |
| SA 104 | 190.4875 | −0.5292 | 34 | 76 | 1576 | 5701 |
| SA 105 | 204.533 | +0.676 | 4 | 0 | 2172 | 0 |
| SA 106 | 220.533 | +0.427 | 2 | 15 | 2864 | 0 |
| SA 107 | 234.8250 | −0.2631 | 28 | 728 | 3889 | 12006 |
| SA 108 | 248.033 | +0.369 | 6 | 3 | 6148 | 0 |
| SA 110 | 280.6000 | +0.34583 | 39 | 589 | 0 | 38562 |
| SA 112 | 310.529 | +0.524 | 7 | 73 | 12087 | 0 |
| SA 113 | 325.3750 | +0.49944 | 42 | 483 | 4046 | 13947 |
| SA 114 | 340.529 | +0.689 | 9 | 5 | 1957 | 0 |
| SA 115 | 355.779 | +0.888 | 10 | 0 | 1170 | 0 |

Table 19.2: Description of the 29 columns in the Photometric Reference Catalog cal569E_v*.cat.

| column name | description |
|---|---|
| SeqNr | sequence number |
| origin | origin of stellar magnitude:<br>Landolt: Landolt catalog, Stetson: Stetson catalog,<br>SDSS5: SDSS DR5,<br>AW2S: Preliminary Catalog for OmegaCAM Secondary Standards |
| Name | Name of star |
| Ra/Ra_err | Right Ascension / its error (deg) |
| Dec/Dec_err | Declination / its error (deg) |
| Epoch | epoch of coordinates: all J2000 |
| Flag | flag, (not used currently) |
| JohnsonU/JohnsonU_err | Johnson U / its error (mag) |
| JohnsonB/JohnsonB_err | Johnson B / its error (mag) |
| JohnsonV/JohnsonV_err | Johnson V / its error (mag) |
| CousinsR/CousinsR_err | Cousins R / its error (mag) |
| CousinsI/CousinsI_err | Cousins I / its error (mag) |
| SloanU/SloanU_err | Sloan u / its error (mag) |
| SloanG/SloanG_err | Sloan g / its error (mag) |
| SloanR/SloanR_err | Sloan r / its error (mag) |
| SloanI/SloanI_err | Sloan i / its error (mag) |
| SloanZ/SloanZ_err | Sloan z / its error (mag) |

2. `refcat.get_list_of_bands()`,
   which returns a list of the photometric bands supported by the catalog.

3. `refcat.get_source_attributes()`,
   which returns information of the data content of a source. The information is stored in a dictionary with the attribute names of the source as keys and their types as values. To just get the attribute names, do: `refcat.get_source_attributes().keys()`.

4. `refcat.make_skycat()`,
   which dumps the catalog for overplotting in `skycat`.

The following query methods have a more elaborate interface. One thing that should be mentioned is that every single star in the catalog has an index for cross-referencing purposes. The methods below all return dictionaries which use these indices as keys.

In order to retrieve the magnitudes for all the stars in the standard star catalog for one particular photometric band (`mag_id`), type:

```
awe> refcat.get_dict_of_magnitudes(mag_id)
```

which will return a dictionary with the indices of the sources as keys, and as values 2-tuples containing the magnitude and its uncertainty. The `mag_id` is a string that should match one of the entries in the list generated by the `get_list_of_bands` method. If one wants to retrieve the magnitudes for only a subset of stars, it is possible to provide the method call with an additional list of indices:

```
awe> refcat.get_dict_of_magnitudes(mag_id, index_list = [1, 2, 13500])
```

which will only give the magnitudes for the stars 1, 2 and 13500 in the list.

Besides this dedicated method, the Photometric Reference Catalog also has an allround query method that can be used to retrieve any information that is needed. This method and its signature are:

```
awe> refcat.get_source_data(column_list, index_list = index_list)
```

which will return a dictionary with the indices of the sources as keys, and as values lists of the requested data items in the same order as specified in the input `column_list`. The input `column_list` is the list of data items to be retrieved, and the optional `index_list` is used to get data from a subset of stars only. The entries in `column_list` should match the keys of the dictionary that is generated by the `get_source_attributes` method. These entries are strings.

**Examples of use**

Retrieve the V magnitudes of all the stars in the catalog:

```
awe> mag_dict = refcat.get_dict_of_magnitudes('JohnsonV')
```

Retrieve the $g'$ magnitudes of stars 10 to 500:

```
awe> inds = range(10, 500)
awe> mag_dict = refcat.get_dict_of_magnitudes('SloanG', index_list = inds)
```

Retrieve the Ra-Dec-Epoch information from all the stars:

```
awe> info_dict = refcat.get_source_data(['ra', 'dec', 'epoch'])
```

Retrieve the name (`star_id`) of the stars 1 and 2:

```
awe> info_dict = refcat.get_source_data(['star_id'], index_list = [1, 2])
```

Retrieve the names of the catalogs from which the sources originate (`origin`):

```
awe> info_dict = refcat.get_source_data(['origin'])
```

**Using a subset of the catalog for photometric calibration**

As was mentioned in §19.1.1, the Photometric Reference Catalog contains stars from various contributary catalogs. To allow the user (and the system) to see/use/select the stars of only one or a select few of the contributaries, the standard star catalog is outfitted with a filter on the origin column. In the current version of the Photometric Reference Catalog this attribute can have one of the following values: `Landolt`, `SDSS5` and `Stetson` for Landolt, SDSS DR5 and Stetson catalogs, respectively, and `AW2S` for the Preliminary Catalog.
To select/see/use only stars from the Photometric Reference Catalog that originate from the 'Landolt' catalog use:

```
awe> refcat.origin_filter.activate('Landolt')
```

To select/see/use only stars from both the 'Landolt' and 'Stetson' sub-catalogs use:

```
awe> refcat.origin_filter.activate('Landolt', 'Stetson')
```

To select/see/use only stars from the 'Stetson', 'SDSS5' and 'AW2S' sub-catalogs use:

```
awe> refcat.origin_filter.activate('Stetson', 'SDSS5', 'AW2S')
```

Note, that the order in which the origin identifiers appear is not important.
To de-activate the filter so that the full view on the catalog is restored:

```
awe> refcat.origin_filter.deactivate()
```

To check whether the filter is actually switched on:

```
awe> refcat.origin_filter.is_active()
```

which will return either `True` or `False`. By default, the filter is disabled.
It is obvious that the setting of the filter affects the return values of the query methods described in §19.1.1.

### Standard and non-standard photometric bands for OmegaCAM

The OmegaCAM system distinguishes two types of bands: **key** bands, and **user** bands. This distinction has its origin in the OmegaCAM calibration plan, and has implications for the inner workings of the photometric pipeline. The **key** bands are the Sloan *ugri* bands. These bands are fully supported by the OmegaCAM system, and form the core of the contents of the Photometric Reference Catalog. Any other band than these four Sloan bands is a **user** band.

To process data that have been observed in a **user** band, a transformation table is needed. This table contains information about which key bands to use in the transformation, and a set of transformation coefficients. Such a table should be present in the database for every filter that belongs to a user band. The contents of such a table could be so simple as providing a substitute key band for a given user band (e.g. for a Gunn $r$ filter, the Sloan $r$ magnitudes will be used), but could also contain a full-blown transformation with given color and a color-term. The transformation table will be discussed in a separate chapter.

The Johnson-Cousins bands and the Sloan $z$ band are strictly speaking also user bands. However, given the wide-spread use of these systems and for convenience, the necessary transformations have already been performed and the results have been stored together with the Sloan *ugri* key bands. The transformation equations used are discussed in §19.1.1. It is, of course, always possible to override these transformed magnitudes stored in the catalog by providing the system with an appropriate transformation table.

### Using your own catalog with standard stars

The Photometric Reference Catalog described above is the default Photometric Reference Catalog automatically used by `AWE`. This standard catalog has the reserved name `cal569E_v*.cat`, with the version number at the asterisk. This naming convention should **never** be used for your own catalog! In the event that you need to use standard stars which are not provided by the default standard catalog you can create your own catalog and ingest it in `AWE`.

A Photometric Reference Catalog requires a specific format. One can create this format from an input ascii format using the `refcat_generator.py` code available in CVS checkout: `cvsroot/catalog/tools`
Typing at the operating system command line

```
linux> awe refcat_generator.py
```

will generate the help documentation.
Once created, you can ingest it into the system doing the following:

```
awe> from astro.main.PhotRefCatalog import PhotRefCatalog
awe> refcat = PhotRefCatalog(pathname = 'myowncatalog.cat')
awe> refcat.store()
awe> refcat.commit()
```

Note the extra `store` command that will put the file on the fileserver.

### 19.1.2 The standard extinction curve

#### Its present contents

To be written.

#### Retrieving the standard extinction curve

To retrieve the standard extinction curve from the database, do the following :

```
awe> from astro.main.PhotExtinctionCurve import PhotExtinctionCurve
awe> extcurve = PhotExtinctionCurve.get()
```

after which it can be used immediately. For example :

```
awe> extcurve.get_extinction(4861.0)
0.13652400000000001
```

with 4861.0 the wavelength in Å, and the answer in *mag/am*. This method is the only query method defined on the extinction curve.

#### Ingesting the standard extinction curve

Before the photometric pipeline can be used, the system should be initialized by putting a standard extinction curve into the system. **Note : this should only have to be done once by the maintainer of the system**. After initialization, every user of the system has access to the standard extincintion curve. To put a standard extinction curve into the system, do the following :

```
awe> from astro.main.PhotExtinctionCurve import PhotExtinctionCurve
awe> extcurve = PhotExtinctionCurve(pathname = 'cal564E.dat')
awe> extcurve.commit()
```

The standard extinction curve *file* can be found in a separate CVS checkout : `/cvsroot/catalog`. The file `cal564E.dat` represents the La Palma extinction curve in units of *mag/am*.

#### References

Landolt, A.U. 1992, AJ, 104 (1), 340
Jester, S., et al. 2005, AJ, 130, 873
Verdoes Kleijn et al. 2006

## 19.2 HOW-TO make a Photometric Source Catalog

The very **first** processing step in the photometric pipeline **always** consists of deriving catalogs from standard field observations. These observations should be fully reduced, i.e. de-biased and flatfielded (`ReducedScienceFrame` objects), and an astrometric calibration should also be available for the observations (`AstrometricParameters` objects). This step in the processing is the only time when image data actually enters the photometric pipeline. After this point, the rest of the photometric pipeline only works with the catalogs produced in this very first step.

### 19.2.1 Content of the Photometric Source Catalog

The Photometric Source Catalog contains one row per identified standard star with 8 values. These 8 columns are described in table 19.3.

### 19.2.2 Making photometric catalogs from the `awe`-prompt

The catalogs produced in this first processing step are represented by the `PhotSrcCatalog` class. This class is the real 'working' class in the photometric system because instances of it are used throughout. These photometric catalogs are the result of associating a SExtractor catalog with a photometric standard star catalog. The following sub-sections describe the two ways in which such catalogs can be produced from the `awe`-prompt.

#### Using a pre-cooked recipe

Deriving a photometric catalog from the `awe`-prompt using a pre-cooked recipe is done thus:

```
1. awe> from astro.recipes.PhotCalExtractResulttable import PhotcatTask
2. awe> task = PhotcatTask(instrument = 'WFC',
   ...                     raw_filenames = ['r336603_3.fits'])
3. awe> task.execute()
```

To get detailed information about the use of the task, type:

```
awe> help(PhotcatTask)
```

which will, for example, show the parameters that can be passed to the constructor of the task.

#### Using the basic building blocks

A more elaborate but also very enlightening way of making a photometric catalog is by using the basic building blocks of the photometric pipeline themselves. This allows tweaking of the processing down to the nitty-gritty details. The most common way of deriving a photometric catalog in this way is given here:

```
1. awe> from astro.main.PhotSrcCatalog import PhotSrcCatalog
2. awe> from astro.main.PhotRefCatalog import PhotRefCatalog
3. awe> from astro.main.AstrometricParameters import AstrometricParameters
4. awe> from astro.main.ReducedScienceFrame import ReducedScienceFrame
5. awe> frame = (ReducedScienceFrame.filename == 'r336603_3.reduced.fits')[0]
6. awe> query = (AstrometricParameters.reduced == frame) &
   ...          (AstrometricParameters.is_valid == 1)
7. awe> astrom_params = query.max('creation_date')
8. awe> refcat = PhotRefCatalog.get()
9. awe> photcat = PhotSrcCatalog()
```

```
10. awe> photcat.refcat = refcat
11. awe> photcat.frame = frame
12. awe> photcat.astrom_params = astrom_params
13. awe> photcat.refcat.retrieve()
14. awe> photcat.frame.retrieve()
15. awe> photcat.frame.weight.retrieve()
16. awe> photcat.make()
17. awe> photcat.commit()
```

In lines (1)-(4) the relevant classes are imported, and in steps (5)-(8) the necessary dependencies are retrieved from the database. In lines (9)-(16), a `PhotSrcCatalog` object is instantiated, its dependencies are set, and the *make* method is called. In step (17), the `PhotSrcCatalog` object is committed to the database. Note the explicit *retrieve* calls in steps (13)-(15) that access the fileserver.

It is important to realize that the separate steps detailed here are roughly the same as the ones performed by the `PhotcatTask` (§19.2.2).

### 19.2.3 Configuring the photometric catalog

Any `PhotSrcCatalog` object has two knobs that allow the user to configure the behaviour of the *make* method and its results. The first one of these deals with configuring SExtractor, the other one with configuring the *make* method itself.

SExtractor can simply be configured through the `sexconf` attribute of the `PhotSrcCatalog` object. To give an example of how this works:

```
awe> photcat.sexconf.PHOT_APERTURES = 30
```

which tells SExtractor to use an aperture with a diameter of 30 pixels in measuring the `FLUX_APER` of the standard stars. More information about the configuraton of SExtractor can be found in the SExtractor manual.

The configuration of the *make* method itself is done through the `process_params` attribute of the `PhotSrcCatalog` object. To get information about the available configuration options, just type:

```
awe> photcat.process_params.info()
```

which will show the available configurable parameters, their meaning, and their default setting.

### 19.2.4 Inspecting the contents of the photometric catalog

To view the content of a `PhotSrcCatalog` object after it has been made, simply invoke its *inspect* method:

```
awe> photcat.inspect()
```

which will result in an output to screen that looks like the one shown in Figure 19.1. The *inspect* plot shows the magnitudes of the individual standard stars as known to the standard star catalog on the x-axis, and their associated raw zeropoints on the y-axis. The *inspect* method of course also functions on `PhotSrcCatalog` objects retrieved from the database.

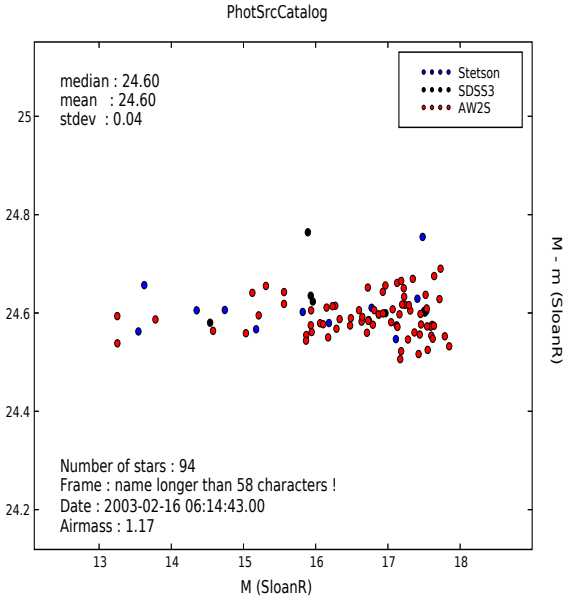



Figure 19.1: The result of invoking the *inspect* method of a `PhotSrcCatalog` object.

### More visualisation options

The *inspect* method of a `PhotSrcCatalog` object re-directs the actual work to a dedicated plot object. This plot object is, by default, configured to show its output on screen with the individual raw zeropoints and magnitudes of the stars contained in the catalog on the vertical and horizontal axis, respectively. However, more visualisation options are available if one uses the plot object directly.

A plot object for `PhotSrcCatalog` can be retrieved through a factory function:

```
awe> from astro.plot.PhotometryPlot import create_plot
awe> plot = create_plot('Photcat')
```

whereafter it will be ready for use. The method to be called for plotting the contents of the `PhotSrcCatalog` object is:

```
awe> plot.show(photcat)
```

where `photcat` is a valid `PhotSrcCatalog` object. Note that calling the *show* method just like that will just produce the same result as the *inspect* method of the `PhotSrcCatalog` object.

Using the plot object allows the user to plot several other quantities on the horizontal axis besides the magnitudes of the standard stars. Other quantities that can be plotted are the X-position, Y-position, and radial position of the stars on the chip. The plot object can be configured to do that as follows:

```
awe> plot.plot_params.XAXIS_TYPE = 'XPOS'
```

with `MAGS`, `XPOS`, `YPOS` and `RADIAL` as possible values. The default used by the *inspect* method is always `MAGS`.

The plot object can also be configured to send its output to a postscript file with a user-specified name:

```
awe> plot.plot_params.FILE_OUTPUT = 1
awe> plot.plot_params.FILE_NAME = 'my_photcat.eps'
```

where the FILE_OUTPUT toggle is set to True.

### 19.2.5 Query methods

Any given PhotSrcCatalog object features a large collection of methods that allow access to its data contents (many of these are actively used by the photometric pipeline). The most important simple methods are:

1. photcat.get_number_of_sources(),
   which returns the number of sources in he catalog.
2. photcat.get_source_attributes(),
   which returns information of the data content of a source. The information is stored in a dictionary with the attribute names of the source as keys and their types as values. To just get the attribute names, do: photcat.get_source_attributes().keys().
3. photcat.get_dict_of_raw_zeropoints(),
   which returns a dictionary with the raw zeropoints of the sources in the catalog as values, and their indices as keys.
4. photcat.get_median_raw_zeropoint(),
   which returns the median raw zeropoint of the sources within the catalog.
5. photcat.get_average_raw_zeropoint(),
   which returns the average and standard deviation of the raw zeropoint distribution.
6. photcat.make_skycat(),
   which dumps the catalog for overplotting in skycat.

The most flexible way of retrieving source information from the catalog is through the following query method:

```
awe> photcat.get_source_data(column_list)
```

which will return a dictionary with the indices of the sources as keys, and as values lists of the requested data items in the same order as specified in the input column_list. The input column_list is the list of data items to be retrieved. The entries in column_list should match the keys of the dictionary that is generated by the get_source_attributes method. These entries are strings.

### 19.2.6 Querying the database

Just like any other persistent data item in the system, a PhotSrcCatalog object has a large collection of attributes that can be used to retrieve it from the database. The most important of these are the instrument, filter, chip and date_obs attributes. The value of the latter corresponds exactly to the DATE_OBS attribute of the ReducedSciencFrame object that went into making the catalog, and serves as the master timestamp for PhotSrcCatalog objects.

**Example queries**

In the first example, the `PhotSrcCatalog` object derived from a science frame taken at 2003-02-11T21:00:00 is retrieved from the database. Only the one that has been marked as valid is requested.

```
awe> date_obs = datetime.datetime(2003,2,11,21)
awe> photcat = (PhotSrcCatalog.date_obs == date_obs) &\
...            (PhotSrcCatalog.is_valid == 1)
```

In the second example, the `PhotSrcCatalog` objects for the night from 2003-02-11 to 2003-02-12 are retrieved, and from these only those that are derived from science frames observed through the WFC broad-band Sloan G filter and from chip A5382-1-7.

```
awe> date_start = datetime.datetime(2003,2,11,12)
awe> date_end = date_start + datetime.timedelta(1)
awe> photcat = (PhotSrcCatalog.date_obs >= date_start) &\
...            (PhotSrcCatalog.date_obs <  date_end) &\
...            (PhotSrcCatalog.filter.name == '220') &\
...            (PhotSrcCatalog.chip.name == 'A5382-1-7') &\
...            (PhotSrcCatalog.is_valid == 1)
```

The last example shows the situation in which several `PhotSrcCatalog` objects have been made from one and the same science frame, and that only the last one created is wanted. The science frame has the name `r300100_4.calib.fits`.

```
awe> photcats = (PhotSrcCatalog.frame.filename == 'r300100_4.calib.fits')
awe> photcat = photcats.max('creation_date')
```

Table 19.3: Description of the 8 columns in a Photometric Source Catalog.

| Column name | Description |
|---|---|
| index | index number of the star as listed in the Photometric Reference Catalog |
| origin | origin of stellar magnitude as listed in Photometric Reference Catalog:<br>Landolt: Landolt catalog,<br>Stetson: Stetson catalog,<br>SDSS5: SDSS DR5,<br>AW2S: Preliminary Catalog for OmegaCAM Secondary Standards |
| ra | Right Ascension (deg) |
| dec | Declination (deg) |
| mag | stellar magnitude in Photometric Reference Catalog,<br>if a transformation table has been applied the value of mag incorporates this transformation. |
| mag_err | error on mag |
| instmag | -2.5*log10(countrate) |
| instmag_err | error on instmag |

## 19.3 Photometric Pipeline (2): Transformation Tables

The photometric pipeline distinguishes between two types of photometric bands: **key** bands, and **user** bands. The **key** bands are the Sloan $u'g'r'i'$ bands. These bands are fully supported by the photometric pipeline, and form the core of the contents of the standard star catalog. Any other band than these four Sloan bands is a **user** band.

Processing data that has been observed in one of the **key** bands is easy in the photometric pipeline: all the necessary data is present in the system. However, an extra data item is required to process data that has been observed in a **user** band. This extra data item is a transformation table. For every filter of which the photometric band is a **user** band, a transformation table should be present in the system. These transformation tables are represented in the photometric pipeline by the `PhotTransformation` class.

### 19.3.1 The data structure of a transformation table

The `PhotTransformation` class represents a table of parameters used to transform magnitudes of standard stars from one photometric system to another (color terms). The equation used to calculate this 'transformed' magnitude $M(\mathrm{T})$ from other bands is:

$$M(\mathrm{T}) = M(\mathrm{prm}) - CT \times [M(\mathrm{scd}) - M(\mathrm{trt})] + C, \tag{19.1}$$

with $CT$ the color term, and $C$ an additional shift that can be applied. The $M(\mathrm{prm})$, $M(\mathrm{scd})$ and $M(\mathrm{trt})$ parameters must each separately be set to any of the bands for which magnitude information is available in the standard star catalog.

The various components of Eqn. 19.1 all map to their own attribute of a `PhotTransformation` object. The $M(\mathrm{prm})$, $M(\mathrm{scd})$ and $M(\mathrm{trt})$ parameters respectively map to the `primary_band`, `secondary_band` and `tertiary_band` attributes. These attributes should always be set. The $CT$ and $C$ parameters are assigned to the `color_term` and `coefficient` attributes of the object. Both these parameters are 0.00 by default. The uncertainties on the two parameters are assigned to the `color_term_error` and `coefficient_error` atrtibutes, respectively.

### 19.3.2 Using a transformation table

The transformation table is used in deriving a photometric catalog, where it is an extra dependency that should be set. *The transformation is applied to the magnitudes of the standard stars as recorded in the standard star catalog.* As is the case for making a photometric catalog *without* a transformation table, there are two ways in which catalogs can be produced from the `awe`-prompt *with* a transformation table.

#### Using a pre-cooked recipe

Deriving a photometric catalog with a transformation table using a pre-cooked recipe from the `awe`-prompt is done thus:

```
1. awe> from astro.recipes.PhotCalExtractResulttable import PhotcatTask
2. awe> raw = 'r336604.fits'
3. awe> task = PhotcatTask(instrument='WFC', raw=raw, chip='A5506-4',
                           transform=1)
4. awe> task.execute()
```

where the extra `transform` boolean switch tells the system to use a transformation table.

**Using the basic building blocks**

The more elaborate way of making a photometric catalog is extended by a few extra 'moves':

```
 1. awe> from astro.main.PhotTransformation import PhotTransformation
 2. awe> from astro.main.PhotSrcCatalog import PhotSrcCatalog
 3. awe> from astro.main.PhotRefCatalog import PhotRefCatalog
 4. awe> from astro.main.AstrometricParameters import AstrometricParameters
 5. awe> from astro.main.ReducedScienceFrame import ReducedScienceFrame
 6. awe> frame = (ReducedScienceFrame.filename == 'r336603_4.reduced.fits')[0]
 7. awe> query = (AstrometricParameters.reduced == frame) &
    ...          (AstrometricParameters.is_valid == 1)
 8. awe> astrom_params = query.max('creation_date')
 9. awe> refcat = PhotRefCatalog.get()
10. awe> transform = PhotTransformation.get('2003-02-11', '#844',
    ...                                     instrument='WFI')
11. awe> photcat = PhotSrcCatalog()
12. awe> photcat.refcat = refcat
13. awe> photcat.transform = transform
14. awe> photcat.frame = frame
15. awe> photcat.astrom_params = astrom_params
16. awe> photcat.refcat.retrieve()
17. awe> photcat.frame.retrieve()
18. awe> photcat.frame.weight.retrieve()
19. awe> photcat.make()
20. awe> photcat.commit()
```

In lines (1)-(5), the `PhotTransformation` class is imported together with the usual suspects. In lines (6)-(10), the necessary dependencies are retrieved from the database. In lines (11)-(19), a `PhotSrcCatalog` object is instantiated, its dependencies are set, and the *make* method is called. Please note step (13), where the extra dependency is set with the transformation table. The photometric catalog is committed to the database in step (20).

### 19.3.3 Retrieving a transformation table from the database

A transformation table for a given filter can be retrieved from the database as shown in the example below:

```
awe> from astro.main.PhotTransformation import PhotTransformation
awe> transform = PhotTransformation.get('2003-02-11', '#844',
...                                     instrument='WFI')
```

which will return the transformation table for filter #844 that is valid for the given date.

### 19.3.4 Inserting a transformation table into the system

To insert a transformation table for a certain filter and instrument into the database, a simple tool is available. This tool is located in the `./awe/astro/toolbox/photometry/` directory of the CVS checkout and is called `ingest_transformation`. To get information about how to use the tool just type:

```
awe $AWEPIPE/astro/toolbox/photometry/ingest_transformation.py
```

and a doc-string will appear on screen. The tool only accepts input for filters and instruments that are actually present in the database. If a certain filter or instrument is not present, the tool refuses to comply.

## 19.4 Photometric Pipeline (3) : Extinction and Zeropoint

This chapter describes how the extinction and zeropoint are derived for a given night. The two steps, for the time being, should be done in tandem. The results are only committed to the database at the very last processing step.

### 19.4.1 Deriving the atmospheric extinction

The photometric pipeline supports several ways of obtaining the atmospheric extinction :

1. using two observations of standard fields at two different airmasses
2. using a default value for the extinction coefficient stored in the database
3. using one single observation of a standard field and a set of already known zeropoints
4. using a combination of a standard extinction curve and an extinction report

The first two of these methods are the most flexible, while the fourth method derives from the *OmegaCAM* photometric calibration plan. It should be pointed out that, contrary to other parts of the photometric pipeline, the first and third way of deriving the atmospheric extinction use input data from whole images. All the extinctions used in the photometric pipeline are in units of *mag/am*.

#### Using two observations of standard fields

The 'classical' case of the two standard field observations taken at two different airmasses is dealt with by the `AtmosphericExtinctionFrames` class. Its dependencies consist of two lists of `PhotSrcCatalog` objects derived earlier from the standard field data, one for every observation. Instances of this class are used as follows:

```
 1. awe> from astro.main.AtmosphericExtinctionFrames import \
    ...                  AtmosphericExtinctionFrames
 2. awe> from astro.main.PhotSrcCatalog import PhotSrcCatalog
 3. awe> import datetime
 4. awe> date_obs_first = datetime.datetime(2003,2,12,20,15)
 5. awe> date_obs_second = datetime.datetime(2003,2,12,20,20)
 6. awe> photcats_1 = (PhotSrcCatalog.date_obs == date_obs_first) &\
    ...               (PhotSrcCatalog.filter.name == '191') &\
    ...               (PhotSrcCatalog.is_valid == 1)
 7. awe> photcats_2 = (PhotSrcCatalog.date_obs == date_obs_second) &\
    ...               (PhotSrcCatalog.filter.name == '191') &\
    ...               (PhotSrcCatalog.is_valid == 1)
 8. awe> extinct = AtmosphericExtinctionFrames()
 9. awe> extinct.polar = list(photcats_1)
10. awe> extinct.equat = list(photcats_2)
11. awe> extinct.make()
```

In lines (1)-(2) the relevant classes are imported, and in steps (6) and (7) the necessary dependencies are retrieved from the database. In lines (8)-(11), an `AtmosphericExtinctionFrames` object is instantiated, its dependencies are set, and the *make* method is called.

### Using a default value from the database

If no suitable data is present to derive the extinction from, it is (or at least should be) possible to retrieve a ready-made extinction coefficient from the database. This particular extinction is covered by the `AtmospericExtinctionCoefficient` class. Such an extinction is retrieved from the database as follows:

```
awe> from astro.main.AtmosphericExtinction import \
...                   AtmosphericExtinctionCoefficient
awe> extinct = AtmosphericExtinctionCoefficient.get('191', 'A5382-1-7')
```

which will give you the value for the INT/WFC B filter and chip A5382-1-7:

```
awe> extinct.value
0.22
```

This extinction can readilly be used.

### Inserting an extinction coefficient into the system

To insert an extinction coefficient for a certain filter/instrument combination into the database, a simple tool is available. This tool is located in the `./awe/astro/toolbox/photometry/` directory of the CVS checkout and is called `ingest_extinction`. To get information about how to use the tool just type:

```
awe $AWEPIPE/Toolbox/photometry/ingest_extinction.py
```

and a doc-string will appear on screen. The tool only accepts input for filters and instruments that are actually present in the database. If a certain filter and/or instrument is not present, the tool refuses to comply.

### Using one observation and known zeropoints

This particular way of deriving the extinction is covered by the `AtmosphericExtinctionZeropoint` class. For instances of this class to work, two dependencies should be set: `polar` and `photoms`. The first dependency expects a list of `PhotSrcCatalog` objects derived from the single observation of the standard field, whereas the second dependency likes to see a list of `PhotometricParameters` objects (see §19.4.2 for more on this class). An example of its use:

```
1. awe> from astro.main.AtmosphericExtinctionZeropoint import \
   ...                   AtmosphericExtinctionZeropoint
2. awe> from astro.main.PhotSrcCatalog import PhotSrcCatalog
3. awe> from astro.main.PhotometricParameters import PhotometricParameters
4. awe> import datetime
5. awe> date_obs = datetime.datetime(2003,2,12,20,15)
6. awe> photcats = (PhotSrcCatalog.date_obs == date_obs) &\
   ...              (PhotSrcCatalog.filter.name == '191') &\
   ...              (PhotSrcCatalog.is_valid == 1)
7. awe> photoms = (PhotometricParameters.timestamp_start <= date_obs) &\
   ...             (PhotometricParameters.timestamp_end > date_obs) &\
   ...             (PhotometricParameters.filter.name == '191') &\
   ...             (PhotometricParameters.is_valid == 1)
8. awe> extinct = AtmosphericExtinctionZeropoint()
9. awe> extinct.polar = list(photcats)
10. awe> extinct.photoms = list(photoms)
11. awe> extinct.make()
```

In lines (1)-(3) the relevant classes are imported, and in steps (6) and (7) the necessary dependencies are retrieved from the database. In lines (8)-(11), an `AtmosphericExtinctionZeropoint` object is instantiated, its dependencies are set, and the *make* method is called.

In the context of deriving a zeropoint, this particular method of deriving an atmospheric extinction seems somewhat out of place. Its primary use lies in the derivation of a nightly extinction report. It is mentioned here for completeness.

#### Using an extinctioncurve and an extinction report

The final, and likely most inflexible, way of deriving the extinction combines the results from a nightly monitoring/extinction report with a standard extinction curve. The class that deals with this situation is the `AtmosphericExtinctionCurve` class. Its use is as follows:

```
1. awe> from astro.main.AtmosphericExtinctionCurve import \
   ...                   AtmosphericExtinctionCurve
2. awe> from astro.main.PhotometricExtinctionReport import \
   ...                   PhotometricExtinctionReport
3. awe> from astro.main.PhotExtinctionCurve import PhotExtinctionCurve
4. awe> from astro.main.Filter import Filter
5. awe> import datetime
6. awe> date = datetime.datetime(2003,2,12)
7. awe> report = (PhotometricExtinctionReport.timestamp_start <= date) &\
   ...            (PhotometricExtinctionReport.timestamp_end > date) &\
   ...            (PhotometricExtinctionReport.is_valid == 1)
8. awe> extcurve = PhotExtinctionCurve.get()
9. awe> filter = (Filter.name == '191')[0]
10. awe> extinct = AtmosphericExtinctionCurve()
11. awe> extinct.report = report[0]
12. awe> extinct.extcurve = extcurve
13. awe> extinct.filter = filter
14. awe> extinct.make()
```

In lines (1)-(4) the relevant classes are imported, and in steps (7)-(9) the necessary dependencies are retrieved from the database (note the `Filter` object). In lines (10)-(14) an `AtmosphericExtinctionCurve` object is instantiated, its dependencies are set and the *make* method is called.

#### Notes on the monitoring/extinction report

The monitoring/extinction report is defined in the *OmegaCAM* photometric calibration plan. This report is derived from a set of data with a very unique feature: every single observation in the data set is observed in FOUR photometric bands SIMULTANEOUSLY. In the photometric pipeline, this particular, dedicated report is represented by the `PhotometricExtinctionReport` class. The *make* method of the `PhotometricExtinctionReport` class is very strict with respect to the `DATE_OBS` of the original standard field observations that go into making it: unless the instrument from which the data to be reduced happens to have the capability to observe FOUR different photometric bands simultanesouly, using this class will be very difficult.

### 19.4.2 Making the zeropoint from the `awe`-prompt

The zeropoint for the night is derived in the final processing step of the photometric pipeline. It is this end result that is used by the image pipeline. The class that represents the final result is `PhotometricParameters`. Instances of this class combine the zeropoint and the atmospheric extinction for the night.

### Using a pre-cooked recipe

Deriving the zeropoint for the night using a pre-cooked recipe is done as follows:

```
awe> from astro.recipes.PhotCalExtractZeropoint import PhotomTask
awe> task = PhotomTask(instrument = 'WFI', raw_filenames = ['r336603_3.fits'])
awe> task.execute()
```

To get detailed and full information about the use of the task, type:

```
awe> help(PhotomTask)
```

which will, for example, show the parameters that can be passed to the constructor of the task.

The `PhotomTask` object will, by default, search the database for a `PhotometricExtionction-Report` object to fullfill the dependency of the atmospheric extinction. If the database does not return a result, the task will try to retrieve an `AtmosphericExtinctionCoefficient` object from the database instead. If both queries fail, the `PhotomTask` will raise an error.

### Using the basic building blocks

Besides using the pre-cooked recipe, it is also possible to use the basic building blocks of the photometric pipeline themselves. This gives much more control over the processing down to the smallest of details. The following code gives an example of what the use of the basic building blocks would look like:

```
1. awe> from astro.main.PhotometricParameters import PhotometricParameters
2. awe> from astro.main.PhotSrcCatalog import PhotSrcCatalog
3. awe> photcat = (PhotSrcCatalog.frame.filename == 'my_standards.fits')[0]
4. awe> photom = PhotometricParameters()
5. awe> photom.photcat = photcat
6. awe> photom.extinct = extinct
7. awe> photom.make()
8. awe> photom.commit()
```

In lines (1)-(2) the relevant classes are imported, and in step (3) the necessary dependency is retrieved from the database. In lines (4)-(7), a `PhotometricParameters` object is instantiated, its dependencies are set, and the *make* method is called. In step (8), the `PhotometricParameters` object is committed to the database. Please note that in step (6) the atmospheric extinction is the one derived previously. The result of calling the *make* method is a zeropoint that is valid for an airmass of 0.00 (the response of your instrument), and an exposure time of 1 second.

### Configuring the photometric CalFile

Any `PhotometricParameters` object has a knob that allows the user to configure the behaviour of its *make* method. To get information about the available configuration options, just type:

```
awe> photom.process_params.info()
```

which will show the available configurable parameters, their meaning, and their default setting.

### Inspecting the results of making the zeropoint

Just as for photometric catalogs, the `PhotometricParameters` object has an *inspect* method that allows the user to see and validate the results of deriving the zeropoint. The method is called thus:

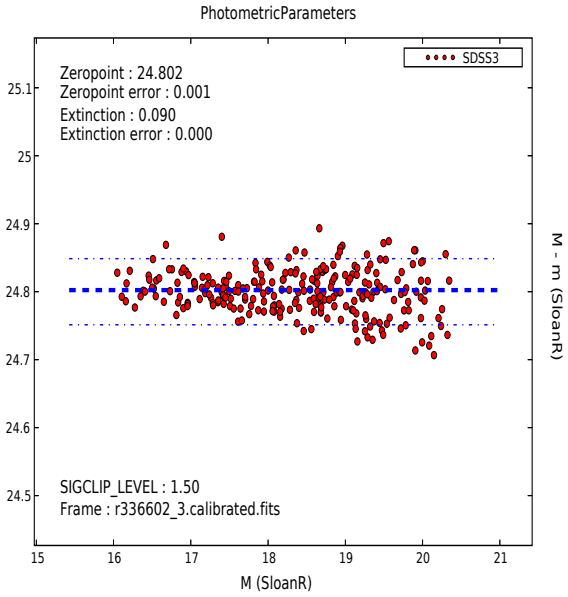



Figure 19.2: The result of invoking the *inspect* method of a `PhotometricParameters` object.

```
awe> photom.inspect()
```

which will result in an output to screen that looks like the one shown in Figure 19.2. The *inspect* plot shows the magnitudes of the individual standard stars as known to the standard star catalog on the x-axis, and their associated (extinction corrected) zeropoints on the y-axis. The blue dot-dashed lines enclose the sources that have been used in deriving the zeropoint, and the thick, blue, dashed line shows the location of the finally derived zeropoint within the distribution of individual zeropoints. The *inspect* method of course also functions on `PhotometricParameters` objects retrieved from the database.

### Inserting a photometric CalFile into the system

To insert a photometric CalFile for a given filter/chip/instrument combination into the database in the case that no photometric calibration can be done, a simple tool is available. The tool is located in the `./awe/astro/toolbox/photometry/` directory of the CVS checkout and is called `ingest_photometrics`. To get information about how to use the tool just type:

```
awe\ $AWEPIPE/astro/toolbox/photometry/ingest_photometrics.py
```

and a doc-string will appear on screen. The tool only accepts input for chips, filters and instruments that are actually present in the database. If a certain chip, filter and/or instrument are not present, the tool refuses to comply.

## 19.5 Photometric Pipeline (4) : Illumination Correction

### 19.5.1 Characterising the illumination variation

#### Using a pre-cooked recipe

Deriving the characterisation of the illumination variation from the awe-prompt using a pre-cooked recipe is done thus :

```
1. awe> from astro.recipes.IlluminationCorrectionVerify import IlluminationVerifyTask
2. awe> task = IlluminationVerifyTask(raw='WFI.1999-06-18T06:02:14.059')
3. awe> task.execute()
```

To get detailed information about the use of the task, type :

```
awe> help(IlluminationVerifyTask)
```

which will, for example, show the parameters that can be passed to the constructor of the task.

#### Using the basic building blocks

A more elaborate way of deriving the characterisation of the illumination variation is by using the basic building blocks of the photometric pipeline themselves. An illumination variation characterisation can be derived from the basic building blocks as follows :

```
 1. awe> from astro.main.IlluminationCorrection import IlluminationCorrection
 2. awe> from astro.main.PhotSrcCatalog import PhotSrcCatalog
 3. awe> from astro.main.RawFitsData import RawFitsData
 4. awe> raw = (RawFitsData.filename == 'WFI.1999-06-18T06:02:14.059.fits')[0]
 5. awe> query = (PhotSrcCatalog.date_obs == raw.DATE_OBS) &\
    ...          (PhotSrcCatalog.is_valid == 1)
 6. awe> len(query)
    8
 7. awe> photcats = [p for p in query]
 8. awe> illum = IlluminationCorrection()
 9. awe> illum.photcats = photcats
10. awe> illum.make()
11. awe> illum.commit()
```

In lines (1)-(3) the relevant classes are imported, and in step (5) the necessary dependency is retrieved from the database. In this case, the dependeny consists of a **list** of photometric catalogs. Note that, in this example, the Python equivalent of a join is used to find these. In lines (8)-(10), an `IlluminationCorrection` object is instantiated, its dependency is set, and the *make* method is called. In step (11), the `IlluminationCorrection` object is committed to the database.

#### Configuring the illumination correction

Any `IlluminationCorrection` object has a knob that allows the user to configure the behaviour of its *make* method. To get information about the available configuration options, just type :

```
awe> illum.process_params.info()
```

which will show the available configurable parameters, their meaning, and their default setting.

### Inspecting the characterisation result

To view the result of making an `IlluminationCorrection` object, simply invoke its *inspect* method:

```
awe> illum.inspect()
```

which will result in an output to screen that looks like the one shown in Figure 19.3. The *inspect* plot shows the result of the overall fit to the zeropoints as a function of their position on the *complete* detector block. Super-imposed on these are some relevant contours. The units on the contours are linear, as is the case for the fit shown on the background. The vertical and horizontal axes give the pixel position on the detector block. The *inspect* method of course also functions on `IlluminationCorrection` objects retrieved from the database.

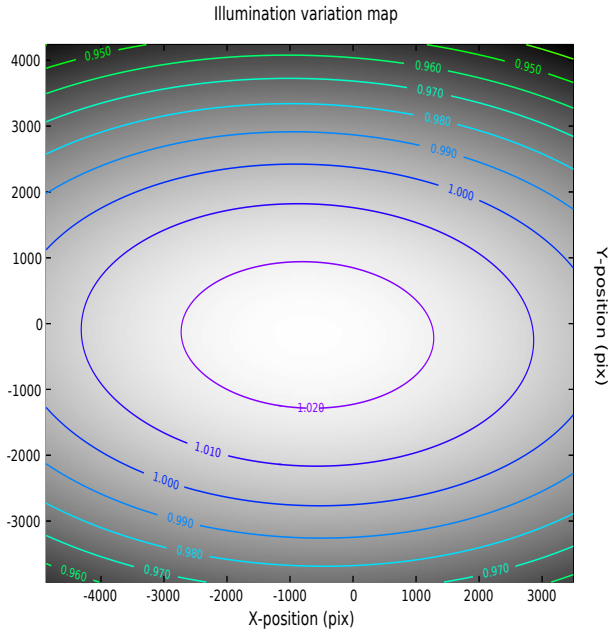



Figure 19.3: The result of invoking the *inspect* method of an `IlluminationCorrection` object.

### Inserting an illumination correction map into the system

To insert an *external* illumination correction map for a certain filter and instrument into the database, a simple tool is available. This tool is located in the `./awe/astro/toolbox/photometry/` directory of the CVS checkout and is called `ingest_illuminationmap`. To get information about how to use the tool just type:

```
awe $AWEPIPE/astro/toolbox/photometry/ingest_illuminationmap.py
```

and a doc-string will appear on screen. The tool only accepts input for a filter/instrument combination that is actually present in the database. If a certain filter or instrument is not present, the tool refuses to comply. The *external* map provided should, of course, adhere to the mathematical structure used by the system.

### 19.5.2 Creating an illumination correction frame

The illumination correction is applied to the *pixel* data in the image pipeline. This is done using an illumination correction frame that is derived from the illumination variation characterisation. This frame is represented in the system by the `IlluminationCorrectionFrame` class. The following sub-sections describe the two ways in which such frames can be produced from the `awe`-prompt.

#### Using a pre-cooked recipe

Deriving an illumination correction frame from the `awe`-prompt using a pre-cooked recipe is done in the following way:

```
1. awe> from astro.recipes.IlluminationCorrection import IlluminationTask
2. awe> task = IlluminationTask(date='2003-02-11', chip='A5382-1-7', \
   ...                          filter='220', instrument='WFC')
3. awe> task.execute()
```

To get detailed information about the use of the task, type:

```
awe> help(IlluminationTask)
```

which will, for example, show the parameters that can be passed to the constructor of the task.

#### Using the basic building blocks

A more elaborate way of making an illumination correction frame is by using the basic building blocks of the photometric pipeline themselves. The most common way of deriving an illumination correction frame using these builing blocks is given here:

```
 1. awe> from astro.main.IlluminationCorrectionFrame import IlluminationCorrectionFrame
 2. awe> from astro.main.IlluminationCorrection import IlluminationCorrection
 3. awe> from astro.main.Chip import Chip
 4. awe> illum = IlluminationCorrection.select(date='2003-02-11', filter='220', \
    ...                                        instrument='WFC')
 5. awe> chip = (Chip.name == 'A5506-4')[0]
 6. awe> illumframe = IlluminationCorrectionFrame()
 7. awe> illumframe.illuminationcorrection = illum
 8. awe> illumframe.chip = chip
 9. awe> illumframe.set_filename()
10. awe> illumframe.make()
11. awe> illumframe.store()
12. awe> illumframe.commit()
```

In lines (1)-(3) the relevant classes are imported, and in steps (4) and (5) the necessary dependencies are retrieved from the database. In lines (6)-(10), an `IlluminationCorrectionFrame` object is instantiated, its dependencies are set, and the *make* method is called. In steps (11) and (12), the `IlluminationCorrectionFrame` object is stored on the fileserver, and then committed to the database.

# Chapter 20

# Calibration: Miscellaneous

## 20.1 HOW-TO Set Timestamps from the awe-prompt

Time stamps and validity of a frame are usually set by the Calibration Time-stamp editor web service CalTS, but they can also be changed at the awe-prompt. After changing these attributes the object can be commited to the database using the recommit method. Notice that only the timestamp_start, timestamp_end and is_valid (super flag) can be (re)commited to the database in this manner, any other attribute that is changed will not be updated in the database.

Python example:

> WARNING: the recommit(s) have been commented out in this example, these will change the timestamps and is_valid of the BiasFrame **in the database**!!

```
     # query for a BiasFrame
awe> qry = BiasFrame.instrument.name == 'WFI'
awe> bias = qry[0]
     # substract one day from the timestamps start
awe> bias.timestamp_start -= datetime.timedelta(1)
     # and set timestamp_end to far future
awe> bias.timestamp_end = datetime.datetime(2010, 1, 1)
     # and commit changes to the database
awe> #bias.recommit()
     # make the BiasFrame invalid
awe> bias.is_valid = 0
awe> #bias.recommit()
     # make the BiasFrame valid again
awe> bias.is_valid = 1
awe> #bias.recommit()
```

## 20.2 HOW-TO Subtract Sky Background

### 20.2.1 Overview

Sky background subtraction is a factor in several places in the reduction process. In most cases SExtractor/SWarp is involved in the algorithm to determine the background image.
Sky background subtraction may be applied first and foremost in **RegriddedFrame**. This is the last opportunity in the data-reduction process before images are coadded into a CoaddedRegriddedFrame. But here is a complete overview of the places where background subtraction plays a role:

- *RegriddedFrame*: see text above
- *ReducedScienceFrame*: when/if an IlluminationCorrectionFrame is applied, a background image is created and the illumination correction is applied to everything but the background. Afterwards the background is added again.
- *SourceList*: when SExtractor is run, default behaviour of SExtractor is to subtract a background.
- In SWarp. Default behaviour of SWarp is to make and subtract a background image. *In* Astro-WISE *this default is changed to no background subtraction.*
- In addition, background images are determined in the algorithms for *HotPixelMap*, *ColdPixelMap*, *CosmicMap* and *SatelliteMap*.

### 20.2.2 Configuring background subtraction

At this point it is not possible to configure the background subtraction in great detail, but it is possible to choose from several options in *RegriddedFrame*.

```
awe> pars = Pars(RegriddedFrame)
awe> pars.show()
RegriddedFrame
 |
 +--process_params
 |   |
 |   +--BACKGROUND_SUBTRACTION_TYPE: 0
 |   +--MAXIMUM_PSF_DIFFERENCE: 0.25
 |
 +--swarpconf
 |   |
 |   +--BACK_DEFAULT: 0.0
 |   +--BACK_FILTERSIZE: 3
 |   +--BACK_SIZE: 128
 |   +--BACK_TYPE: AUTO
etc.
```

Note in particular the process parameter BACKGROUND_SUBTRACTION_TYPE. The possible values for this parameter are:

- 0: Leave background subtraction to SWarp
- 1: Create a background image outside SWarp, which gives better results than SWarp

- 2: Subtract a constant as background, which is determined by iteratively clipping around the median pixel value

See §8.4 for details about how to configure process parameters.

Note that a check is performed to prevent subtracting background both outside and inside SWarp; in other words, when setting the BACKGROUND_SUBTRACTION_TYPE parameter to something other than 0, the RegriddedFrame.swarpconf.SUBTRACT_BACK option has to be 'N', which is the default.

**The background image**

When option 1 is chosen for the BACKGROUND_SUBTRACTION_TYPE, a background image is created. How is this image created? The process consists of these steps:

1) Determine all pixels which are part of sources. This is done by running SExtractor on the image and using the "segmentation" check image.

2) Exclude those pixels and calculate the median of the other pixels.

3) Replace the pixels attributed to sources with the median calculated in the previous step.

4) Run SExtractor to obtain the "background" check image. This is the background image that will be subtracted.

## 20.3 HOW-TO Subwindow statistics

To make it easier to do automatic quality control, statistics in sub-windows are derived for calibration files (including raw frames). On the basis of these statistics various tests can be derived in order to disqualify bad data.

### 20.3.1 How to work with subwindows

Statistics in subwindows are stored in the form of the SubWinStat class. This class has a "frame" dependency pointing to the frame the statistics were derived from. The class hierarchy here may be somewhat counter-intuitive: there is no link from e.g. a RawBiasFrame to a SubWinStat object, instead there is one from the SubWinStat object to the RawBiasFrame. Therefore, to get the subwindow statistics for a particular frame, the following queries should be done:

```
awe> raw = (RawDarkFrame.filename == 'WFI.2001-03-27T21:24:07.546_7.fits')[0]
awe> subwin = (SubWinStat.frame == raw)[0]
```

The SubWinStat class also has a dependency "windows", which contains a list of Imstat objects, which in turn house the statistics such as "mean", "median", "stdev" etc. The Imstat class also contains attributes "x_lower", "x_upper", "y_lower", and "y_upper" defining the region the statistics were derived from. Example:

```
awe> for w in subwin.windows: print w.x_lower, w.y_lower, w.x_upper, w.y_upper
...
49 1 559 512
49 513 559 1024
49 1025 559 1536
49 1537 559 2048
49 2049 559 2560
49 2561 559 3072
49 3073 559 3584
49 3585 559 4096
560 1 1070 512
560 513 1070 1024
560 1025 1070 1536
560 1537 1070 2048
560 2049 1070 2560
560 2561 1070 3072
560 3073 1070 3584
560 3585 1070 4096
1071 1 1581 512
1071 513 1581 1024
1071 1025 1581 1536
1071 1537 1581 2048
1071 2049 1581 2560
1071 2561 1581 3072
1071 3073 1581 3584
1071 3585 1581 4096
1582 1 2092 512
1582 513 2092 1024
1582 1025 2092 1536
1582 1537 2092 2048
```

```
1582 2049 2092 2560
1582 2561 2092 3072
1582 3073 2092 3584
1582 3585 2092 4096
```

There are 4 groups of 8 subwindows here for a total of 32 subwindows, the default.

### 20.3.2 Verify

A number of verify methods are implemented that use subwindows. The actual checks are TBD or improved by experience.

### 20.3.3 Deriving SubWinStat yourself

For RegriddedFrames and CoaddedRegriddedFrames subwindow statistics are not derived by default. It is possible to create subwindow statistics for these images if you want to, however. Example:

```
awe> query = CoaddedRegriddedFrame.filename == 'Sci-GVERDOES-WFI-------#844---C
oadd---Sci-53814.4484972.fits'
awe> frame = query[0]
awe> sub = SubWinStat()
awe> sub.frame = frame
awe> sub.process_params.NUMBER_OF_WINDOWS_X = 10
awe> sub.process_params.NUMBER_OF_WINDOWS_Y = 15
awe> sub.make()
awe> sub.commit()
```

## 20.4 HOW-TO Understand Weights

The weighting scheme in the Astro-WISE system is relatively elaborate as a result of the regridding performed. In the course of the image pipeline (seq631-seq636 as described in the Data Flow System) 3 different weights may be constructed, the first weight is described in the §20.4.1 and the latter two are described in §20.4.2.

### 20.4.1 Science frames and their weight

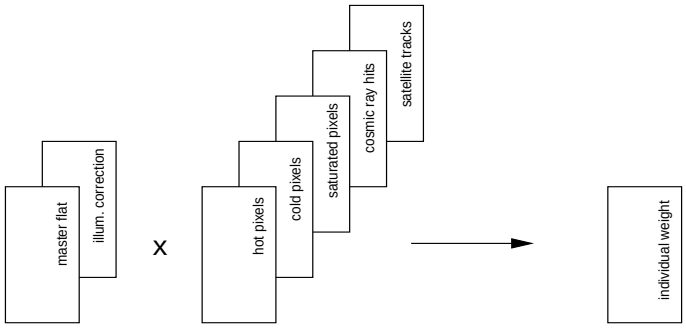



Figure 20.1: Individual weights

The science image is the end product of the pipeline for observations made in Stare mode, where no regridding operation is performed. Each science image has an associated weight.

The (normalized) flat-field describes the response of the telescope and instrument to a source of uniform radiation. In other words, it shows the relative amount of light received as a function of pixel position. This is a measure of the weight of a pixel relative to a pixel at a different position; the produced weights are relative weights.

Some pixels in an image are defect, counting too many or too few photons. These pixels are stored in hot- and cold pixel maps (values 1 for good, 0 for bad). In addition in a raw science image pixels may be unusable due to other causes. Pixel maps are created for saturated pixels, pixels affected by cosmic ray hits and pixels in satellite tracks. Saturated pixels are pixels whose counts exceed a certain threshold. In addition, saturation of a pixel may lead to 'dead' neighbouring pixels, whose counts lie below a lower threshold. Cosmic ray events can be detected using special source detections filters (retina filters), with SExtractor. These are essentially neural networks, trained to recognize cosmic rays, taking a set of neighbouring pixels as input. Satellite tracks can be discovered by a line-detection algorithm such as the Hough transform, where significant signal along a line produces a 'peak' in the transformed image. This peak can be clipped, and transformed back into a pixelmap that masks the track. The weight can therefore be written as:

$$W_{ij} \propto F_{ij} P_{hot} P_{cold} P_{saturated} P_{cosmic} P_{satellite} \tag{20.1}$$

It is possible that an illumination correction (photometric superflat) is used in the pipeline. As a result of stray light received by the detector the background of raw images is not flat. This effect is present both in science images and in flat-fields. In this case, dividing by the flat-field in the image pipeline results in a flat background, but in a non-uniform gain across the detector. In order to correct for this difference an illumination correction frame is made. This image is the result of the difference of the catalog magnitude of standard stars and their magnitude

measured on reduced science frames, as a function of pixel position. The illumination correction can be viewed as a correction on the (incorrect) flat-field. Correctly applying the illumination correction therefore results in a non-flat background.

When an illumination correction is used the weight changes as follows:

$$W_{ij} \propto \frac{F_{ij}}{I_{ij}} P_{hot} P_{cold} P_{saturated} P_{cosmic} P_{satellite} \tag{20.2}$$

In terms of our code base the object oriented class hierarchy for calibrated science images is shown below (specifying only relevant class dependencies):

```
ReducedScienceFrame                  (seq632)
  +-RawScienceFrame                  (seq631)
  +-MasterFlatFrame                  (req546)
  +-BiasFrame                        (req541)
  +-FringeFrame                      (req545)
  +-IlluminationCorrectionFrame      (req548)
  +-WeightFrame                      (seq633)
      +-MasterFlatFrame              (req546)
      +-IlluminationCorrectionFrame  (req548)
      +-ColdPixelMap                 (req535)
          +-DomeFlatFrame            (req535)
      +-HotPixelMap                  (req522)
          +-BiasFrame                (req541)
      +-SaturatedPixelMap
          +-RawScienceFrame
      +-CosmicMap
          +-RawScienceFrame
          +-HotPixelMap
          +-ColdPixelMap
          +-SaturatedPixelMap
          +-MasterFlatFrame
          +-IlluminationCorrectionFrame
      +-SatelliteMap
          +-RawScienceFrame
          +-HotPixelMap
          +-ColdPixelMap
          +-SaturatedPixelMap
          +-MasterFlatFrame
          +-IlluminationCorrectionFrame
```

### 20.4.2 Weights created by SWarp

Image coaddition is divided in two parts, resampling and combination. Image resampling addresses the problem that two independent observations of the same area of sky will, in general, result in images whose coordinate systems are different. Projection of these coordinate systems to a new coordinate system requires a mapping $x, y \Rightarrow \alpha, \delta \Rightarrow x', y'$. Since the area of sky covered by an input pixel will in general not map directly to an area of sky covered by a single output pixel, some sort of interpolation is required. Unfortunately interpolation will inevitably result in aliasing artifacts, hence, a careful choice of interpolation kernel is required.

Once all input images and their weights have been resampled onto the grid specified by the coordinate system of the coadded image, it is straightforward to compute the coadded image

and its weight. Given resampled input images $1 \leq i \leq N$ entering co-addition, one can define for each pixel $j$:

- the local uncalibrated flux $f_{ij} = \overline{f_{ij}} + \Delta f_{ij}$, where $\overline{f_{ij}}$ is the sky background, and $\Delta f_{ij}$ the contribution from celestial sources,
- the local uncalibrated variance $\sigma^2_{ij} = \overline{\sigma^2_{ij}} + \Delta\sigma^2_{ij}$, where $\overline{\sigma^2_{ij}}$ is background noise and $\Delta\sigma^2_{ij}$ the photon shot noise of celestial sources,
- the local, normalized weight $w_{ij}$,
- the electronic gain of the CCD $g_i$,
- the relative flux scaling factor $p_i$, deduced from the photometric solution, with $p_i\Delta f_{ij} = p_l\Delta f_{lj}$ for all $i, l, j$,
- the weight scaling factor $k_i$.

Optimal weighting is obtained using

$$k_i w_{ij} = \frac{1}{p_i^2\overline{\sigma^2_{ij}}},$$

leading to a co-added flux (assuming the background of the input images has been subtracted),

$$f_j = \frac{\sum_i k_i w_{ij} p_i \Delta f_{ij}}{\sum_i k_i w_{ij}},$$

and variance

$$\sigma^2_j = \frac{\sum_i k_i^2 w_{ij}^2 p_i^2 \sigma^2_{ij}}{(\sum_i k_i w_{ij})^2},$$

The software package SWarp, developed by E. Bertin (IAP) for Terapix provides a set of regridding and co-addition algorithms specifically optimized to handle large area CCD mosaics.

### Regridded frames and their weights

Whenever a calibrated science image is created in Dither mode (seq635) it is resampled to one of a number of pre-defined field centers (part of seq636). This affects both the science and weight images.

The default configuration of Swarp for making RegriddedFrames and their weights results in weight frames that are the resampled weight frame of the ReducedScienceFrame multiplied by $1/\sigma^2$, where $\sigma^2$ is the variance of the local background in units ADU$^2$.

The class hierarchy for regridded images is as follows:

```
RegriddedFrame
  +-AstrometricParameters
  +-PhotometricParameters
  +-ReducedScienceFrame
      +-(see section 1)
  +-WeightFrame           (produced by SWarp)
```

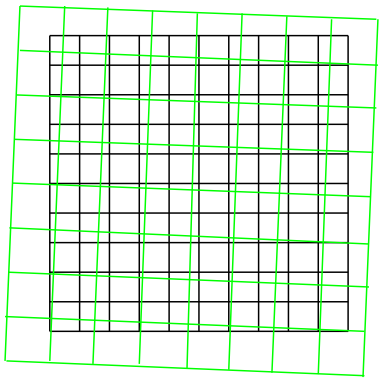


Figure 20.2: Regridded weights

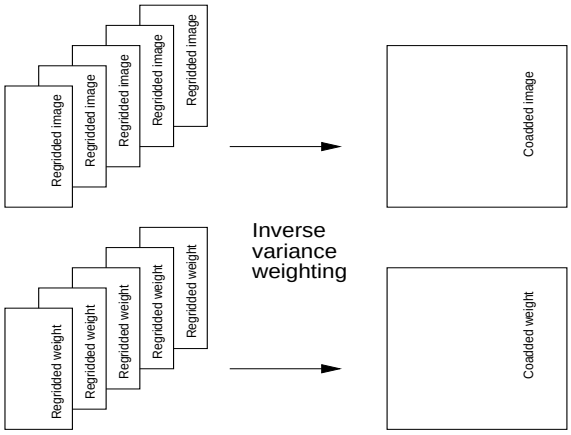



Figure 20.3: Coadded weights

### Coadded frames and their weights

When a collection of the regridded images described in the previous section are coadded a new weight is constructed by adding the (regridded) weights. In this step variance weighting is used to get a total weight.

The default configuration of Swarp for making CoaddedRegriddedFrames and their weights results in weight frames that contain the summation of the weights of the RegriddedFrames scaled by $1/FLXSCALE^2$, where $FLXSCALE^2$ is conversion factor from counts to the units of the CoaddedRegridded science frame.[1] Thus the weightframe is $1/\sigma^2$, where $\sigma^2$ is the variance of the local background with $\sigma$ in the same units as the CoaddedRegriddedFrame science frame.

The class hierarchy for coadded frames is as follows:

```
CoaddedRegriddedFrame
  +-RegriddedFrame(s)
      +-(see section 2.1)
```

[1] TBD: is it the sum of each weightframe of a RegriddedFrame scaled with its own $FLXSCALE$ (i.e., RegriddedFrame.FLXSCALE)?

```
+-WeightFrame       (produced by SWarp)
```

### 20.4.3 Weights in quality control

Three methods are defined for most calibration files, subdividing the quality control. The *verify* method is used to provide internal checks, often this takes the form of sanity checks on parameters of e.g. a BiasFrame. The *compare* method compares aspects of e.g. a BiasFrame (standard deviation for example) with that of a previous instance. *Inspect* is intended to show plots or images so that after an interactive analysis a decision can be made on the quality of the object.

In particular, statistics in subwindows of all images are used. While determining these statistics it is possible to provide a pixelmap of acceptable pixels. Below is a table that shows which weights/pixelmaps are (can be) used.

| Class | HotPixelMap | ColdPixelMap | SaturatedPixelMap | CosmicMap | SatelliteMap | Individual Weight |
|---|---|---|---|---|---|---|
| RawFrame | - | - | - | - | - | - |
| BiasFrame | - | - | - | - | - | - |
| DomeFlatFrame | + | + | - | - | - | - |
| TwilightFlatFrame | + | + | - | - | - | - |
| MasterFlatFrame | + | + | - | - | - | - |
| FringeFrame | + | + | - | - | - | - |
| ReducedScienceFrame | + | + | + | + | + | + |
| RegriddedFrame | - | - | - | - | - | + |

Table 20.1: Table of weights/pixelmaps and their application in quality control

# Chapter 21

# Image Pipeline

## 21.1 HOW-TO Image Pipeline: overview

The image pipeline is the part of the system that processes science data. In this section, the image pipeline is discussed. A summary is given of the atomic tasks that make up the pipeline, and their place therein is described. Also given is an overview of how the image pipeline as a whole can be steered through the DPU interface. For the interface and use of every individual task, please read the corresponding HOW-TO.

### 21.1.1 The atomic tasks and their context

The atomic tasks that make up the image pipeline are summarized in Table 21.1, together with their role in the system and the identifier under which these are known to the DPU interface.

The sequence of tasks that make up the image pipeline is shown in Figure 21.1. In this figure, each one of the individual tasks is represented by one box. The arrows indicate the flow of the pipeline, and the shaded part shows a particular (optional) branch therein.

Contrary to the calibration pipelines, the start and end points of the image pipeline are flexible; one can choose to only de-bias and flatfield the data, but it is also possible to run the full pipeline including the global astrometry. The exception is the photometric calibration which has to be done separately. This requires performing Reduce, Astrometry and Photometry on a raw science frame of a standard star field. The resulting photometric parameters are then applied to the science image during the Regrid process.

### 21.1.2 Astrometry in the image pipeline

The astrometry in the image pipeline can be done in one of two ways. The first, and default, way is the traditional single-chip astrometry. This path through the image pipeline is shown in black in Figure 21.1. A more elaborate way of deriving the astrometric calibration is by combining the data of overlapping images to get a more accurate result for the single chips. This particular branch in the image pipeline, and the various tasks performed therein, is shown in light grey. Note that this branch augments the image pipeline, it does not supersede it.

### 21.1.3 Running the image pipeline with the DPU

There are three ways in which the image pipeline can be run through the DPU interface: (1) one task at a time, (2) using a pre-defined sequence *without* global astrometry, (3) using a pre-defined sequence *with* global astrometry.

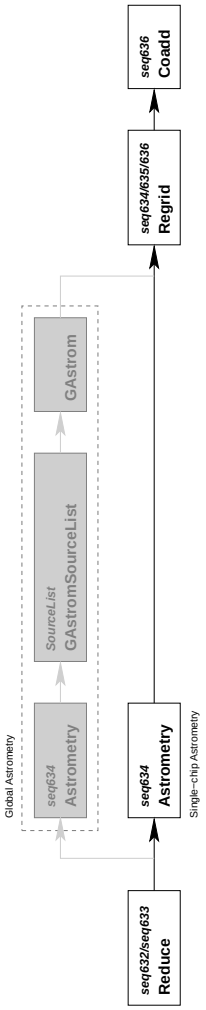



Figure 21.1: The order and flow of the atomic processing steps in the image processing part of the system. The lightly shaded part shows the (optional) branch into the global astrometry.

Table 21.1: The various atomic processing steps that make up the image pipeline and the identifiers through which these can be selected by the user from the DPU interface.

| Processing step | Purpose | DPU identifier |
|---|---|---|
| `Reduce` | De-biasing and flatfielding | `Reduce` |
| `Astrometry` | Single-chip astrometry | `Astrometry` |
| `Photometry` | Photometric calibration | `Photom` |
| `Regrid` | Regridding pixel data | `Regrid` |
| `Coadd`[1] | Coadding pixel data | `Coadd` |
| | | |
| `GAstromSourceList` | Making a sourcelist for global astrometry | `GAstromSL` |
| `GAstrom`[1] | Deriving the global astrometry | `GAstrom` |

[1]This processing step is performed on a single node

The first way of running the image pipeline is simple: provide the DPU interface with one task identifier (see Table 21.1) and the appropriate query parameters. Example:

```
awe> dpu.run('Reduce', i='WFI',
             raw_filenames=['WFI.2000-04-29T00:40:57.671_1.fits'])
```

which will result in only de-biasing and flatfielding the selected frame. Another example:

```
awe> dpu.run('Reduce', i='WFI',
             raw_filenames=['WFI.2000-04-28T23:06:49.353_1.fits'])
```

which will result in determining photometric parameters, such as zeropoint and extinction coefficient, based on the standard star field frame 'WFI.2000-04-28T23:06:49.353_1.fits'.

Besides this simple way of steering the image pipeline, it is also possible to provide the DPU interface with an identifier that selects a whole pre-defined sequence of tasks. The present set of pre-defined sequences (with and without global astrometry) is given in Table 21.2. All these sequences define 'sane' image pipelines that are internally consistent.

The identifiers given in Table 21.2 have the following structure:

```
(starting point of the pipeline)>(end point of the pipeline),
```

which will yield a traditional image pipeline with only single-chip astrometry. If one wants to include the global astrometry in the reduction, the structure is modified as follows:

```
(starting point of the pipeline)>GAstrometry>(end point of the pipeline),
```

where the extra `GAstrometry` clause explicitly instructs the image pipeline to switch to the global astrometry branch (the grey branch in Fig. 21.1).

### Examples of using pre-defined sequences of tasks

Run the complete image pipeline all the way from de-biasing and flatfielding up to and including co-addition, but without global astrometry:

```
awe> dpu.run('Reduce>Coadd', instrument='WFI', date='2000-04-28',
             filter='#842', object='PG1525_B', commit=1)
```

Table 21.2: The pre-defined sequences of image pipeline processing steps and their identifiers for the DPU interface. Note the way in which the global astrometry is enabled.

| Processing steps | DPU identifier |
|---|---|
| `Reduce + Astrometry` | `Reduce>Astrometry` |
| `Reduce + Astrometry + Regrid` | `Reduce>Regrid` |
| `Reduce + Astrometry + Regrid + Coadd` | `Reduce>Coadd` |
| `Reduce + (GAstrometry) + Regrid` | `Reduce>GAstrometry>Regrid` |
| `Reduce + (GAstrometry) + Regrid + Coadd` | `Reduce>GAstrometry>Coadd` |
| `Regrid + Coadd` | `Regrid>Coadd` |

`(GAstrometry) = Astrometry+GAstromSourceList+GAstrom`

Now do the same, but with global astrometry enabled (using short options this time):

```
awe> dpu.run('Reduce>GAstrometry>Coadd', i='WFI', d='2000-04-28', f='#842',
             o='PG1525_B', C=1)
```

Note that this particular sequence contains two processing steps that will be done on a single node, using results from the previous processing step.

Only de-biasing, flatfielding, and single-chip astrometry (for e.g. photometric standard fields):

```
awe> dpu.run('Reduce>Astrometry', i='WFI',
             raw_filenames=['WFI.2000-04-28T23:06:49.353_1.fits'])
```

Only de-biasing, flatfielding, single-chip astrometry, and regridding:

```
awe> dpu.run('Reduce>Regrid', i='WFI',
             raw_filenames=['WFI.2000-04-28T23:06:49.353_1.fits'])
```

## 21.2 HOW-TO Create a `ReducedScienceFrame`

When an instrument is added to the Astro-WISE database environment, it is initialised (bootstrapped) with calibration frames that are valid "forever". This allows one to always be able to reduce `RawScienceFrames` albeit with less than optimal results.

> NOTE: In many cases, however, calibration files taken near to the date-obs of the science data can exist and be valid for that data. This may occur if another observer's data has been reduced first. Although these calibration frames may give optimal results, it is always possible to create new calibration frames specific to the data being processed. The recipes will always choose the most recent calibration frames by default.

### 21.2.1 Making `ReducedScienceFrames` using the DPU

The most ideal way to process data in `AWE` is to process it with the DPU (see §7.7).

The DPU interface is built into `AWE` so that its use is simple and transparent. A basic DPU usage looks like:

```
awe> dpu.run('task_name', option1='', option2='', . . .)
```

For the `ReduceTask`, the DPU command would look more like:

```
awe> dpu.run('Reduce', d='2001-01-22', i='WFI', f='#841', o='AXAF')
```

or

```
awe> dpu.run('Reduce', i='WFI', raw_filenames=['WFI.2000-01-01T08:57:15.410_3.fits'])
```

The `ReduceTask` options via the DPU are as follows:

- `i` (instrument, mandatory): string
- `d` (date): string of the form CCYY-MM-DD
- `f` (filter): string
- `raw_filenames` (raw_filenames): list of strings
- `o` (object): string with possible wildcards * and ?
- `oc` (overscan): integer=0..10 (optional, default=6)
- `C` (commit): integer=0..1 (optional, default=0)

The first example shows how to run the DPU `ReduceTask` using date, filter, and object information. This allows one to process all data for a given object on a specific day taken in a certain filter, for *all* CCDs of the mosaic. The second example shows how to run the DPU `ReduceTask` using specific data with known raw filenames.

The options can be used in any order and can be omitted (except '`i`'), but the likelihood of locating required data depends on relaying a minimum of information as shown in the examples.

### 21.2.2 Making a ReducedScienceFrame using the ReduceTask

Although the ideal method to process RawScienceFrames is the DPU, this task can be performed on a per chip basis. The ReduceTask is the main part of the recipe $AWEPIPE/astro/recipes/Reduce.py. Two examples of the syntax for the ReduceTask are given below:

```
awe> from astro.recipes.Reduce import ReduceTask
awe> task = ReduceTask(date='2001-01-22', chip='ccd51', filter='#841',
...                    object='AXAF', i='WFI')
awe> task.execute()
```

or

```
awe> task = ReduceTask(raw_filenames=['WFI.2000-01-01T08:57:15.410_3.fits'])
awe> task.execute()
```

The ReduceTask options are as follows:

- date: string of the form CCYY-MM-DD
- chip: string
- filter: string
- raw_filenames: list of strings
- object: string with possible wildcards * and ?
- overscan: integer=0..10 (optional, default=6)
- commit: integer=0..1 (optional, default=0)

The first example shows how to run the ReduceTask using date, chip, filter, and object information. This allows one to process all data for a given object on a specific day taken in a certain filter, and only one CCD of the mosaic. The second example shows how to run the ReduceTask using specific data with known raw filenames.

The options can be used in any order and can be omitted, but the likelihood of locating required data depends on relaying a minimum of information as shown in the examples.

### 21.2.3 Making a ReducedScienceFrame using the basic building blocks

The third, and most powerful, way of creating a ReducedScienceFrame is by using the basic building blocks of the system directly from the awe-prompt. This method is used if one wants to manipulate the system down to the nitty-gritty details. An example of how a ReducedScienceFrame is made from the awe-prompt in this fashion is given below:

```
1. awe> from astro.main.RawFrame import RawScienceFrame
2. awe> from astro.main.ReducedScienceFrame import ReducedScienceFrame
3. awe> from astro.main.MasterFlatFrame import MasterFlatFrame
4. awe> from astro.main.ColdPixelMap import ColdPixelMap
5. awe> from astro.main.HotPixelMap import HotPixelMap
6. awe> from astro.main.BiasFrame import BiasFrame

7. awe> raw  = (RawScienceFrame.filename == 'WFI.2000-01-01T08:57:15.410_3.fi
   ...  ts')[0]
```

```
 8. awe> hot  = HotPixelMap.select_for_raw(raw)
 9. awe> cold = ColdPixelMap.select_for_raw(raw)
10. awe> flat = MasterFlatFrame.select_for_raw(raw)
11. awe> bias = BiasFrame.select_for_raw(raw)

12. awe> reduced = ReducedScienceFrame()
14. awe> reduced.raw  = raw
15. awe> reduced.hot  = hot
16. awe> reduced.cold = cold
17. awe> reduced.bias = bias
18. awe> reduced.flat = flat
19. awe> reduced.raw.retrieve()
20. awe> reduced.hot.retrieve()
21. awe> reduced.cold.retrieve()
22. awe> reduced.bias.retrieve()
23. awe> reduced.flat.retrieve()
24. awe> reduced.set_filename()

25. awe> reduced.make()
26. awe> reduced.store()
27. awe> reduced.weight.store()
28. awe> reduced.commit()
```

In steps (1)-(6), the minimum number of classes relevant for this processing step are imported. Other classes that could be used here are `FringeFrame` and `IlluminationCorrectionFrame`. In steps (7)-(11), the database is queried for the necessary dependencies. In step (7), the `RawScienceFrame` to be processed is retrieved, and in steps (8)-(11) the calibration files to use. In steps (12) and (13), the `ReducedScienceFrame` object is instantiated, and one of its process configuration parameters is set. In steps (14)-(24), the dependencies of the `ReducedScienceFrame` are set, and the pixel-data underlying the calibration files are retrieved from the fileserver. In step (24), the filename of the `ReducedScienceFrame` to be is created and set. Finally, in steps (25)-(28), the `ReducedScienceFrame` is made, its pixel-data is stored, and the object itself is committed to the database. Note that in step (27), the `WeightFrame` associated with the `ReducedScienceFrame` object is also stored; the weights are a very important by-product of making the `ReducedScienceFrame` that should never be forgotten.

### 21.2.4 Output Logs

In both the DPU and non-DPU processing methods, log files are created (of the form CCYYMMDD_hhmmssss.log). For both cases, logs are printed to both the screen and the log file. For the non-DPU method, nothing needs to be done to retrieve the logs, they just come as they are created. For the DPU method, the processing is done remotely, and they have to be retrieved manually. There is a simple mechanism with which to do this, a method of the DPU object called `get_logs()`. When this method is invoked, all the finished jobs are retrieved one at a time[1], displayed to screen, and printed to the log file. An example of this mechanism is given below:

```
awe> dpu.jobids()
[313L]
awe> dpu.get_logs()
```

[1] The `get_logs()` method also can be invoked with a job ID (or a list of job IDs) as an argument.

```
<log printed to file and screen>

awe> dpu.jobids()
[]
```

For more complete information on the functionality of the DPU, see §7.7.

### 21.2.5 Viewing the results

#### Query for and retrieve ReducedScienceFrames

When one of the above commands are run, it should generally be run first without the `commit` switch set (`C`=0, the default) so as to test whether all steps were successful. Once the logs have been examined and no serious problems found, the command can be run with the `commit` switch set to `C`=1.

If all went well, the `RawScienceFrame` has had all the calibration frames available applied to it to create the `ReducedScienceFrame`. Now this frame can be retrieved from the data server for inspection. This is done by first querying the database for the `ReducedScienceFrames` just created:

```
awe> q = ReducedScienceFrame.raw.filename == 'WFI.2000-01-01T08:57:15.410_3.fits'
awe> len(q)
1
awe> q = ReducedScienceFrame.raw.filename.like('WFI.2000-01-01T08:57:15.410*')
awe> len(q)
8
```

Once the desired `ReducedScienceFrame` is in the query, it is a simple matter to look at the images statistics and retrieve it:

```
awe> for f in q:
...      print f.imstat.mean, f.imstat.median, f.imstat.stdev
111.557922989 111.463745117 8.56794497829
107.94240659 107.812011719 8.38193208213
106.126105882 106.035461426 7.48044620812
97.2398969938 97.0784301758 8.0901417073
98.4600191901 98.3441619873 7.53172611373
91.9374131475 91.8213806152 7.21070922762
118.980466828 118.880493164 8.97856755639
106.038520679 105.926620483 8.20778339642
awe> for f in q:
...      f.retrieve()
15:37:34 - Retrieving Sci-USER-WFI-------#877-ccd54-----Sci-53664.4795766.fits
15:37:37 - Retrieving Sci-USER-WFI-------#877-ccd55-----Sci-53664.4796525.fits
15:37:40 - Retrieving Sci-USER-WFI-------#877-ccd52-----Sci-53664.4797027.fits
15:37:43 - Retrieving Sci-USER-WFI-------#877-ccd53-----Sci-53664.4797485.fits
15:37:46 - Retrieving Sci-USER-WFI-------#877-ccd57-----Sci-53664.4801416.fits
15:37:49 - Retrieving Sci-USER-WFI-------#877-ccd56-----Sci-53664.4801463.fits
15:37:52 - Retrieving Sci-USER-WFI-------#877-ccd50-----Sci-53664.4803221.fits
15:37:55 - Retrieving Sci-USER-WFI-------#877-ccd51-----Sci-53664.4803202.fits
```

### Display ReducedScienceFrames

Once the images are retrieved, they can be viewed in a number of ways. There currently exists no efficient built-in mechanism to view images in `AWE`, so external viewers viewers should be used. One external viewer in common use is ESO's SkyCat tool. The syntax used to view one of the images (the first in the query in this case) with SkyCat is:

```
awe> os.system('skycat %s' % (q[0].filename))
```

To view the entire mosaic with SkyCat, the mosaic must first be converted into a multi-extension FITS (MEF) file. The entire process is illustrated below:

```
awe> newname = 'Sci-USER-WFI-#877-53664.5.fits'
awe> mef = Image(newname)
awe> mef.frames = q
awe> mef.make()
awe> os.system('skycat %s' % (newname))
```

Once SkyCat appears, click the "Display as one Image" button to see the entire image. See §23.3 for more detailed information.

### Ask ReducedScienceFrames about themselves

The database objects that are created (`ReducedScienceFrames` in this example) hold information about their history. This information can be found by inspecting various attributes of the object. For example:

```
awe> q = (ReducedScienceFrame.filename != '') &
...      (ReducedScienceFrame.instrument.name == 'WFI')
awe> len(q)
37965
awe> frame = q[0]
awe> frame.filename
'Sci-WVRIEND-WFI-------#845-ccd56-----Sci-53628.6100091.fits'
awe> frame.raw.filename
'WFI.2000-12-20T01:21:43.805_7.fits'
awe> frame.bias.filename
'2000-04-26cal541_ccd56.fits'
awe> frame.flat.filename
'2000-04-21cal546-#845_ccd56.fits'
```

## 21.3 HOW-TO Derive an Astrometric Solution

Astrometric solutions in `AWE` can be done in two different ways: on a chip-by-chip basis and globally (multiple chips at once). For the single-chip solution, an `AstrometricParametersTask` is used, and for the multi-chip solution, both `GAstromSourceListTask` and `GAstromTask` are used. See the appropriate HOW-TO section for instructions on how to do these:

- `AstrometricParametersTask` (§18.1)
- `GAstromSourceListTask` (§18.2)
- `GAstromTask` (§18.3)

## 21.4 HOW-TO Create a `RegriddedFrame`

### 21.4.1 Making `RegriddedFrames` using the DPU

For the `RegridTask`, the DPU command would look something like:

```
awe> dpu.run('Regrid', d='2001-01-22', i='WFI', f='#841', o='AXAF')
```

The `RegridTask` options via the DPU are as follows:

- `i` (instrument, mandatory): string
- `d` (date): string of the form CCYY-MM-DD
- `f` (filter): string
- `o` (object): string, possibly using wildcards * and/or ?
- `raw` (raw_filenames): list, a list of filenames of raw science frames that have reduced science frames
- `red` (red_filenames): list, a list of filenames of reduced science frames
- `gra` (grid_ra): float, right ascension of grid target
- `gdec` (grid_dec): float, declination of grid target
- `gps` (grid_pixelscale): float, requested pixelscale (
- `C` (commit): integer=0..1 (optional, default=0)

This example shows how to run the DPU `RegridTask` using date, filter, and object information. This allows one to process all data for a given object on a specific day taken in a certain filter, for *all* CCDs of the mosaic.

The options can be used in any order and can be omitted (except '`i`'), but the likelihood of locating required data depends on relaying a minimum of information as shown in the examples.

The GridTarget is the RegriddedFrame(s) its regrid center. The RA and DEC of the GridTarget are determined by a system that divides the sky in a set of pre-determined plates. Note: for large fields of view like OMEGACAM, the determined grid target ra and dec can be those of different place. This will cause errors in a later step called CoaddTask. However ,it is possible for the user to force all regridded frames to the same grid target. For the DPU the following two steps should be taken:

1. Determine grid target of one frame For instance do:

   ```
   awe> dpu.run(red_filenames=[<filename>])
   ```

   The logs of this task step show the grid target ra and dec of the frame.

2. Apply the grid target of step one to all other frames The logs of the first step show the grid target ra and dec, these should now be included as arguments, like:

   ```
   awe> dpu.run(red_filenames=[<all filenames>],gra=<grid target ra as determined from step 1>,gd
   ```

### 21.4.2 Making a `RegriddedFrame` using the `RegridTask`

Although the ideal method to create `RegriddedFrames` is the DPU, this task can be performed in on a per chip basis. The `RegridTask` is the main part of the recipe `$AWEPIPE/astro/recipes/Regrid.py`. An example of the syntax for the `RegridTask` is given below:

```
awe> reg = RegridTask(date='2001-01-22', chip='ccd51', filter='#841',
...                   object='AXAF')
awe> reg.execute()
```

The `RegridTask` options are as follows:

- `date`: string of the form CCYY-MM-DD
- `chip`: string
- `filter`: string
- `object`: string, possibly using wildcards * and/or ?
- `raw` (raw_filenames): list, a list of raw science frames that have reduced science frames
- `red` (red_filenames): list, a list of reduced science frames
- `gra` (grid_ra): float, right ascension of grid target
- `gdec` (grid_dec): float, declination of grid target
- `gps` (grid_pixelscale): float, requested pixelscale (
- `fsg` (force_single_gridtarget): force the grid target to be similar for all input frames
- `commit`: integer=0..1 (optional, default=0)

This example shows how to run the `RegridTask` using date, chip, filter, and object information. This allows one to process all `ReducedScienceFrames` for a given object on a specific day taken in a certain filter, and with only one CCD of the mosaic.

The options can be used in any order and can be omitted, but the likelihood of locating required data depends on relaying a minimum of information as shown in the examples.

The GridTarget is the RegriddedFrame(s) its regrid center. The RA and DEC of the GridTarget are determined by a system that divides the sky in a set of pre-determined plates. Note: for large fields of view like OMEGACAM, the determined grid target ra and dec can be those of different place. This will cause errors in a later step called CoaddTask. However ,it is possible for the user to force all regridded frames to the same grid target. The argument force_single_gridtarget has to be includeded, like for instance

```
awe> reg = RegridTask(red_filenames=[<filenames>], fsg=True)
```

### 21.4.3 Making a `RegriddedFrame` using the basic building blocks

The third, and most powerful, way of creating a `RegriddedFrame` is by using the basic building blocks of the system directly from the `awe`-prompt. This method is used if one wants to manipulate the system down to the nitty-gritty details. An example of how a `RegriddedFrame` is made from the `awe`-prompt in this fashion is given below :

```
 1. awe> from astro.main.RegriddedFrame import RegriddedFrame
 2  awe> from astro.main.RegriddedFrame import GridTarget
 3. awe> from astro.main.ReducedScienceFrame import ReducedScienceFrame
 4. awe> from astro.main.AstrometricParameters import AstrometricParameters
 5. awe> from astro.main.PhotometricParameters import PhotometricParameters
 6. awe> from astro.main.GainLinearity import GainLinearity

 7. awe> reduced       = (ReducedScienceFrame.filename == 'Sci-GVERDOES-WFI---
    ...  ----#842-ccd55-Red---Sci-53811.3816955.fits')[0]
 8. awe> astrom_query  = (AstrometricParameters.reduced == reduced) &\
    ...                  (AstrometricParameters.quality_flags == 0) &\
    ...                  (AstrometricParameters.is_valid == 1)
 9. awe> astrom_params = astrom_query.max('creation_date')
10. awe> photom_params = PhotometricParameters.select_for_raw(reduced.raw)
11. awe> gain          = GainLinearity.select_for_raw(reduced.raw)

12. awe> from astro.util.PlateSystem import PlateSystem
13. awe> platesystem = PlateSystem()
14. awe> ra, dec = platesystem.get_field_centre(reduced.astrom.CRVAL1,
    ...                                         reduced.astrom.CRVAL2)
15. awe> grid_target = GridTarget()
16. awe> grid_target.RA = ra
17. awe> grid_target.DEC = dec
18. awe> grid_target.pixelscale = 0.20

19. awe> regrid = RegriddedFrame()
20. awe> regrid.process_params.MAXIMUM_PSF_DIFFERENCE = 0.30
21. awe> regrid.swarpconf.SUBTRACT_BACK = 'Y'
22. awe> regrid.swarpconf.BACK_SIZE = 256

23. awe> regrid.reduced       = reduced
24. awe> regrid.grid_target   = grid_target
25. awe> regrid.astrom_params = astrom_params
26. awe> regrid.photom_params = photom_params
27. awe> regrid.gain          = gain
28. awe> regrid.reduced.retrieve()
29. awe> regrid.reduced.weight.retrieve()
30. awe> regrid.set_filename()

31. awe> regrid.make()
32. awe> regrid.store()
33. awe> regrid.weight.store()
34. awe> regrid.commit()
```

In steps (1)-(6), the classes relevant for this processing step are imported. In steps (7)-(11), the database is queried for the necessary dependencies. In step (7), the `ReducedScienceFrame` to be regridded is retrieved, and in steps (8)-(11) the calibration data to use. The steps (12)-(18) are unique for making a `RegriddedFrame`; the `GridTarget` is a non-`ProcessTarget` object that provides the `RegriddedFrame` its regrid center. The RA and DEC of the `GridTarget` are determined by a system that divides the sky in a set of pre-determined plates (hence the steps (12)-(14)). In steps (19)-(22), the `RegriddedFrame` object is instantiated, and a few of its process

configuration parameters are set. In steps (23)-(30), the dependencies of the `RegriddedFrame` are set, and the pixel-data of the input `ReducedScienceFrame` and its associated weight are retrieved from the fileserver. In step (30), the filename of the `RegriddedFrame` to be is created and set. Finally, in steps (31)-(34), the `RegriddedFrame` is made, its pixel-data is stored, and the object itself is committed to the database. Note that in step (33), the `WeightFrame` associated with the `RegriddedFrame` object is also stored; the weights are a very important by-product of making the `RegriddedFrame` that should never be forgotten.

## 21.5 HOW-TO Create a CoaddedRegriddedFrame

Several RegriddedFrames created from ReducedScienceFrames (see §21.2 or §21.4) can be coadded to form a deeper image with a substantial reduction of chip defects and divisions (a CoaddedRegriddedFrame).

> NOTE: At this time, the CoaddTask does not allow frames from multiple instruments to be coadded. It should be possible to do this provided all the RegriddedFrames were regridded to the same grid target (R.A., DEC., and pixel scale[2]), but this operation is currently not supported.

### 21.5.1 DPU Method

At the moment, the CoaddTask is a purely serial operation which takes place on only one node of the DPU. Some possible reasons to use the DPU for this task are because the machines on the DPU are much faster than the local machine, or because no local machine is available.

For the CoaddTask, the DPU command would look something like:

```
awe> dpu.run('Coadd', d='2000-01-01', i='WFI', f='#843', o='Science1_?-*')
```

The CoaddTask options via the DPU are as follows:

- i (instrument, mandatory): string
- f (filter, mandatory): string
- d (date): string of the form CCYY-MM-DD
- o (object): string or "regular expression"
- C (commit): integer=0..1 (optional, default=0)

This example shows how to run the DPU CoaddTask using date, filter, and object information. This allows one to process all data for a given object on a specific day taken in a certain filter, for *all* CCDs of the mosaic.

The options can be used in any order and can be omitted (except 'i'), but the likelihood of locating required data depends on relaying a minimum of information as shown in the examples.

### 21.5.2 Non-DPU Method

This task can also be performed in on a per chip basis. The CoaddTask is main part of the recipe $AWEPIPE/astro/recipes/Coadd.py. This is most basic front-end for creating CoaddedRegriddedFrames in AWE. An examples of the syntax for the CoaddTask is given below:

```
awe> coa = CoaddTask(instrument='WFI', date='2000-01-01', chip='ccd50',
...                  filter='#843', object='Science1_?-*')
awe> coa.execute()
```

The CoaddTask options are as follows:

- instrument, mandatory: string
- filter, mandatory: string

[2] Current default pixel scale is 0.200 *arcsec/pixel*

- `date`: string of the form CCYY-MM-DD
- `chip`: string
- `object`: string or "regular expression"
- `commit`: integer=0..1 (optional, default=0)

This example shows how to run the `CoaddTask` using date, chip, filter, and object information. This allows one to process all data for a given object on a specific day taken in a certain filter, and only one CCD of the mosaic.

The options can be used in any order and can be omitted, but the likelihood of locating required data depends on relaying a minimum of information as shown in the examples.

### 21.5.3 Coadd algorithm

The algorithm behind the coaddition is described via the coaddition of two RegriddedFrames which have the (object)name reg1 and reg2. The value $f_{out}$ of a pixel in the resulting CoaddedRegriddedFrame (which we give object name coad) is computed as follows:

$$f_{out} = \Sigma_i(w_i * FLXSCALE_i * f_i)/\Sigma_i(w_i), \tag{21.1}$$

where the summation is over the RegriddedFrames.
$FLXSCALE_i$=value of FLXSCALE attribute of each input RegriddedFrame (i.e.,, reg1.FLXSCALE).
$w_i = weight_i/FLXSCALE_i^2$ where $weight_i$ is the value of the pixel in the associated weight-frame (i.e., reg1.weight).
The $f_i$ is the pixel value in the regridded frame.
The value $w_{out}$ of the pixel in the resulting weight frame associated with the coad (i.e., coad.weight) is computed as:

$$w_{out} = \Sigma_i(w_i) \tag{21.2}$$

### 21.5.4 Coadd units

The pixel units of the CoaddedRegriddedFrame are fluxes relative to the flux corresponding to magnitude=0. In other words, the magnitude $m$ corresponding to a pixel value $f_0$ is:

$$m = -2.5log_{10}f_0. \tag{21.3}$$

The relation between pixel units in CoaddedRegriddedFrame and the RegriddedFrames from which it was derived is as follows. For a RegriddedFrame object named $reg$ we have:

$$reg.FLXSCALE == 10.0^{-0.4\times reg.ZEROPNT} \tag{21.4}$$

A magnitude=0 picels will have counts $counts_{reg}(mag = 0)$ in the RegriddedFrame:

$$counts_{reg}(mag = 0) == 10^{0.4\times reg.ZEROPNT} = 1/reg.FLXSCALE \tag{21.5}$$

(Note that reg.ZEROPNT is exposure-time specific, therefore counts instead of countrate.) Suppose one makes a CoaddedRegriddedFrame from this single RegriddedFrame $reg$. In this case the relation between the counts $count_{reg}$ in the RegriddedFrame and pixel value $v_{coad}$ in the CoaddedRegriddedFrame is made to be:

$$v_{coad} = counts_{reg} * reg.FLXSCALE \tag{21.6}$$

Suppose you know the physical flux density of the magnitude=0 object to be $f_0$ Jy. The physical flux corresponding to $v_{coad}$ is then:

$$f_{coad}(\text{Jy}) = f_0 * v_{coad} \tag{21.7}$$

For example, in the AB magnitude system $f_0 == 3631$ Jy and hence:

$$f_{coad}(\text{Jy}) = 3631 * v_{coad} \tag{21.8}$$

If ones make a CoaddedRegriddedFrame out of multiple overlapping RegriddedFrames the resulting flux is the weighted average of the input RegriddedFrames fluxes (see Section 21.5.3).

## 21.6 SourceLists in the Astro-WISE System

After having reduced the science data, source lists can be derived. This basically boils down to running SExtractor on this data, but to make sure that the extracted sources are also stored in the database for later scrutiny, some extra steps are performed. The way this is handled in the Astro-WISE system is described in the following subsections.

### 21.6.1 HOW-TO: Create Simple SourceLists From Science Frames

In the Astro-WISE system source lists are represented by `SourceList` objects. The `SourceList` class is imported by default when starting `AWE`, but if you need to import it in a script, this is done as follows:

```
awe> from astro.main.SourceList import SourceList
```

In order to derive a source list from a reduced science frame, an instance of the `SourceList` class must be created and the reduced science frame must be assigned to this object as a dependency. The three types of science frames for which a source list are commonlt made are:

- `ReducedScienceFrame`: requires separate astrometric solution (`AstrometricParameters`)
- `RegriddedFrame`: a resampled ReducedScienceFrame with incorporated astrometric solution
- `CoaddedRegriddedFrame`: a mosaic of RegriddedFrames

Invoking the 'make' method of the `SourceList` object creates the source list which is then automatically stored into the database. From the `awe`-prompt this reads for a RegriddedFrame :

```
awe> query = RegriddedFrame.filename == 'frame.fits'
awe> frame = query[0]
awe> frame.retrieve()
awe> frame.weight.retrieve()
awe> sourcelist = SourceList()
awe> sourcelist.frame = frame
awe> sourcelist.make()
awe> sourcelist.commit()
```

which will create the source list in the database. The 'make' method of the `SourceList` object first calls the 'make_sextractor_catalog' method (which obviously runs SExtractor), followed by a call to the 'make_sourcelist_from_catalog' method that ingests the sources into the database. In a similar fashion one can make a `SourceList` for a `CoaddedRegriddedFrame`. However, for a ReducedScienceFrame a separate astrometric solution is required. In this case the `awe`-prompt reads :

```
awe> query = ReducedScienceFrame.filename == 'frame.fits'
awe> frame = query[0]
awe> astrom = (AstrometricParameters.reduced == frame).max('creation_date')
awe> frame.retrieve()
awe> frame.weight.retrieve()
awe> sourcelist = SourceList()
awe> sourcelist.frame = frame
awe> sourcelist.astrom_params = astrom
awe> sourcelist.make()
```

```
awe> sourcelist.commit()
```

which will create the source list in the database. The 'make' method of the `SourceList` object first calls the 'make_sextractor_catalog' method (which obviously runs SExtractor), followed by a call to the 'make_sourcelist_from_catalog' method that ingests the sources into the database.

The configuration of SExtractor and its output can be manipulated through the `SourceList` interface as well. This is done through the `sexconf` and `sexparam` dependencies, respectively. For example, if one wants to run SExtractor with a detection threshold of 12, and to have `MAG_APER` and `MAGERR_APER` as additional output, the `AWE`-session above changes into (skipping the database query and retrieve operations):

```
awe> sourcelist = SourceList()
awe> sourcelist.frame = frame
awe> sourcelist.astrom_params = astrom
awe> sourcelist.sexconf.DETECT_THRESH = 12
awe> sourcelist.sexparam = ['MAG_APER', 'MAGERR_APER']
awe> sourcelist.make()
awe> sourcelist.commit()
```

with the extra output parameters added to the definition of the sources contained in the list.

A special feature has been implemented to allow for skipping some specified records in the SExtractor catalog. This works as follows:

```
awe> sourcelist = SourceList()
awe> sourcelist.record_skiplist = [13, 169]
```

The last statement will make sure that the sources on record 13 and 169 of the SExtractor catalog are NOT ingested into the list.

### 21.6.2 SegmentationImage

When a SourceList is created, by default a *SegmentationImage* is also made. A segmentation image is a file that SExtractor can generate, that defines which pixels are part of what object as detected by SExtractor.

A SegmentationImage is derived from the class *CheckImage*. If you have made a SourceList, its associated SegmentationImage can be found with a query such as this one:

```
awe> sl = (SourceList.SLID == 423431)[0]
awe> segm = (SegmentationImage.sourcelist == sl)[0]

# and display it ...
awe> segm.display()
```

### 21.6.3 Using SourceList with SExtractor double-image mode

When a SourceList is made for an image one has the option to specify a secondary image. This makes the SExtractor software which SourceList uses, run in the so-called "double-image mode".

To explain the process, this is how you would normally run SExtractor in double-image mode:

```
sex -c sex.conf image1.fits image2.fits
```

Here "image1.fits" is called the detection image and "image2.fits" is the measurement image. The detection image and measurement image must have identical dimensions. The detection

image is the image used to determine which pixels are part of sources. Physical properties are derived from the measurement image, using the pixels attributed to sources in the detection image. Changing the measurement image for another image will not modify the number of detected sources, nor will it affect their pixel positions or basic shape parameters. All other parameters (photometric parameters in particular) will use measurement image pixel values, which allows one to easily measure pixel-to-pixel colours.

The images must have identical dimensions because in double-image mode all sources detected in the detection image have their photometry measured at exactly the same pixel locations in the measurement image.

Using SExtractor image mode is only accurately possible when using *RegriddedFrames* or *CoaddedRegriddedFrames* that have been regridded to the same grid. A cutout of the overlapping region is automatically made during the creation of the SourceList. For other frames (i.e. *ReducedScienceFrames*) no attempt it made to determine an overlap; the images are assumed to cover exactly the same area on the sky.

When using SourceListTask the measurement images are specified via the "filenames" variable and all corresponding detection images are specified via the "detection_filenames" variable. Example:

```
awe> task = SourceListTask(filenames=['image2.fits'],
                           detection_filenames=['image1.fits'])
awe> task.execute()
```

or with short options:

```
awe> task = SourceListTask(f=['image2.fits'], df=['image1.fits'])
awe> task.execute()
```

Note that the primary image used in the SourceList is the *measurement* image; while this is the second filename that is given to SExtractor.

For your information, here are most (if not all) parameters that are derived from the detection image in double image mode:

```
X_IMAGE          X2_IMAGE       A_IMAGE
Y_IMAGE          Y2_IMAGE       B_IMAGE
XMIN_IMAGE       XY_IMAGE       ELONGATION
XMAX_IMAGE       CXX_IMAGE      ELLIPTICITY
YMIN_IMAGE       CYY_IMAGE      KRON_RADIUS
YMAX_IMAGE       CXY_IMAGE

THRESHOLD      (same for all sources)
MU_THRESHOLD   (same for all sources)
```

### 21.6.4 HOW-TO: Use External SourceLists

The description of the use of `SourceList` objects given in §21.6.1 deals with deriving source lists directly from a science frame. However, already existing source lists can also be stored into the database through the use of `SourceList` objects. This is useful if one wants to store a catalog of standard stars into the database. This is done by instantiating a `SourceList` object with assigning the external source list name to `catalog`, and calling the 'make_sourcelist_from_catalog' method. As seen from the `awe`-prompt:

```
awe> sourcelist = SourceList()
awe> sourcelist.catalog = 'external.fits'
```

```
awe> sourcelist.make_sourcelist_from_catalog()
awe> sourcelist.commit()
```

and the ingestion of the external source list is complete.

For this mechanism to work, the external catalog has to meet some conditions. First and foremost, the external catalog has to be of the same type and layout of a Sextrator catalog like the ones produced in the Astro-WISE system. This means that the external catalog should be in LDAC fits format, and must have an `OBJECTS` and `FIELDS` table. Secondly, the `OBJECTS` table should have the following columns : `RA` or `ALPHA_SKY`, `DEC` or `DELTA_SKY`, `A`, `B`, `Theta` or `POSANG`, and `FLAG`. Besides these mandatory columns, any other column may be present in the catalog with no restrictions on the name.

### 21.6.5 HOW-TO: Use SourceLists

Once a `SourceList` is ingested into the database, the usual method of retrieving database objects should be used to access the stored information, i.e.

```
awe> sourcelist = (SourceList.SLID == 0)[0]
```

Each `SourceList` has an unique SoureList Identifier (`SLID`) and/or a name. A `SourceList` name does not have to be unique, so the following statement might result in many sourcelists:

```
awe> sourcelists = (SourceList.name == 'MySourcelists')
```

Each source in a `SourceList` has a unique Source Identifier (`SID`), which is assigned during ingestion into the data base and which starts at zero. The number of sources in the sourcelist is obtained with the `len` function:

```
awe> number_of_sources = len(sourcelist.sources)
```

or with the special attribute `number_of_sources`:

```
awe> number_of_sources = sourcelist.number_of_sources
```

The column information is obtained as follows:

```
awe> column_info = sourcelist.sources.get_attributes()
```

`column_info` is a dictionary with column names as keys and column types as contents. Column data is for example retrieved with the following statement:

```
awe> RA = sourcelist.sources.RA
```

where `RA` is a list containing the values of column 'RA'. Row data is retrieved in a similar way, for example:

```
awe> first_source = sourcelist.sources[0]
```

where `first_source` is a dictionary with column names as keys and column values as contents.

There are some methods defined for sourcelists. At the moment these are:

- `sourcelist.sources.make_skycat(sid_list=None, filename=None)`

  `make_skycat` creates a "dump" of the SourceList object that can be used to overplot a frame in ESO skycat.
  If `sid_list` is given it should be a list of SID's (Source IDentifiers) which will be output to the skycat output file.
  The default `filename` is the name of the SourceList with extension .scat.

- `sourcelist.sources.area_search(self_search=True, htm_depth=20, Area=None)`

  `area_search` searches the specified area for sources within a given distance from a position {i.e. `Area=(RA,DEC,Distance)`} or within an area delimited by three or four positions {i.e. `Area=[(RA0,DEC0),(RA1,DEC1),(RA2,DEC2)]`}.
  All positions and distances are in degrees.
  For the search, htm trixel ranges are searched at the specified depth (default is 20). If `self_search==True`, only the current SourceList is examined, if `self_search==False`, all SourceLists will be examined.
  Return value is a dictionary with the SLID's (SourceList IDentifiers) for keys and lists of SID's (Source IDentifiers) for values. Example:

```
awe> r = sourcelist.sources.area_search(Area=[(1.0,0.0),(0.0,0.0),(0.0,1.0),(1.0,1.0)])
awe> print 'Sources found: ', len(r[sourcelist.SLID])
```

- `sourcelist.sources.get_data(dict)`

  `get_data` returns the data associated with given attributes. `dict` is a dictionary where the keys represent the attributes returned and their subsequent values should be on input `None` or an empty list (`[]`).
  On output the actual values of the attributes are stored as key values.
  The function returns a list of Source IDentifiers (SID's). Example:

```
awe> dict = {'RA':[], 'DEC':[], 'MAG\_ISO':[]}
awe> r = sourcelist.sources.get\_data(dict)
```

- `sourcelist.sources.sql_query(dict, query_string)}`

  `sql_query` performs an SQL query on source attributes. `query_string` may contain only valid SQL query syntax and the names of the attributes between double quotes (").
  `dict` is a dictionary where the keys represent the attributes returned and their subsequent values should be on input `None` or an empty list (`[]`).
  On output the actual values of the attributes are stored as key values.
  The function returns a list of Source IDentifiers (SID's). Example:

```
awe> dict = {'RA':[], 'DEC':[], 'MAG\_ISO':[]}
awe> r = sourcelist.sources.sql_query(dict, '"MAG_ISO"<18.5 AND "A">0.2')
```

- `sourcelist.sources.make_image_dict(sids, mode='sky')`

  `make_image_dict` returns a dictionary which can be used as input for the interface to the image server.
  `sids` may contain one SID or a list of SID's.
  `mode` is used to specify whether sky (default) or grid coordinates are wanted. Example:

```
awe> imgdict = sourcelist.sources.make_image_dict(0)
awe> from astro.services.imageview.imgclient import imgclient
awe> ic = imgclient(imgdict, wide_high=[150, 150])
awe> ic.getimg()
```

### 21.6.6 HOW-TO: Associate SourceLists

To spatially associate two different sourcelists the make method of the `AssociateList` class can be used. The association is done in the following way:

- First the area of overlap of the two sourcelists is calculated. If there is no overlap no associating will be done.
- Second the sources in one sourcelist are paired with sources in the other sourcelist if they are within a certain distance from eachother. Default distance is 5". The pairs get an unique associate ID (`AID`) and are stored in the associatelist. A filter is used to select only the closest pairs.
- Finally the sources which are not paired with sources in the other list and are inside the overlapping area of the two sourcelists are stored in the associatelist as singles. They too get an unique `AID`.

NOTE: in default mode all objects in the to-be associated sourcelists, which have SExtractor Flag $> 0$, are filtered out. To keep objects which have Flag $> 0$, one must type 'AL.process_params.SEXTRACTOR_FLAG_MASK=255' in the example below.
In Python this is done as follows:

```
awe> AL = AssociateList()
awe> sl0 = (SourceList.SLID == 0)[0]
awe> sl1 = (SourceList.SLID == 1)[0]
awe> AL.input_lists.append(sl0)
awe> AL.input_lists.append(sl1)
awe> AL.set_search_distance(5.0)
awe> AL.associatelisttype=1
awe> AL.make()
awe> AL.commit()
```

If one does not want to filter for closest pairs, the following statement should be executed before the make:

```
awe> AL.single_out_closest_pairs(False)
```

Optionally one can also specify the search area. Only sources from both sourcelists which lie inside this area are matched. This can be done as follows, before you do `AL.make()`

```
awe> AL.set_search_area(llra,lldec,lrra,lrdec,urra,urdec,ulra,uldec)
```

where

```
llra  = lower left R.A. of search area
lldec = lower left Dec.  of search area
lrra  = lower right R.A. of search area
lrdec = lower right Dec.  of search area
urra  = upper right R.A. of search area
urdec = upper right Dec.  of search area
ulra  = upper left R.A. of search area
uldec = upper left Dec.  of search area
```

A previously created `AssociateList` can also be input to a new `AssociateList`. Simply put the existing `AssociateList` in the `input_lists` as shown in the following example:

```
awe> AL = AssociateList()
```

```
awe> ALs = (AssociateList.ALID == 0)
awe> SLs = (SourceList.SLID == 2)
awe> AL.input_lists.append( ALs[0] )
awe> AL.input_lists.append( SLs[0] )
awe> AL.make()

awe> AL.commit()
```

The member sourcelists can be derived from the `AssociateList` attribute `sourcelists`. The above example creates a so-called 'chain' association: The new `SourceList` is paired with the last `SourceList` in the `sourcelists` of the existing `AssociateList`. Figure 21.2 gives an example of a chain association.

The pairing can also be done in a different way in that the pairing is always done between the new `SourceList` and the first one in the `sourcelists` list of the input AssociateList. This is a so-called 'master' association. Figure 21.3 gives an example of a master association.

Another method of pairing sourcelists is by matching sources of a number of sourcelists simultaneously. Here only sources which have at least one companion end up in an association, so singleton sources are out. This is a so called 'matched' association. Figure 21.4 gives an example of a matched association. Once a matched associatelist has been created, it can not be used as input to another association.

The `AssociateList` attribute `associatelisttype` sets the type of association: 1 for chain, 2 for master and 3 for matched type, chain is the default.
Once an `AssociateList` is created the number of associates can be obtained as follows:

```
awe> print len(AL)
```

There are two functions which count the associations:

- `count_from_members(members=0, mode='EQ')` counts the number of associations which have a specified number of members. With `mode` you can specify whether you want associations counted with `'LT'`, `'LE'`, `'EQ'`, `'GE'` or `'GT'` the specified number of members. For example, to count the number of pairs in an associatelist derived from only 2 sourcelists one could do the following:

  ```
  awe> print AL.associates.count_from_members(members=2,mode='EQ')
  ```

  To count the number of singles one could do:

  ```
  awe> print AL.associates.count_from_members(members=1,mode='EQ')
  ```

- `count_from_flag(mask=None, mode='ANY', count=None, countmode='EQ')` returns the number of associations which contain sources from specified sourcelists. `mask` works as a bitmask, each bit set representing a sourcelist which was input to the associatelist. For example bit 2 set means that only the wanted associations which contain a source from sourcelist number 2 will be counted (depends on `mode`). `mode` determines how the masking works. `mode` can be: `'ALL'` (count only the associations which contain sources from exact the specified sourcelists), `'INTERSECT'` (count only the associations which contain sources from at least the specified sourcelists) and `'ANY'` (count only the associations which contain sources from at least one of the specified sourcelists).

  It is also possible to specify a bitcount operation with parameter `count`. `count` determines the number of different sourcelists which participate in an association, i.e. if `count = 3` then only associations are counted which have sources from 3 different sourcelists (depends

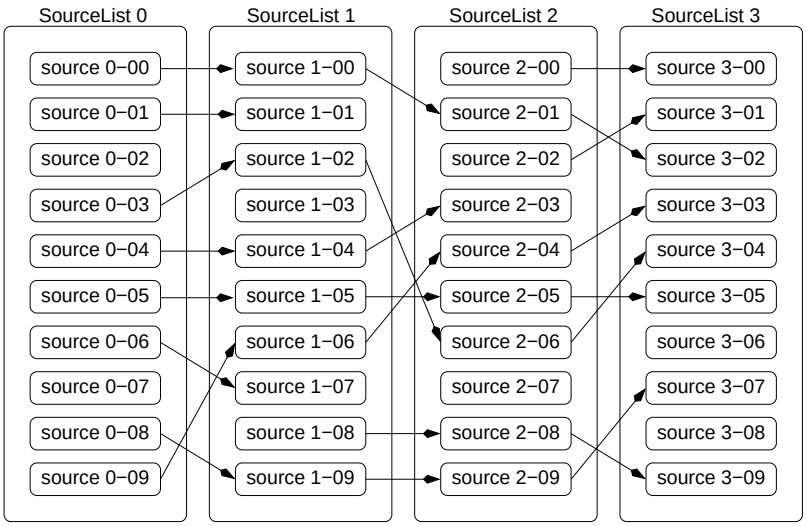



Figure 21.2: An example of a chain association with 4 sourcelists. Each sourcelist is always paired with the previous sourcelist

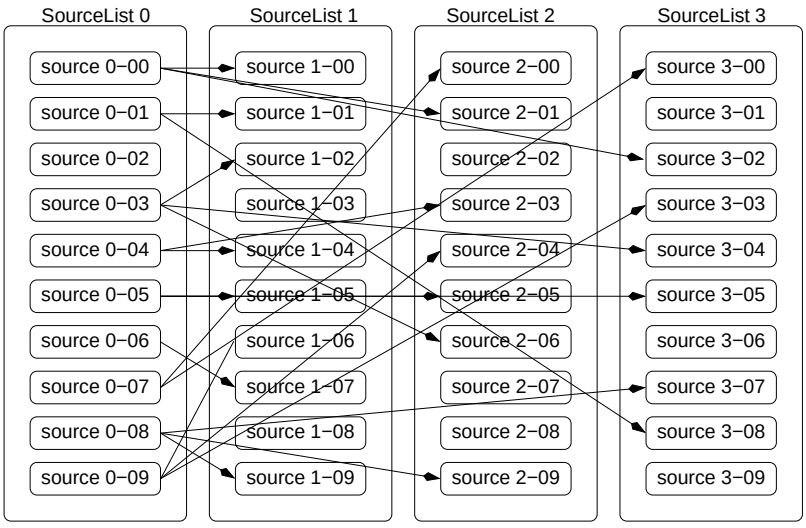



Figure 21.3: An example of a master association with 4 sourcelists. All sourcelists are paired with the first sourcelist

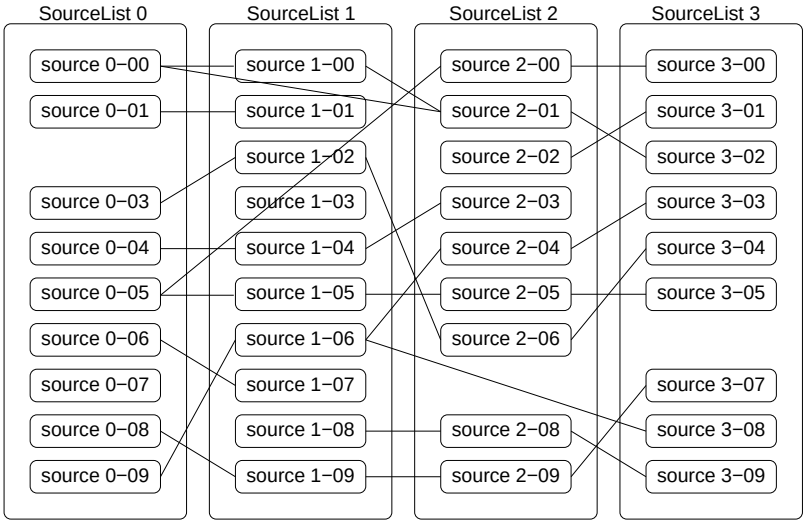



Figure 21.4: An example of a matched association with 4 sourcelists. All sourcelists are paired with the other sourcelists simultaneously. All interconnected pairs make one association.

on `countmode`). `countmode` determines whether the number of participation sourcelists should be less than (`'LT'`), less than or equal (`'LE'`), equal (`'EQ'`), greater than or equal (`'GE'`) or greater than (`'GT'`) `count`.

For example, to count the number of pairs in an associatelist derived from only 2 sourcelists (i.e. the first and the second sourcelist so `mask = 1 + 2`) one could do the following:

```
awe> print AL.associates.count_from_flag(mask=3,mode='ALL')
```

To count the number of singles one could do:

```
awe> print AL.associates.count_from_flag(mask=1,mode='ALL')
awe> print AL.associates.count_from_flag(mask=2,mode='ALL')
```

The first command counts the number of singles from the first Sourcelist, the second command those from the second sourcelist.

With the function `get_data` one can get the `SLID`, `SID` or in fact any attribute of member sources. The function is described as follows:
`get_data(attrlist=[], mask=None, mode='ANY', count=None, countmode='EQ')` returns requested attributes of the associated sources. The function returns a dictionary with as key the `AID` (Association IDentifier) and as value the attribute values according to the modified list of specified attributes. For each requested sourcelist a separate list of attributes is returned.
`attrlist` contains the wanted attributes. Note that `SLID` and `SID` will always be returned and that `SLID` is the first and `SID` the second item. On return `attrlist` will be modified accordingly.

`mask`, `mode`, `count` and `countmode` work as in `count_from_flag`.

The following example shows how to obtain the `RA`, `DEC` and `MAG_ISO` attributes from an associatelist:

```
awe> attrlist = ['RA', 'DEC', 'MAG_ISO']
awe> r = AL.associates.get_data(attrlist,mask=3,mode='ALL')
awe> aids = [k for k in r.keys()]
awe> aids.sort()
awe> print 'AID', attrlist
awe> for aid in aids[:10]:
awe>     for row in r[aid]:
awe>         print aid, row
```

The output looks like:

```
AID ['SLID', 'SID', 'RA', 'DEC', 'MAG_ISO']
0 [44, 0, 280.86871938032101, 0.36483871219556502, -4.8930621147155797]
0 [45, 12, 280.868644520494, 0.36475058531692101, -6.9002099037170401]
1 [44, 1, 280.86753832027802, 0.28356582313986001, -4.3539190292358398]
1 [45, 24, 280.86745181025799, 0.28365591902279302, -6.2037878036498997]
2 [44, 2, 280.86763239079198, 0.27250082665266701, -5.1681485176086399]
2 [45, 11, 280.86766082401601, 0.27245588811475802, -6.6134591102600098]
3 [44, 3, 280.867557240373, 0.26285186993244197, -5.4812445640564]
3 [45, 19, 280.867494620603, 0.262849873260255, -7.01251125335693]
4 [44, 4, 280.86742472466602, 0.25502430125932601, -5.4748520851135298]
4 [45, 16, 280.86738586918, 0.25500542211914801, -6.6722841262817401]
5 [44, 7, 280.86738062915998, 0.275450798599644, -4.89949655532837]
5 [45, 20, 280.86741490360203, 0.27540606762365499, -6.2980375289917001]
6 [44, 8, 280.86823775792999, 0.32890289165032699, -6.9213681221008301]
6 [45, 40, 280.86818782868102, 0.328900495041689, -8.8428869247436506]
7 [44, 9, 280.86780428279701, 0.26950356072341503, -7.1055769920349103]
7 [45, 32, 280.86776951099699, 0.26946985609771201, -8.1041078567504901]
8 [44, 13, 280.86790089506002, 0.345712819247041, -6.1025667190551802]
8 [45, 38, 280.86788627250701, 0.345742041485333, -7.4083743095397896]
9 [44, 14, 280.86739941046397, 0.333885857133919, -5.7742152214050302]
9 [45, 41, 280.86730665910602, 0.33388354451305802, -6.7259593009948704]
```

As for sourcelists, a function called `make_image_dict` has been implemented:
`al.associates.make_image_dict(aids, mode='sky')`

`make_image_dict` returns a dictionary which can be used as input for the interface to the image server.
`aids` may contain one AID or a list of AID's.
`mode` is used to specify whether sky (default) or grid coordinates are wanted. Example:

```
awe> imgdict = al.associates.make_image_dict(0)
awe> from astro.services.imageview.imgclient import imgclient
awe> ic = imgclient(imgdict, wide_high=[150, 150])
awe> ic.getimg()
```

For inspecting the distances between all associated source pairs one can use the method `get_distances`:
`get_distances` calculates the distances between all possible pairs in an association. The output is a dictionary with as key the associate ID (AID) and as value a list containing for each pair a tuple which consists of three items: `((SLID1, SID1), (SLID2, SID2), DISTANCE)`. `SLID1` is

always `<= SLID2` and when `SLID1=SLID2, SID1 < SID2`.

### 21.6.7 Scientific Examples Using AssociateLists

In this section some examples for scientific use of AssociateLists are shown.

The first example shows how to use an associate list to find extraordinary sources. Suppose we have an `AssociateList` which is created by associating two sourcelists which were derived from images with different filters, say `B` and `V`. To find those sources which have a `B - V > 1.5` one could do the following:

```
awe> attrlist = ['RA', 'DEC', 'MAG_ISO']
awe> r = al.associates.get_data_on_associates(attrlist, mask=3, mode='ALL')
awe> print 'Found %d pairs' % (len(r))
awe> i = attrlist.index('MAG_ISO')
awe> newr = {}
awe> for aid in r.keys():
awe>     if (r[aid][0][i] - r[aid][1][i]) > 1.5:
awe>         newr[aid] = r[aid]
awe> print 'Found %d pairs with MAG_ISO diff > 1.5' \% (len(newr))
```

Note that we use `mask = 3` to indicate that we want only those associations which have a source from the first (bit 1 = 1) and second (bit 2 = 2) sourcelist (mask = 1 + 2).

The second example shows how to find 'drop-outs' in an `AssociateList` which has been created associating three sourcelists, each having been obtained for different filters. We are looking for sources which appear only in the second and third sourcelist (i.e. they were not detected with the filter used for obtaining the first sourcelist). This could be done as follows:

```
awe> attrlist = ['RA', 'DEC', 'MAG_ISO']
awe> r = al.associates.get_data_on_associates(attrlist, mask=6, mode='ALL')
awe> print 'Found %d pairs' % (len(r))
```

Note that we use `mask = 6` to indicate that we want only those associations which have a source from the second (bit 2 = 2) and third (bit 3 = 4) sourcelist (mask = 2 + 4).

The third example shows how to obtain the data to make a `B - V` vs. `U - V` plot. Suppose we have associated three sourcelists, each obtained for different filters `U, B, V`. Then we could do this as follows:

```
awe> attrlist = ['RA', 'DEC', 'MAG_ISO']
awe> r = al.associates.get_data_on_associates(attrlist, mask=7, mode='ALL')
awe> print 'Found %d triples' % (len(r))
awe> i = attrlist.index('MAG_ISO')
awe> x = []
awe> y = []
awe> for aid in r.keys():
awe>     x.append(r[aid][1][i] - r[aid][2][i])
awe>     y.append(r[aid][0][i] - r[aid][2][i])
```

Note that we use `mask = 7` to indicate that we want only those associations which have a source from the first (bit 1 = 1), the second (bit 2 = 2) and third (bit 3 = 4) sourcelist (mask = 1 + 2 + 4).

### 21.6.8 Visualizing associated sources: creating a skycat catalog

It is possible to visualize which sources are associated in an AssociateList by creating a *skycat* catalog. This catalog can be used in the *skycat* FITS viewer. Use the following method:

- `make_skycat_on_associates(slid, mask=None, mode='ALL')` takes as input a SourceList identifier; the SourceList in the AssociateList for which to produce the skycat catalog. The arguments *mask* and *mode* are the same as described in the `count_from_flag` method. If the optional arguments are not specified the resulting catalog will contain the associates that have members in each input SourceList.

Example:

```
awe> al.make_skycat_on_associates(slid=123456)
```

### 21.6.9 HOW-TO: CombinedList

CombinedList is a library which was written to create a SourceList from AssociateList. At the moment the size of the newly created SourceList is limited to 200,000 sources, the bigger SourceLists are possible with the special function provided by the user request.

#### Main principles

The Astro-Wise user has an ability to associate SourceLists cross-identifying sources. The functionality of AssociateList is described in the corresponding how-to.

AssociateList contains links to sources in SourceList only and does not provide any ability to combine attribute values of cross-identified SourceLists. CombinedList fills this gap and allow user to create a permanent SourceList with combined attributes for all sources. This is especially useful for creating of multiband catalogs. At the same moment CombinedList is not a persistent class but a library of methods. The input for the CombinedList methods is an AssociateList, the output is a SourceList.

The main task - creating a stable multiband catalog - is realized by adding a new attributes. The user can set an attribute which he would like to treat as a magnitude (normally this is `MAG_ISO` or other magnitude attributes from sextractor) and create an average magnitude for associated sources (`MAG_1`) accompanied by the rms of the magnitude (`MAGERR_1`), corresponding flags (if user provided information about attribute with flags, `MAGFLAG_1`), and number of sources which were joint (`MAGN_1`).

In the case of the same filters for SourceLists (Fig. 21.5) the newly created SourceList will have the only attribute set for magnitude, in the case of different filters (Fig. 21.6) two sets of magnitudes will be created.

The information about filters is saved in the string `filters` in the following format `'MAG_1:<filter_name>,MAG_2:<f`
There are up to 20 filters possible.

The user has an ability to set a type of operations which will be provided with input AssociateList. AssociateList is created over SourceLists, sources in SourceLists which corresponds to AID (ID of associations) will be combined according to the specification from user and inserted into a newly created SourceList. The user has an ability to insert into this newly created SourceList not only associated sources but non-associated ones as well.

Let us see the case of an overlapped SourceLists (SL1 and SL2, Fig. 21.7, sources are associated in the area of AssociateList AL). In the case of `COMBINE_METHOD=1` all sources, including non-associated from the area out of bounding box of AssociateList will be included into newly created SourceList (Fig. 21.7). In the case of `COMBINE_METHOD=2` associated sources from the area

Figure 21.5: CombinedList on the same filter

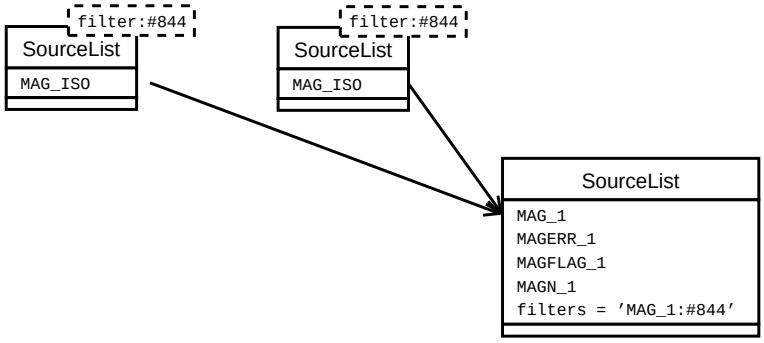



Figure 21.6: CombinedList on different filters

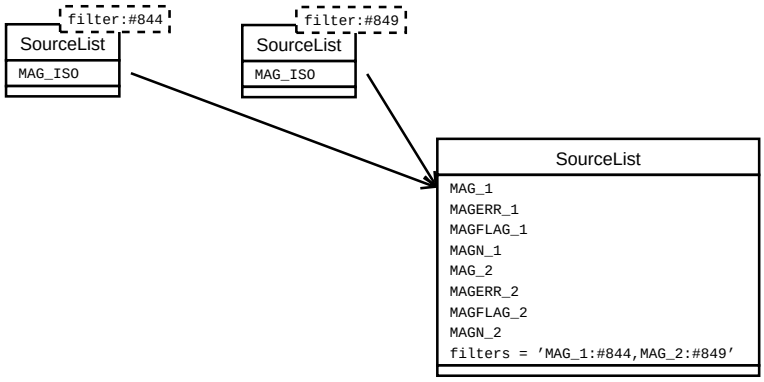
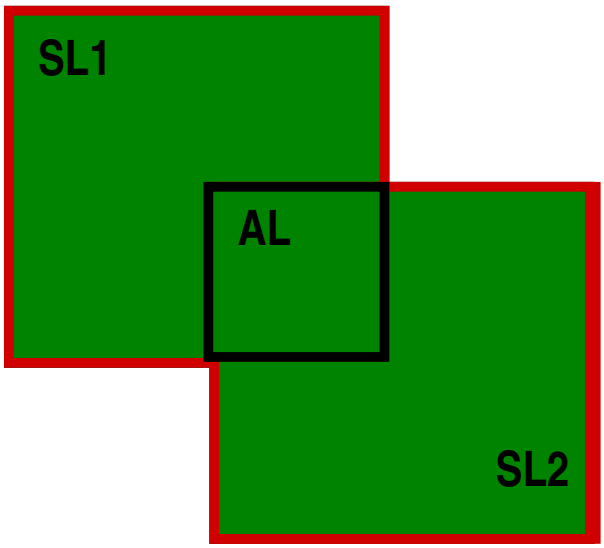


Figure 21.7: Spatial localization in the case of COMBINE_METHOD=1




covered by AssociateList will be included only (Fig. 21.8), and in the case of `COMBINE_METHOD=3` only non-associated sources will be inserted (Fig. 21.8). Please, note, that usually area covered by AssociateList contains non-associated sources as well (sources which do not satisfy to requirements implied by the user for AssociateList). These sources will be included as well if user chooses `COMBINE_METHOD=1` and `COMBINE_METHOD=3`.

**The life cycle of SourceList**

The realization of CombinedList library allows to reuse of a newly created SourceList as an input for a new AssociateList. Fig. 21.10 shows the possible life cycle of the new SourceList. New attributes are:

`associatelist` - the ID of the parent AssociateList,

`filters` - the list of magnitudes in the SourceList,

`COMBINE_METHOD` - the method used by CombinedList to make new coordinates/magnitudes,

for each photometric band 4 attributes - `mag_i` for magnitudes, `magErr_i` for errors of magnitudes, `magFlag_i` for the flag and `magN_i` for number of stars combined for this value of magnitude.

As result there are three types of SourceLists (in the single SourceList class):

- "original" SourceList - a SourceList from sextractor
- "external data source" SourceList - a sourcelist ingested from an external to Astro-Wise catalog. This SourceList can contain a number of magnitudes per source
- "combined" SourceList produced by CombinedList

Figure 21.8: Spatial localization in the case of COMBINE_METHOD=2

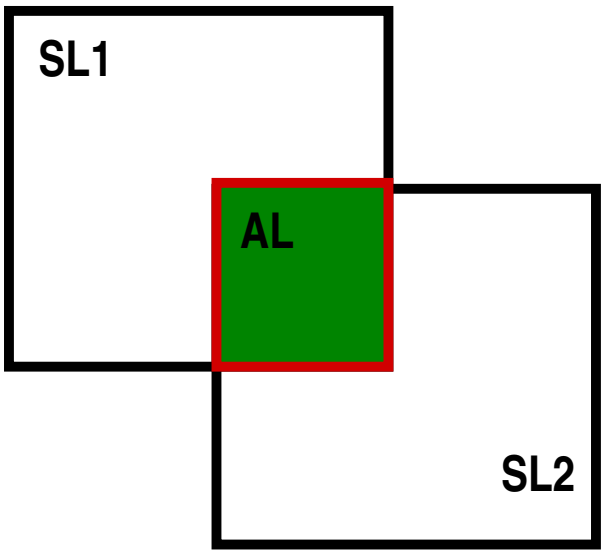



Figure 21.9: Spacial localization in the case of COMBINE_METHOD=3

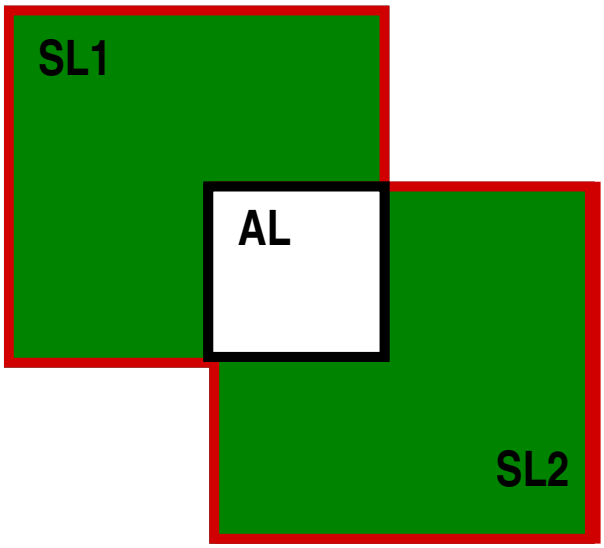
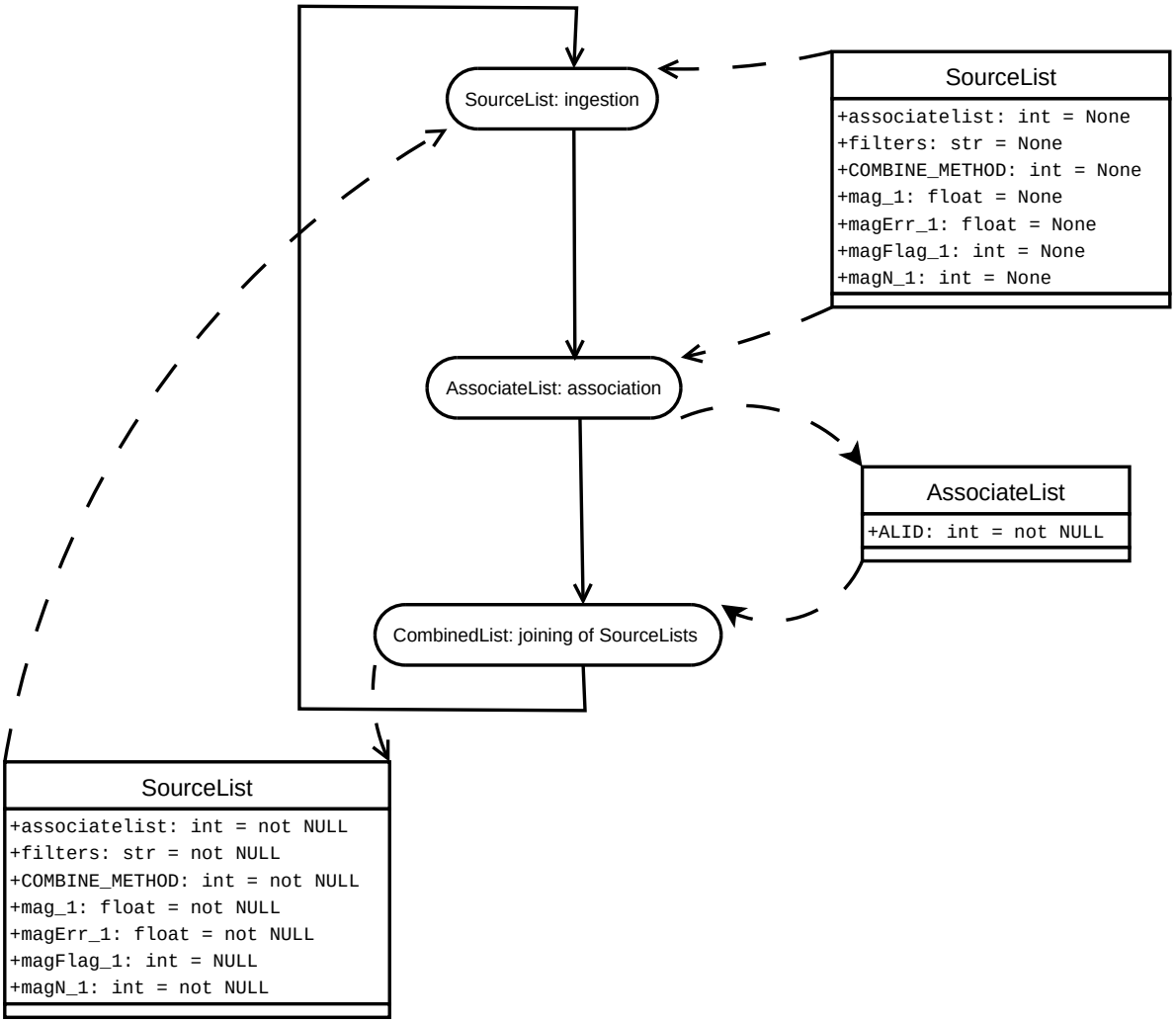


Figure 21.10: New SourceList cycle



### Functions and Attributes

**Initializing** CombinedList must be initiated with the AssociateList of type 1 or 2.

```
cl=CombinedList(input_AssociateList)
```

The place of CombinedList is in **astro.main.CombinedList**.

### Setting method of combining of sources

```
cl.set_combined_method(COMBINE_METHOD)
```

`COMBINE_METHOD` can be 1, 2 or 3. By default `COMBINE_METHOD=2`.

### Setting user defined magnitudes

```
cl.set_user_defined_magnitudes(attributes)
```

`attributes` is a dictionary which specify magnitude-like attributes defined by the user. The format of the dictionary is

```
attributes={<attribute_name>:<filter_name>,}
```

for example,

```
attributes={'MAG_IZO':'#844','MAG_AUTO':'#849'}
```

Please, note that the definition like this:

```
attributes={'MAG_IZO':'#844','MAG_AUTO':'#844'}
```

will join `MAG_IZO` and `MAG_AUTO` in one attribute in newly created SourceList.

**Setting user defined attributes** Unlike user defined magnitudes user defined attributes will be joint in a new SourceList in an attribute with the same name. For example,

```
cl.set_user_defined_attributes(attributes)
```

where attributes is a list

```
attributes=['B','MAG_RAD']
```

will create in a new SourceList two attributes `B` and `MAG_RAD`. The aggregation function by default is `AVG`, but the user can set any Oracle aggregation function as attribute with the same name. For example,

```
cl.set_aggregate_functions(attributes)
```

where attributes is a dictionary

```
attributes={<attribute_name>:<aggregate_function>}
```

For example,

```
attributes={'B':'MAX','MAG_RAD':'MIN'}
```

**Setting magnitude flags** The user can accompany a magnitude with a flag taken from the SourceList using a function

```
cl.set_magnitude_flags(attributes)
```

where

```
attributes={<magnitude_name>:<flag_attribute>}
```

For example, if the user would like to insert magnitude flags for `MAG_ISO` from `MAG_FLAG`

```
attributes={'MAG_ISO':'MAG_FLAG'}
```

By default the maximum flag will be taken but the user can select aggregate function the same way it was described above.

```
attributes={'MAG_FLAG':'MIN'}
```

**Debugging** The user can set a debugging during creating of a SourceList

```
cl.set_debug()
```

The debugging information shows user executed SQL statements and time spent for execution.

### Making and committing

```
cl.make()
```

and

```
cl.commit()
```

the last function shows the information for created SourceList.

### Use in practice

AssociateLists is created. How to make a new SourceList from AssociateList? This can be done with CombinedList library.

First, let us select an AssociateList

```
awe> als=(AssociateList.ALID==7842)
awe> als[0].info()
Name of AssociateList :Assoc-GVERDOES------------------53923.5717701.0000007842.alist
AssociateList ID      :7842
Associates in list    :26360
```

Then we initialize a CombinedList with the selected AssociateList

```
awe> cl=CombinedList(als[0])
```

We have to select method which we will use to combine data. There are three of them: `combined_method=1` will include in the resulting SourceList all sources (associated by AssociateList and non-associated), `combined_method=2` will include only sources which are presented in AssociateList, and `combined_method=3` will include only sources which are presented in SourceLists (used to create AssociateList) but not in AssociateList itself. By default `combined_method=2`.

```
awe>cl.set_combined_method(1)
```

We have also to specify which attributes of input SourceLists (input to AssociateList) we would like to see in the output SourceList. There are two modes: to treat attributes as a magnitude attribute (4 new attribute will be created - `MAG_1` to store average value, `MAGFLAG_1` to store flag for the magnitude, `MAGERR_1` to store rms of the value, `MAGN_1` to store a number of input values, 2 or 1 in the example) or to specify the attribute as user-defined with the user-selected aggregate function.

For example, we want to see in the output SourceList `MAG_ISO` which is a magnitude in the Astro-WISE filter `'#844'`.

```
awe>cl.set_user_defined_magnitudes({'MAG_ISO':'#844'})
```

At the same time we want to see in the output SourceList `FLUX_RADIUS` and `YM2`, and we want a maximum value for the first attribute not an average.

```
awe>cl.set_user_defined_attributes(['FLUX_RADIUS','YM2'])
awe>cl.set_aggregate_functions({'FLUX_RADIUS':'MAX'})
```

Next we make and commit a new SourceList

```
awe>cl.make()
awe>cl.commit()
SourceList: Name of SourceList : SL-YOURNAME-0000423231
SourceList ID      : 423231
Sources in list    : 34645
Parameters in list : 15

 |
 +-COMBINE_METHOD: 1
 +-OBJECT:
 +-SLID: 423231
 +-associatelist: 7842
 +-astrom_params: None
 +-chip: None
 +-creation_date: 2008-08-14 15:46:20.921300
 +-detection_frame: None
 +-filename:
 +-filter: None
 +-filters: MAG_1:#844
 +-frame: None
 +-globalname:
 +-instrument: None
 +-is_valid: 1
 +-llDEC: -43.2780281843
 +-llRA: 201.747154592
 +-lrDEC: -43.2780165148
 +-lrRA: 200.962548787
 +-name: SL-YOURNAME-0000423231
 +-number_of_sources: 34645
 +-object_id: '5220897963995C33E0407D81C5063020'
 +-process_params: <astro.main.SourceList.SourceListParameters object at 0xb42bc5ac>
 +-sexconf: <astro.main.Config.SextractorConfig object at 0xb42bcd4c>
 +-sexparam: <class 'common.database.typed_list.typed_list'>(<type 'str'>, [])
 +-sources: {'MAGFLAG_1': <type 'long'>, 'YM2': <type 'float'>,
```

```
    'MAG_1': <class'common.util.types.single_float'>,
    'FLUX_RADIUS': <class 'common.util.types.single_float'>,
    'MAGERR_1': <class 'common.util.types.single_float'>,
    'MAGN_1': <type 'long'>}
 +-ulDEC: -42.7262721851
 +-ulRA: 201.743662417
 +-urDEC: -42.7262607388
 +-urRA: 200.966071491
None
```

As we can see, new SourceList with SLID=423231 has attributes `MAG_1` (contains `MAG_ISO` from input SourceLists), `FLUX_RADIUS` (maximum `FLUX_RADIUS` from input SourceLists) and `YM2` (average `YM2` from input SourceLists) and contains 34645 sources.

# Chapter 22

# Analysis Tools

## 22.1 HOW-TO use Galfit in Astro-WISE

### 22.1.1 Introduction

GALFIT is a galaxy/point source fitting algorithm that fits 2-D parameterized, axisymmetric, functions directly to images (Peng, Ho, Impey, & Rix 2002, AJ, 124: 266). The program has been developed by Chien Peng who maintains a Galfit homepage.

### 22.1.2 Astro-WISE implementation

The program Galfit (version 2.0.3b Feb 2005) has been incorporated into Astro-WISE. This is done by providing a Python wrapper around the Galfit program which stores the Galfit input and output to the Astro-WISE database.

The main classes in the Galfit object model for Astro-WISE (i.e. those classes that are stored in the database, and must be queried on to get results) are these:

- *GalFitModel*: The main Galfit class. Conceptually this **is** the model of a single galaxy. It contains methods to create a model image, residual image etc. Additionally it contains a list of *GalFitComponents* (see below), which together constitute the model of the galaxy.
- *GalFitComponent*: parent class for *GalFitSersic*, *GalFitDevauc*, *GalFitSky*, etc.
- *GalFitSersic*: stores the parameters of the Sersic function, contains initial values, final values, and errors of fitted parameters, as well as a switch which defines whether the parameter was fixed or free in the fit (see table 22.1).
- *GalFitDevauc*, *GalFitSky*, etc.: analogous to GalFitSersic
- *GalFitList*: used as a tool to link a certain group of GalFitModels together through an identifier, and optionally a name.
- *GalFitParameters*: stores the global configuration parameters (named A-S in the Galfit configuration file), see table 22.2 for Astro-WISE names.
- *GalFitConstraint*, *GalFitAbsConstraint*, *GalFitDiffConstraint*, etc.: controls and stores constraints on fitted parameters

The input which Galfit needs to run are an image of the object to be fitted, information on the 2-D profile to be fitted and various parameters which configure how to perform the fitting. The galaxy images to be fitted by Galfit are defined via a SourceList ID (SLID) and the source IDs (SID) of sources in it. For each source a cut-out of the image is made, centered on the source position which is then fed to Galfit.

The 2-D parameterized functions (e.g., a sersic profile) that can be fitted to this cut-out image are called "components" in Astro-WISE. The components that currently can be fitted in Astro-WISE are listed in Table 22.1. The definition of the components can be found in the Galfit manual.

### 22.1.3 Running GalFit

For example to fit a sersic profile plus a sky background to 4 sources from the sourcelist with SLID=57424 which have SID=7,3,9,33 using the cpu on your own machine, enter at the `awe-` prompt:

```
# Import the class GalFitTask:
from astro.recipes.GalFit import GalFitTask

task = GalFitTask(instrument='WFI', slid=57424, sids=[7,3,9,33],
                  models=[{'name:'sersic'}, {'name':'sky'}], commit=1)
task.execute()
```

the same input as above but using the cpus of the parallel cluster of computers:

```
# Import Processor class: needed to run processes on parallel cluster:
from astro.recipes.mods.dpu import Processor

# Import Environment class: needed to run processes on parallel cluster:
from common.config.Environment import Env

# Instantiate an object of class dpu: needed to run processes on parallel cluster:
dpu = Processor(Env['dpu_name'])

# Run GalFit (see previous example for explanation of parameters):
dpu.run('GalFit', i='WFI', slid=57424, sids=[1,3,9,33], m=[{'name':'sersic'},\
        {'name':'sky'}], C=1)
```

#### Working with GalFitList

The GalFitList class is intended as a simple way to group GalFitModels (and GalFitComponents). The way to use this class is to first create and commit one, and then specify its GalFitList identifier GFLID when running the GalFit task on the DPU or locally:

```
# Create the GalFitList object
l = GalFitList()
l.name = 'test-run-1'
l.make()
l.commit()

# [schmidt] 16:27:12 - Set GalFitList identifier GFLID to 100231
```

Table 22.1: The result of each profile that is fitted by Galfit is stored in a GalFitComponent subclass. Col.(1): The 2-D profile. Col.(2): the name as given as input to the task/DPU. Col.(3): the name of the class. Col.(4) names of the profile parameter names. Col.(5): description of the profile parameters.

| profile | profile name for task/DPU | class name | parameters[1] | parameter description |
|---|---|---|---|---|
| (1) | (2) | (3) | (4) | (5) |
| | | | **x** | position x [pixel] |
| | | | **y** | position y [pixel] |
| **common parameters for all functions, except Sky** | | | **posang** | position angle (PA) [Degrees: Up=0, Left=90] |
| | | | **ratio** | axis ratio (b/a) |
| | | | **shape** | diskiness/boxiness |
| De Vaucouleurs | devauc | GalFitDevauc | as Sersic (N=4) | |
| Exponential Disk | expdisk | GalFitExpdisk | as Sersic (N=1) | |
| Gaussian | gaussian | GalFitGaussian | mag | total magnitude |
| | | | fwhm | FWHM of Gaussian |
| Modified King | king | GalFitKing | mu | surface brightness |
| | | | Rc | core radius |
| | | | Rt | truncation radius |
| | | | alpha | sharpness of transition alpha |
| Moffat | moffat | GalFitMoffat | mag | total magnitude |
| | | | fwhm | FWHM of Gaussian |
| | | | pow | value of powerlaw |
| Nuker | nuker | GalFitNuker | mu | surface brightness |
| | | | Rb | radius at which mu is determined |
| | | | alpha | sharpness of transition |
| | | | beta | outer powerlaw slope |
| | | | gamma | inner powerlaw slope |
| Sersic | sersic | GalFitSersic | mag | total magnitude |
| | | | reff | effective radius ($R_e$) [pixels] |
| | | | N | sersic index (deVauc=4) |
| Sky background | sky | GalFitSky | value | sky background [ADU counts] |
| | | | grad_x | dsky/dx (sky gradient in x) |
| | | | grad_y | dsky/dy (sky gradient in y) |
| Ferrer | ferrer | GalFitFerrer[2] | | |
| Isophote | isophote | GalFitIsophote[2] | | |
| Powersic | powersic | GalFitPowersic[2] | | |
| Psf | psf | GalFitPsf[2] | | |

[1] All parameters come in four variants e.g.: x (best-fit), ix (initial value), dx (error), free_x (fit/fix)
[2] Not implemented yet

```
# Refer to the GalFitList object by specifying its GFLID, as reported
# after committing the GalFitList (see above).
dpu.run('GalFit', i='WFI', slid=75637, sids=range(10,20), gflid=100231, C=1)
```

Now it is easy to query on the group of GalFitModels you called "test-run-1" and inspect their residual images:

```
query = GalFitModel.GFLID == 100211
for model in query: model.get_residual()
```

This will create all residual images, which can then be inspected with a FITS viewer.

### 22.1.4 Querying the database for GalFitModel results

To print the results from the fit made above for source with SID=7 enter at the `awe`-prompt:

```
# Import the class GalFitComponent which is defined in GalFitModel:
from astro.main.GalFitModel import GalFitComponent

# Query the database for the GalFitComponents which contain the results
# for source with SID=3 in SourceList with SLID=57424 which
# were fitted with a sersic profile:
query = (GalFitComponent.name == 'sersic') &
        (GalFitComponent.SLID==57424) &
        (GalFitComponent.SID==3)

# Loop over the results in the query:
for c in query:
    # Print x position and the effective radius:
    print c.SID, c.GFID, c.x, c.reff, c.N, c.iN, c.iratio, c.ishape
```

To print the results from the fit made above enter at the `awe`-prompt in a script:

```
# Import the class GalFitComponent which is defined in GalFitModel:
from astro.main.GalFitModel import GalFitComponent

# Query the database for the GalFitComponents which contain the results
# for those sources in SourceList with SLID=57424 which
# were fitted with a sersic profile:
query = (GalFitComponent.name == 'sersic') & (GalFitComponent.SLID==57424)

# Loop over the results in the query:
for c in query:
    # Print x position and the effective radius:
    print c.SID, c.GFID, c.x, c.reff, c.N, c.iN, c.iratio, c.ishape
```

If you have run Galfit with different parameters various times on the same set of sources. Print the results for the last run.

```
# Define three sources by tuples of SLID, SID, here sources 31, 52, 73 from
# SourceList with SLID=57424
mysources = [(57424, 31), (57424, 52), (57424, 73)]
models = []
for slid, sid in mysources:
```

```
    q = (GalFitModel.SLID == slid) & (GalFitModel.SID == sid)
    model = q.max('GFID')
    models.append(model)
for m in models:
    m.show_model_parameters()
```

To get a listing of all parameters of a Sersic profile:

```
# Import all classes defined in GalFitModel (GalFitSersic being one of them):
from astro.main.GalFitModel import *

# Prints a listing of all parameters of a Sersic profile:
GalFitSersic.get_persistent_properties()
```

### 22.1.5 Configuring GalFitModel

While it is possible to configure GalFitModel manually, a configuration is determined automatically when no configuration is specified. The following steps can be distinguished in this process:

- Based on the Sextractor parameters of the source you derive a GalFitModel for, a region is extracted from the larger image the SourceList is made from. The size of the region is related to the semi-major axis of the source (Sextractor "A" parameter).
- Pixelscale and magnitude zeropoint are obtained from the AstrometricParameters resp. SourceList objects.
- By default a Sersic and Sky profile are fit to the modelled source.
- The initial parameters of the specified models are set based on the Sextractor parameters. For a Sersic profile the initial magnitude value (imag) is set to MAG_ISO in the SourceList. Similarly the Xpos and Ypos parameters are used to define the initial position.
- Neighbouring sources that are both close and bright enough to influence the fit are detected and assigned their own Sersic profile.

Galfit's configuration file can be considered as consisting of two separate parts. One part is a number of general configuration parameters, such as the input and output files, pixelscales etc. The other part is an arbitrarily large collection of initial values of functions (e.g. Sersic, Nuker, King) that are fit to the galaxy and potentially other sources in its neighbourhood.

#### Configuring general process parameters of GalFitModel

To get a listing of all general parameters of Galfit (which are contained in the class GalFitModel, see also table 22.2) type the following lines in the `AWE` prompt:

```
# Import the class Pars.
from astro.util.Pars import Pars

# Make an object which contains the general process parameters for GalFit.
p = Pars(GalFitModel)
p.show()
```

Table 22.2: Overview of GalFitParameters

*class GalFitParameters*

| parameter name | description | type | default/range |
|---|---|---|---|
| data | Input data image (FITS file) | str | galfitdata.fits[1] |
| model | Output data image (FITS file) | str | galfitmodel.fits[2] |
| badpixels | filename of bad pixels file | str | [3] |
| sigma | filename of sigma image | str | [4] |
| constraints | filename of constraints file | str | [5] |
| psf | filename of PSF image | str | "" |
| conv_x | size in x of convolution window | float | 0.0 |
| conv_y | size in y of convolution window | float | 0.0 |
| display | type of display (regular, curses, both) | str | regular |
| fit_xmin | subsection of input FITS file to fit | int | 0[7] |
| fit_xmax | subsection of input FITS file to fit | int | 0[7] |
| fit_ymin | subsection of input FITS file to fit | int | 0[7] |
| fit_ymax | subsection of input FITS file to fit | int | 0[7] |
| interactive | modify/create objects interactively | int | 0[8] |
| no_fit | do not fit, just output model image | int | 0 |
| pixelscale_x | pixel scale in x [arcsec/pixel] | float | 0.0[9] |
| pixelscale_y | pixel scale in y [arcsec/pixel] | float | 0.0[9] |
| subsampling | PSF fine sampling factor relative to data | float | 1.0 |
| zeropoint | photometric zeropoint | float | 0.0[10] |
| region_xmin | subsection of FITS file to extract | int | 0[11] |
| region_xmax | subsection of FITS file to extract | int | 0[11] |
| region_ymin | subsection of FITS file to extract | int | 0[11] |
| region_ymax | subsection of FITS file to extract | int | 0[11] |

[1] Dependent on SourceList.frame.filename in the Astro-WISE class hierarchy
[2] Dependent on filename of "data"
[3] Derived from weight map
[4] Derived from weight map. Caveat: ReducedScienceFrames which are not calibrated with Astro-WISE ingested, but ingested into the system as a ReducedScienceFrame by users can contain weight image formats which differ from the standard. The ACS ReducedScienceFrames are an example. Users can obtain correct sigma images in such cases by adapting GalFitModel.py.
[5] Not currently supported
[7] The entire region defined by region_* parameters is fit by default
[8] Galfit interactive mode is not supported
[9] Derived from AstrometricParameters object
[10] Derived from SourceList/sextractor input
[11] This is not a Galfit parameter, it is used in Astro-WISE to select the region in the image that was input of the SourceList

This uses the general method to show process parameters as discussed in the process parameters howto (see section 8.4) . This how-to also explains how to use this method to set parameters for Galfit. Here is nevertheless one example. To set the cut-out region around the source which will be modeled by Galfit, one should specify a dictionary:

```
p = {'GalFitModel.process_params.region_xmin': 1,
    {'GalFitModel.process_params.region_xmax': 10,
    {'GalFitModel.process_params.region_ymin': 1,
    {'GalFitModel.process_params.region_ymax': 10}
```

To get the necessary coordinates, these values have to be offset by the source position. From the source parameters retrieve the Xpos and Ypos of the source:

```
sl = (SourceList.SLID == 57424)[0]
xpos = sl.sources[10]['Xpos']
ypos = sl.sources[10]['Ypos']
```

then determine the region you want around this source and fill in as above. The dictionary specified above can be given to the DPU to change the process parameters:

```
dpu.run('GalFit', i='WFI', slid=57424, sids=[1,3,9,33], m=[{'name':'sersic'},
        {'name':'sky'}], p=p, C=1)
```

or to the Task:

```
task = GalFitTask(instrument='WFI', slid=57424, sids=[1,3,9,33],
                  models=[{'name':'sersic'}, {'name':'sky'}], pars=p, commit=1)
task.execute()
```

The following commands clarify the correspondence between the names of parameters in the original C code of Galfit and the names given in the Astro-WISE implementation. The class of a specific profile (e.g., GalFitSersic) contains the mapping of parameters specifically for that profile. The class GalFitParameters contains the mapping of general parameters.

```
# Import all classes defined in GalFitModel:
from astro.main.GalFitModel import *
GalFitParameters.CONFIG_FILE_MAP
GalFitSersic.CONFIG_FILE_MAP
```

A dictionary is returned which has as key the parameters name in the original C code and as value the name in the Astro-WISE implementation.

### Configuring initial fitting parameters of model components

To set the initial fit parameters and/or fix parameters, their names and values must be specified in the list of dictionaries given in the "models" (task level) or "m" (dpu level) argument:

```
# Task level:
task = GalFitTask(instrument='WFI', slid=57424, sids=[1,3,9,33],
                  models=[{'name':'sersic', 'iN':1.5, 'free_N':0},
                          {'name':'sky'}], commit=1)
task.execute()

# DPU level:
dpu.run('GalFit', i='WFI', slid=57424, sids=[1,3,9,33], m=[{'name':'sersic',
        'iN':1.5, 'free_N':0}, {'name':'sky'}], C=1)
```

### Constraining the fit parameters

Fit parameters can be constrained in 4 different ways by Galfit. These different constraints are represented by 4 Python classes:

- *GalFitAbsConstraint:* This constraint constrains a parameter between two values.
- *GalFitRelConstraint:* This constraint constrains a parameter between a range around the initial value.
- *GalFitDiffConstraint:* This constraint constrains the difference between the same parameter for two different objects/components.
- *GalFitRatioConstraint:* This constraint constrains the ratio of the same parameter for two different objects/components.

A list of constraints can be specified in the task as follows:

```
# Task level:
task = GalFitTask(instrument='WFI', slid=57424, sids=[1,3,9,33],
                  constraints=[{'name':'abs', 'comp':1, 'param': 'x',
                                'min':13, 'max':13}], commit=1)

# DPU level:
dpu.run('GalFit', i='WFI', slid=57424, sids=[1,3,9,33], cs=[{'name':'abs',
        'comp':1, 'param': 'x', 'min':13, 'max':13}], C=1)
task.execute()
```

In other words, a constraint can be defined in a dictionary, which maps the properties of the constraint object. A list of such dictionaries represents a list of constraints.

### Using a PSF image

In the configuration file of Galfit a PSF image filename can be specified. Galfit uses this image to convolve the model before comparing it to the data. PSF image files are represented by the *PSFImage* class.

There are two ways to create PSFImages:

1 Use TinyTim to create it. TinyTim can only be used to create PSF images for the ACS wide-field camera of the HST. See the TinyTim howto (section 22.2).

2 Use an existing PSF image file and store it.

Here is how you can store an existing PSF image:

```
p = PSFImage(pathname='my_psf_image.fits')
p.make()
p.store()
p.commit()
```

This will update the filename of the PSF image to be unique.

Now using the stored PSFImage is done by specifying its filename in the appropriate process parameter. The presence of the filename in the process parameters object triggers a query to find the specified file in the database:

```
task = GalFitTask(instrument='ACS', slid=110521, sids=[50],
                  pars={'GalFitModel.process_params.psf': 'Cal-EHELMICH-----
                  --------TinyTim---PSF-54227.5867871-83e1bd5c6e15be667a
                  3cc511ccca13a6ee043514.fits'}, commit=1)
task.execute()
```

### 22.1.6 Description of useful methods of GalFitModel

- get_model()

  Creates the model image and returns it as a BaseFrame object.

- get_residual()

  Creates the residual image and returns it as a BaseFrame object.

- get_science()

  Extracts and downloads the region in the science image for which the model was derived, and returns it as a BaseFrame object.

- get_weight()

  Extracts and downloads the region in the weight image for which the model was derived and returns it as a BaseFrame object.

- show_model_parameters()

  Display a list of all ellipse parameters.

- get_model_parameters()

  Returns a list of dictionaries of all components. I.e. each item of the list is a dictionary which contains the description of one GalFitComponent.

### 22.1.7 Caveats

- Automatically generated sigma images incorrect.

  ReducedScienceFrames which are not calibrated with Astro-WISE ingested, but ingested into the system as a ReducedScienceFrame by users can contain weight image formats which differ from the standard. The ACS ReducedScienceFrames are an example. Users can obtain correct sigma images in such cases by adapting GalFitModel.py.

## 22.2 HOW-TO Use TinyTim

The TinyTim package (http://www.stsci.edu/software/tinytim/tinytim.html) is used to create PSF images for the instruments of the Hubble Space Telescope. In Astro-WISE only the ACS instrument for the WFC detectors for Hubble is supported, and hence the use of TinyTim is limited to that configuration.

TinyTim images are used in particular when running surface photometry tools such as Galfit.
When running TinyTim several choices must be made:
The meaning of the configurable parameters and their defaults:

- **Av=None or embv=None** Interstellar extinction of x magnitudes is applied to the object spectrum, specified in the form of E(B-V) or visual extinction (Av). Do not specify both.
- **angle=None, see also major=None, minor=None** Convolve the PSF with a elliptical Gaussian kernel approximating x milliarcseconds RMS of jitter along its major axis, y mas jitter along the minor axis, with the major axis rotated by z degrees from the +X image axis. All three parameters must be specified if any of them is.
- **chip=1** For which of the WFC ccds should tinytim make a PSF model chip 1 or 2.
- **coeff=2000.0** Depending on the value of **spectrum**, if spectrum=2 (blackbody) this is the Temperature in Kelvin of the black body, if spectrum=3 (Power law) this is the powerlaw coefficient.
- **diameter=3.0** Diameter of PSF in arcsec (maximum=25.7arcsec). Recommended size=3.0arcsec.
- **ebmv=None or Av=None** Interstellar extinction of x magnitudes is applied to the object spectrum, specified in the form of E(B-V) or visual extinction (Av). Do not specify both.
- **filename=psf_image**: DO NOT CHANGE this default
- **filter=F555W**: name of filter to model.
- **instrument=15**: DO NOT CHANGE this default
- **jitter=None** Convolve the PSF with a symmetrical Gaussian kernel approximating x milliarcseconds RMS of jitter.
- **major=None, see also minor=None, angle=None** Convolve the PSF with a elliptical Gaussian kernel approximating x milliarcseconds RMS of jitter along its major axis, y mas jitter along the minor axis, with the major axis rotated by z degrees from the +X image axis. All three parameters must be specified if any of them is.
- **minor=None, see also major=None, angle=None** Convolve the PSF with a elliptical Gaussian kernel approximating x milliarcseconds RMS of jitter along its major axis, y mas jitter along the minor axis, with the major axis rotated by z degrees from the +X image axis. All three parameters must be specified if any of them is.
- **paramfile=tiny.conf**. DO NOT CHANGE this default.
- **spectrum=2**. The only SUPPORTED VALUES here are **2** and **3** from the following list:
  - 1) Select a spectrum from list
  - 2) Blackbody

  - 3) Power law : F(nu) = nu$^i$
  - 4) Power law : F(lambda) = lambda$^i$
  - 5) Read user-provided spectrum from ASCII table

- **wmag=None** Increase or decrease by a factor of x the default number of wavelengths used in computing a polychromatic PSF (ie. wmag=0.5 would use only half the default number of wavelengths, while wmag=2 would double the number, more finely sampling the response curve). Increasing wmag provides a somewhat smoother PSF at radii beyond about 3. A minimum of one wavelength will be used.
- **x=-1** Enter position x on detector in INTEGER pixels (X range = 0-4095)
- **y=-1** Enter position y on detector in INTEGER pixels (Y range = 0-2047)

An object model was made to store these choices and run TinyTim.

### 22.2.1 Running TinyTim

There is a DBRecipe to make TinyTim. The only arguments it takes is a dictionary specifying the process parameters and a switch to commit the result to the database and dataserver:

```
awe> task = TinyTimTask(pars={'TinyTimImage.tinytimconf.x': 200,
                              'TinyTimImage.tinytimconf.y': 250}, commit=1)
```

The configurable parameters can be shown as conventional, using the "Pars" class:

```
awe> p = Pars(TinyTimImage)
awe> p.show()
etc.
```

## 22.3 HOW-TO use Galphot in Astro-WISE

### 22.3.1 Introduction

Galphot is a surface photometry tool, which fits ellipses to isophotes in galaxy profiles. It was written by Marijn Franx and is available for download from his website. The version on his website only works by using IRAF; for Astro-WISE a number of changes were done to make it work outside of IRAF.

### 22.3.2 Astro-WISE implementation

The main classes in the Galphot object model for Astro-WISE (i.e. those classes that are stored in the database, and must be queried on to get results) are these:

- *GalPhotModel*: The main Galphot class. Conceptually this **is** the model of a single galaxy. The model image is stored on the dataservers and can be retrieved. GalPhotModel contains methods to create the residual image etc. The model also contains a list of GalPhotEllipses.
- *GalPhotEllipse*: This is the equivalent of one line in the output table produced by Galphot. A GalPhotModel contains a list of GalPhotEllipses.
- *GalPhotParameters*: This class stores the Galphot configuration parameters (see table 22.3).
- *GalPhotList*: This class is used as tool to link any group of GalPhotModels together through an identfier and optionally a name.

Each GalPhotModel is identified by a unique number (GPID). This also connects the GalPhotEllipses to the GalPhotModel they belong to.

GalPhotModels are made based on SourceLists, so in order to run Galphot within Astro-WISE one first has to make a SourceList and determine which source one wants to model with Galphot. This source is then identified with the identifier combination SLID, SID, which uniquely defines one source in a SourceList.

### 22.3.3 First step: making a SourceList or querying existing SourceLists

First we must decide which source to run galphot on. This means either making a new SourceList (running Sextractor) or querying existing SourceLists. How to make SourceLists is described in section 21.6 For now we assume you know which SourceList you have made, and now want to find the SIDs of suitable galaxies.

```
# Query for SourceList with identifier 57424
sl = (SourceList.SLID == 57424)[0]

# This will be the output dictionary
d = {'SID':[], 'MAG_ISO':[], 'A':[], 'B':[], 'E_WCS':[]}

# Do a query to find bright galaxy-like objects
r = sl.sources.sql_query(d, '"MAG_ISO"<20.0 AND "E_WCS"<0.5')
```

At this point "d" contains the list of identifiers, magnitudes and A, B of the sources in the SourceList that matched the query. The list "d['SID']" can be given as the "sids" argument to the GalPhotTask (see the next section).

### 22.3.4 Running Galphot; using the GalPhotTask

Here is how to create a GalPhotModel for the sourcelist with ID 57424 and source with ID 71 within it:

```
task = GalPhotTask(instrument='WFI', slid=57424, sids=[71], commit=1)
task.execute()
```

or equivalently using the DPU:

```
dpu.run('GalPhot', i='WFI', slid=57424, sids=[71], C=1)
```

### 22.3.5 Configuring GalPhotModel

Galphot can be configured with the *Pars* class as conventional in Astro-WISE :

```
p = Pars(GalPhotModel)
p.show()
```

this results in a list of configurable parameters (see table 22.3), now set one of them:

```
p.GalPhotModel.process_params.r1 = 10.0
```

Feed the configuration to the task using the "get()" method of Pars:

```
task = GalPhotTask(instrument='WFI', slid=57424, sids=[71], pars=p.get(),
                   commit=1)
task.execute()
```

The process parameter dictionary can also be specified by hand directly:

```
d = {'GalPhotModel.process_params.r1': 10.0,
     'GalPhotModel.process_params.r2': 20.0}

task = GalPhotTask(instrument='WFI', slid=57424, sids=[71], pars=d, commit=1)
task.execute()
```

### 22.3.6 Masking other sources in the field

The following pixels/sources may be masked:

- Pixels with a weight of 0 (always).
- Rectangles and/or circles specified in a mask file.

  *The GalPhotTask can be given a list of mask file filenames. This only works when working locally; no files are uploaded to the DPU. See below.*
- Sources other than the primary source that are in the cutout may be automatically masked. This is done on the basis of either the SourceList or the SExtractor "segmentation" check image.

  *This is controlled with the process parameters "mask_type" and "mask_scale". See below.*

Table 22.3: Overview of GalPhotParameters

*class GalPhotParameters*

| **parameter** | **description** | **type** | **default** |
|---|---|---|---|
| ngal | Number of galaxies to model, fixed to 1 | int | 1 |
| iter1 | Number of iterations without calculating residuals | int | 3 |
| iter2 | Number of iterations with calculating residuals | int | 3 |
| hmax | Maximum number of harmonicals | int | 6 |
| nsammax | Maximum number of samples along the ellipse | int | 200 |
| debug | Debug parameter | int | 0 |
| npolres | Order in interpolation in intensity | int | 4 |
| rmin | Minimum radius | float | 0.5 |
| rmax | Maximum radius | float | 75.0 |
| radfac | Radial scaling factor | float | 1.1 |
| rshap | Maximum radius for modifying ellipticity and PA | float | 10.0 |
| rcen | Maximum radius for modifying centers | float | 10.0 |
| errshap | Maximum error in shape of ellipse | float | 0.5 |
| errcen | Maximum error in position of ellipse | float | 0.1 |
| dposmax | Maximum change in center per iteration | float | 0.1 |
| dellmax | Maximum change in ellipticity per iteration | float | 0.1 |
| dangmax | Maximum change in position angle per iteration | float | 0.1 |
| r1 | Maximum radius for single annulus sampling | float | 20.0 |
| r2 | Maximum radius for multiple annulus sampling | float | 0.0 |
| fracmin | Minimum fraction of good points along the ellipse | float | 0.4 |
| cliplow | Fraction of points to clip at low end | float | 0.1 |
| cliphigh | Fraction of points to clip at high end | float | 0.1 |
| linear | Linear scaling | int | 0 |
| extend | Extend radial interval after fit | int | 1 |
| outfill | Calculate residual outside of galaxy | int | 1 |
| xpos | X-position (in region) of galaxy to model | float | -1.0 |
| ypos | Y-position (in region) of galaxy to model | float | -1.0 |
| region_xmin | Minimum x-coordinate of cutout region in larger frame | int | -1[1] |
| region_xmax | Maximum x-coordinate of cutout region in larger frame | int | -1[1] |
| region_ymin | Minimum y-coordinate of cutout region in larger frame | int | -1[1] |
| region_ymax | Maximum y-coordinate of cutout region in larger frame | int | -1[1] |
| region_scale | Factor which determines the size of the region | float | 12.0[2] |
| mask_areas | List of GalPhotMask (circles, squares) objects | list | [][3] |
| mask_type | "sourcelist", "segmentation" or ""for masking | str | segmentation[3] |
| mask_scale | Semi-major axis scale factor while masking | float | 3.0[3] |

[1]This is not a Galphot parameter, it is used in Astro-WISE to define the region extracted from SourceList.frame
[2]This is not a Galphot parameter, instead it is used to define the region_* parameters
[3]This is not a Galphot parameter, instead it is used to write the "deletion" file

**Automatic masking**

The automatic masking process is controlled by the following process parameters:

- **mask_type**: Values: 'segmentation', 'sourcelist' or ''. If the value is **segmentation** the segmentation check image created along with each SourceList will be used to identify which pixels belong to other sources than the modelled source and they will be masked. If the value is **sourcelist**, the sourcelist is scanned for nearby and bright sources, and a circle of size mask_scale*A is used to mask such sources. If the value is '' (empty string) no masking of nearby sources will be performed.
- **mask_scale**: See above.

**Manual masking**

A mask file (".del" file, in galphot convention) can be created by the user. This text file contains lines of either 3 or 4 numbers, describing circles and rectangles respectively. Example:

```
25 28 10      # Describes a circle at position (25,28) with radius 10 pixels.
20 30 35 45   # Describes the rectangle [20:30, 35:45]
```

Note that the pixel coordinates are in the coordinate frame of the original image, rather than the cutout region. If such a file is created it can be specified when running the task:

```
task = GalPhotTask(instrument='WFI', slid=155211, sids=[17044],
                   mask_files=['masks.txt'], commit=0)
```

If a mask file is specified, one must be given for each SID specified in the "sids" list.

### 22.3.7 Using an existing model as initial values

It is possible to use an existing GalPhotModel as input, i.e. as an "intable", in the Galphot configuration. This is done by specifying the existing GalPhotModel's GPID in the "gpid_in" option of the task:

```
task = GalPhotTask(instrument='WFI', slid=57424, sids=[71], gpid_in=6721,
                   commit=1)
task.execute()
```

or equivalently in the dpu call:

```
dpu.run('GalPhot', i='WFI', slid=57424, sids=[71], gpid_in=6721, C=1)
```

### 22.3.8 Using GalPhotList

The GalPhotList class is intended as a simple way to group GalPhotModels (and GalPhotEllipses). The way to use this class is to create it first, and only then run Galphot. I.e. first create and commit a GalPhotList, and then specify its GalPhotList identifier GPLID when running the GalPhot task on the DPU or locally:

```
# Create the GalPhotList object
l = GalPhotList()
l.name = 'test-run-1'
l.make()
```

```
l.commit()

# [schmidt] 16:27:12 - Set GalPhotList identifier GPLID to 100231

# Refer to the GalPhotList object by specifying its GPLID, as reported
# after committing the GalPhotList (see above).
dpu.run('GalPhot', i='WFI', slid=75637, sids=range(10,20), gplid=100231, C=1)
```

Now it is easy to query on the group of GalPhotModels you called "test-run-1" and inspect their residual images:

```
query = GalPhotModel.GPLID == 100211
for model in query: model.get_residual()
```

This will create all residual images, which can then be inspected with a FITS viewer.

### 22.3.9 Querying for results

When you have made a GalPhotModel, the ellipse parameters are stored in the database as a list of GalPhotEllipse objects, and the residual image is stored on a dataserver. How does one get to this information?

```
# Query for GalPhotModel based on its GPID

q = GalPhotModel.GPID == 20
model = q[0]

# Query for GalPhotModel based on SLID/SID

q = (GalPhotModel.SLID == 57424) & (GalPhotModel.SID == 71)
model = q.max('GPID')

# Find all GalPhotEllipses (radii) for the source defined by SLID=57424 and
# SID=71, for which the fitted intensity is larger than 50000 (ADU)

q = (GalPhotEllipse.SLID == 57424) & (GalPhotEllipse.SID == 71) &\
    (GalPhotEllipse.i > 50000)
GPIDs = [ellipse.GPID for ellipse in q]
gpids = []
for gpid in GPIDs:
    if not gpid in gpids:
        gpids.append(gpid)

models = []
for gpid in gpids:
    m = (GalPhotModel.GPID == gpid)[0]
    models.append(m)

# Now we can have a look at the residual images:

for m in models:
    m.retrieve()
```

Table 22.4: Overview of GalPhotEllipse parameters

*class GalPhotEllipse*

| parameter | description | unit |
|---|---|---|
| r, dr | The root of the product of the major and minor axis of the ellipse | pixel |
| i, di | Intensity at the ellipse | counts/pixel$^2$ |
| s, ds | The slope of the intensity, the derivative of i wrt r | None |
| x, dx | Central x position in the region | pixel |
| x_orig | Central x position in the original image | pixel |
| y, dy | Central y position in the region | pixel |
| y_orig | Central y position in the original image | pixel |
| eps, deps | The ellipticity of the ellipse, 1-b/a | None |
| pos, dpos | The position angle of the ellipse, measured with respect to the axis y=0, counter clockwise | None |
| c1, c2, ..., c6 | The cos(n*theta) term of the residuals along the ellipse | None |
| dc1, dc2, ..., dc6 | Errors in c1, c2, ..., c6 | None |
| s1, s2, ..., s6 | The sin(n*theta) term of the residuals along the ellipse | None |
| ds1, ds2, ..., ds6 | Errors in ds1, ds2, ..., ds6 | None |
| f1, f2, f3, f4 | Flag? | None |

```
    m.display()

# And plot ellipse parameters against eachother:

for m in models:
    x = [ellipse.i for ellipse in m.ellipses]
    y = [ellipse.r for ellipse in m.ellipses]
    pylab.scatter(x,y)
```

### 22.3.10 Description of useful public methods of GalPhotModel

- get_model()

  Downloads the model image and returns it as a BaseFrame object.

- get_residual()

  Creates the residual image and returns it as a BaseFrame object.

- get_science()

  Extracts and downloads the region in the science image for which the model was derived, and returns it as a BaseFrame object.

- get_weight()

  Extracts and downloads the region in the weight image for which the model was derived and returns it as a BaseFrame object.

- show_model_parameters()

  Display a list of all ellipse parameters.

- get_model_parameters()

  Returns a list of dictionaries of all ellipse parameters. I.e. each item of the list is a dictionary which contains the description of one ellipse. See table 22.4 for a description of the parameters.

## 22.4 HOW-TO: Photometric redshifts

The photred code (Bender et al. 2001 (2001defi.conf...96B) and Gabasch et al. 2004 (2004A&A...421...41G)) provides a way to derive redshifts from photometric observations.

> **WARNING**: The code currently uses the `MAG_ISO` fluxes and magnitudes from the SExtractor catalogs. For `SourceLists` generated from images with different seeing, the colors will be inaccurate, and the derived photometric redshifts as well.

The user interface consists of two major Python classes, `PhotRedConfig` and `PhotRedCatalog`. The `PhotRedConfig` contains the information about the filters and SEDs used. In `PhotRedCatalog` the used `SourceLists`, the two resulting `SourceLists` with the best-fitting stellar and galactic SEDs and redshifts are stored, as well as an `AssociateList` of all these lists to allow easy access to the complete information about a given object.

```
awe> from astro.main.PhotRedCatalog import *
```

imports all the needed classes from PhotRedCatalog into the current `AWE` session.

### 22.4.1 PhotRedConfig

The `PhotRedConfig` class needs some input calibration files that are stored on the dataserver. In most cases these should be already available in the system. If not, at the end of this HOW-TO, instructions are given on how to import Filters and SEDs.

#### PhotRedFilter

The `PhotRedFilter` stores the transmission curve of the filter and is associated with an Astro-WISE Filter object.

```
awe> filt = (Filter.name == '#842' )[0]
awe> pf = (PhotRedFilter.filter == filt )[0]
```

#### PhotRedSED

A `PhotRedSED` object stores the unprocessed SED of a model galaxy, a table of wavelengths and fluxes, and is referenced by a name given on import, e.g. 'mod_e' for an elliptical galaxy.

```
awe> pse = (PhotRedSED.sed_name == 'mod_e.sed' )[0]
awe> ps1 = (PhotRedSED.sed_name == 'mod_s210.sed' )[0]
```

#### PhotRedStarlib

The `PhotRedStarlib` stores a collection of `PhotRedSEDs` of different stars. It is referenced by the filename of the list of SED names.

```
awe> starlib=(PhotRedStarlib.filename=='starlib_pickles.lis')[0]
```

PhotRedConfig

The PhotRedConfig takes a combination of SEDs and filter names, creates the processed SEDs and Starlib and stores these and other information relevant for PhotRed itself. A PhotRedConfig object is referenced by a unique name, given at creation. This allows the reuse of an existing PhotRedConfig object.

The number of filters selected at creation time can be larger than the number of filters used for PhotRed (as long as the filters of all the SourceLists given to PhotRedCatalog are present in the configuration. Thus it is advisable to create the PhotRedConfig with all PhotRedFilters available, making the combination of SEDs reusable with different combinations of SourceLists.

```
awe> pc = PhotRedConfig()
awe> pc.SEDs=[pse,ps1,ps2,ps3]
awe> pc.filters=[pfu,pfb,pfv,pfr,pfi]
awe> pc.starlib=(PhotRedStarlib.filename=='starlib_pickles.lis')[0]
awe> pc.name='MyPhotRedConfig'
awe> pc.make()
```

### 22.4.2 PhotRedCatalog

The PhotRedCatalog is the main component, doing the actual work determining the best fitting stellar / galactic SED. It reads in the SEDs processed by PhotRedConfig, redshifts them and calculates the least-squares fit of the magnitudes obtained from the combination of the redshifted SEDs and the filter curves against the observed data. The least-squares fit is determined by minimizing:

$$\chi^2(z, SED) = \frac{1}{N_{filt}} \sum_{i=1}^{N_{filt}} \frac{\left[f_i - \alpha f_i\left(z, SED\right)\right]^2}{\sigma_i^2 + \left[0.005\alpha f_i\left(z, SED\right)\right]^2}$$

The probability $P_T$ of a source being at a given redshift is calculated as:

$$P_T = P_\chi \cdot P_L \cdot P_z = e^{-\frac{1}{2}\chi^2} \cdot e^{-k_\beta\left(\frac{M-M_\star}{\sigma}\right)^\beta} \cdot e^{-k_\gamma\left(\frac{z}{z_{lim}}\right)^\gamma}$$

As input it takes the SourceLists and PhotRedConfig. The pr.master stores the master SourceList against which the association of the other SourceLists is done. It has to be in the SourceLists array as well, to allow correlation between the SourceLists, the extinction and model_error arrays.

```
awe> pr = PhotRedCatalog()
awe> pr.config=pc
awe> pr.master=sV
awe> pr.sourcelists=[sU,sV,sB,sR,sI]
```

Additionally, individual values for extinction (used for relative corrections of the photometric calibration of the individual SourceLists) and model_errors (to allow some spread between the distinct SEDs) can be specified. The length and the order of the arrays must in both cases be the same as the length iof the list of sourcelists.

```
awe> pr.extinc=[0.,0.,0.,0.,0.]
awe> pr.model_error=[0.,0.,0.,0.,0.]
```

The PhotRedCatalog object can be assigned a name for future reference. The resulting SourceLists are created and stored in the database by calling the make function:

```
awe> pr.name='FDF_UBRI'
awe> pr.make()
```

### 22.4.3 The output SourceLists

The resulting SourceLists are stored under in the PhotRedCatalog object as:

```
awe> pr.datpz1  # Data of best-fitting galactic SED
```

and

```
awe> pr.datstar # Data of best-fitting stellar SED
```

and associate with the master AssociateList.

### 22.4.4 The visualization routines

For visual inspection a plot of the best-fitting SED, the best-fitting stellar SED, the datapoints and the redshift probability distribution can be done by calling:

```
awe> pr.plot( 23 ) # For the object with AID 23 in associate_list
```

### 22.4.5 An example from users view

Import the needed Python classes:

```
awe> from astro.main.PhotRedCatalog import *
```

#### PhotRedConfig

Create a configuration. This is only needed if no suitable configuration is present in the system, because existing configurations could and should be reused. First select the respective PhotRedFilter objects from the database,

```
awe> pf1 = (PhotRedFilter.filename == '#843.filter' )[0]
awe> pf2 = (PhotRedFilter.filename == '#844.filter' )[0]
awe> pf3 = (PhotRedFilter.filename == '#846.filter' )[0]
awe> pf4 = (PhotRedFilter.filename == '#878.filter' )[0]
awe> pf5 = (PhotRedFilter.filename == '#879.filter' )[0]
```

then select the SEDs you want to use from the database,

```
awe> ps01 = (PhotRedSED.filename == 'manucci_soc.sed' )[0]
awe> ps02 = (PhotRedSED.filename == 'manucci_sac.sed' )[0]
awe> ps03 = (PhotRedSED.filename == 'manucci_sbc.sed' )[0]
awe> ps04 = (PhotRedSED.filename == 'mod_e.sed' )[0]
awe> ps05 = (PhotRedSED.filename == 'mod_s010.sed' )[0]
awe> ps06 = (PhotRedSED.filename == 'mod_s020.sed' )[0]
awe> ps07 = (PhotRedSED.filename == 'mod_s030.sed' )[0]
...
```

and create the PhotRedConfig object using the standard starlib.

```
awe> pc = PhotRedConfig()
awe> pc.SEDs=[pse,ps1,ps2,ps3]
awe> pc.filters=[pfu,pfb,pfv,pfr,pfi]
awe> pc.starlib=(PhotRedStarlib.filename=='starlib_pickles.lis')[0]
awe> pc.name='WFI_BgRIz'
awe> pc.make()
```

`PhotRedCatalog`

To create the photometric redshifts, a `PhotRedConfig` object and a list of `SourceLists` is needed. First select the `PhotRedConfig` and the `SourceLists` from the database:

```
awe> pc=(PhotRedConfig.name==’PhotRedConfig-1114174946.26’)[0]
awe> sU=(SourceList.SLID==5)[0]
awe> sB=(SourceList.SLID==6)[0]
awe> sV=(SourceList.SLID==7)[0]
awe> sR=(SourceList.SLID==8)[0]
awe> sI=(SourceList.SLID==9)[0]
```

With these, the `PhotRedCatalog` object can be created. Using the V-Band data (in this example) as the master `SourceList` and the MAG_APER magnitudes, only objects detected in all 5 filters are associated.

```
awe> pr = PhotRedCatalog()
awe> pr.config=pc
awe> pr.master=sV
awe> pr.sourcelists=[sU,sV,sB,sR,sI]
awe> pr.extinc=[0.,0.,0.,0.,0.]
awe> pr.model_error=[0.1,0.1,0.1,0.1,0.1]
awe> pr.min_num_sources=5
awe> pr.mag=’MAG_APER’
awe> pr.flux=’FLUX_APER’
awe> pr.fluxerr=’FLUXERR_APER
awe> pr.name=’photoz_1’
awe> pr.make()
```

Once the object is made, all data is stored in the `pr.associate_list AssociateList`.

### 22.4.6 Ingestion of Filters and SEDs

The ingestion of new / additional SEDs or filter lightcurves is a straightforward process.

`PhotRedFilter`

The input file for `PhotRedFilter` is a simple ASCII file with two columns, wavelength in Ångstroms and the transmission of the filter at this wavelength. The values should be sorted with ascending wavelength, and to avoid possible problems contain 2 lines with 0 transmission at the beginning and at the end. The canonical extension for these objects is ”.filter”.

To create the Python object, a new `PhotRedFilter` object pointing to the file on disk is created, and the corresponding `Filter` object is selected. After this the make routine stores the relevant metadata in the database and the filter curve object on the dataserver.

```
awe> photredfilter = PhotRedFilter( pathname=’wfi_r.filter’)
awe> photredfilter.filter=(Filter.mag_id==’Cousins R’)
awe> photredfilter.make()
```

`PhotRedSED`

The input file for `PhotRedSED` is a simple text file as well, again with two columns, wavelength in Ångstroms and the normalized flux of the SED. A new `PhotRedSED` object pointing to the file on disk is created, and the object's `make` method is invoked. The metadata is again stored in the database, and the file stored on the dataserver.

```
awe> photredsed = PhotRedSED( pathname='mod\_e.sed')
awe> photredsed.make()
```

## 22.5 HOW-TO: MDia

### 22.5.1 Introduction

The Munich Difference Imaging Analysis (MDia) package is a tool for photometry in (very) crowded fields. The software is written in C++ and part of the "mupipe" package, which has been developed at the University Observatory in Munich for the use in a pixel-lensing project. The underlying algorithm has been proposed by Alard and Lupton in 1998 (ApJ...503..325A) and later improved by Alard in 2000 (A&AS..144..363A).

### 22.5.2 Astro-WISE implementation

The MDia pipeline has been incorporated into Astro-WISE by J. Koppenhöfer. This is done by providing a Python wrapper around the C++ programs which stores the input and output to the Astro-WISE database.

Creating light curves with MDia is a two step process. The first step is the creation of a `ReferenceFrame` using the best seeing images. In the second step, this `ReferenceFrame` is used to create difference images of all `RegriddedFrames` one wants to analyze. Using the difference images, precise photometry is obtained via PSF-fitting. In this way, light curves are created and stored in the database for further analysis.

### 22.5.3 Compiling and installing the C++ code

The MDia code can be found in:

`opipe/Experimental/MDIA/MDia.tar.gz`

Extract this archive to whatever directory you like. After that follow the instructions given in the README file. Up to now, the code is tested for 32-bit machines only. The MDia code needs the library "ltl" which is usually installed during the `awetomatic` process.

### 22.5.4 Creating a `ReferenceFrame`

The `ReferenceFrame` is created using the images (typically 10) with the best seeing. These can be selected by using the `psf_radius` property of any `ReducedScienceFrame` or `RegriddedFrame`. Try to reject images with high background in order to get an optimized $S/N$ in the `ReferenceFrame` (very often cloudy images have a small `psf_radius`, so take care!!!).

In order to get good results the astrometry of all images you want to combine must be as good as possible. Therefore use global astrometry on the best seeing images and regrid again if needed. Having prepared the final list of `RegriddedFrames` you can create a `ReferenceFrame` in three different ways:

1. using the DPU:

```
awe> dpu.run( 'Reference', i='WFI',
...           reg_filenames=['Sci-USER-WFI-#844-Reg-54653.3.fits',...], C=1 )
```

2. locally using a task:

```
awe> from astro.recipes.Reference import *
awe> task = ReferenceTask( reg_filenames=['Sci-USER-WFI-#844-Reg-54653.34244.fits',...],
...                        commit=0)
awe> task.execute()
```

3. locally using the `make()` method of class `ReferenceFrame`:

```
awe> from astro.main.ReferenceFrame import *
awe> ref = ReferenceFrame()
awe> ref.OBJECT = 'OTSF-1c'
awe> ref.regridded_frames = best_frames
awe> ref.make()
awe> ref.store()
awe> ref.commit()
```

where `best_frames` is a list of filenames of the `RegriddedFrames` you want to combine.

### 22.5.5 Creating Lightcurves

After having created a `ReferenceFrame`, one can use it together with a number of `RegriddedFrames` and create light curves of multiple sources simultaneously. The database object of use is an instance of class `MDia` which takes as input the `ReferenceFrame`, a list of `RegriddedFrames`, optionally some `SourceLists` and a set of process parameters (`MDiaParameters`). The output consists of $N$ lightcurves in ASCII or FITS format with $N$ being the number of sources analyzed.

Again, the astrometric accuracy is crucial in this step. The recommended procedure is the following:

- First create a SourceList of the `ReferenceFrame`.
- Then use this `SourceList` as a replacement for the USNO catalog and (re-)calculate the `AstrometricParameters` for all images to be analyzed. Finally, regrid all the individual images using the improved `AstrometricParameters`. By doing it this way, a very high relative astrometric precision between the `ReferenceFrame` and the `RegriddedFrames` is achieved.

To create lightcurves in `AWE` use one of the following ways:

1. using the DPU:

```
awe> dpu.run( 'MDia', i='WFI', ref_filename=['Sci-USER-WFI-#844-Ref-54653.3.fits'],
...           reg_filenames=['Sci-USER-WFI-#844-Reg-54653.3.fits',...], C=1 )
```

2. locally using a task:

```
awe> from astro.recipes.MDia import *
awe> task = MDiaTask( ref_filename=['Sci-USER-WFI-#844-Ref-54653.3.fits'],
...                   reg_filenames=['Sci-USER-WFI-#844-Reg-54653.34244.fits',...],
...                   commit=0)
awe> task.execute()
```

3. locally using the `make()` method of class `LightCurve`:

```
awe> from astro.main.LightCurve import *
awe> my_lightcurves = LightCurve()
awe> my_lightcurves.reference_frame  = reference_frame
awe> my_lightcurves.regridded_frames = frames
awe> my_lightcurves.make()
awe> my_lightcurves.store()
awe> my_lightcurves.commit()
```

where `reference_frame` is a list with one item, namely the filename of the `ReferenceFrame` and `frames` is a list of filenames of the `RegriddedFrames` you want to analyze.

## 22.6 Documentation

A manual with detailed description of all process parameters as well as the underlying software will be avialable soon on the Astro-WISE web pages.

## 22.7 HOW-TO: VODIA

### 22.7.1 Introduction

VODIA (**V**ST **O**megaCAM **D**ifference **I**mage **A**nalysis) is a package optimized to detect small photometric variations in crowded fields. The software is based on the DIA package (**D**ifference **I**mage **A**nalysis, written by Woźniak 2000, (Acta Astron. 50, 421; see also Woźniak et al. 2002, Acta Astron. 52, 129) that makes use of the "optimal PSF matching algorithm" with a space-varying kernel (Alard & Lupton 1998, ApJ 503, 325; Alard 2000, A&AS 144, 363). The software includes 6 (+1) independent C programs (corresponding to 6 different steps) and does not use any external library. The inspect method makes use of a fortran program for preliminary analysis of the results. The different steps are:

**1. wcs2pix**
The astrometric solution is used to aligne (and cut accordingly) the scientific images.

**2. Mstack**
To obtain the reference frame from the stacking of the best seeing images.

**3. Getpsf**
Extract PSF from Reference to extract PSF photometry from the difference images.

**4. Aga**
A convolution kernel is found and used to create the difference images.

**5. Getvar**
Detects variable candidates.

**6. Phot**
Produces light curves of variable candidates.

**6a. PhotRef**
Produces light curves of all objects detected in the reference image (the user can choose between Phot and PhotRef depending on the scientific goal).

### 22.7.2 Astro-WISE implementation

VODIA has been incorporated into Astro-WISE by A. Volpicelli and F. Getman. This is done by providing a Python wrapper around the C programs which stores the input and output to the Astro-WISE database/data storage. One `FORTRAN` program is used only by the `inspect` method for preliminary analysis of the results.

### 22.7.3 Compiling and installing the C code

The VODIA code can be found in Astro-WISE CVS at:

`opipe/Experimental/VODIA/VODIA.tar.gz`

Extract this archive to whatever directory you like. To compile run '`make`'. To install: '`make install`'. Installation will copy binaries to user bin directory or you can copy them to system directory manually. Up to now, the code is tested for 32-bit machines only.

### 22.7.4 Running VODIA

```
[import all python scripts]

awe> from astro.main.VariabilityFrame import VariabilityFrame
awe> from astro.external import Dia
awe> from astro.main.DiaConfig import DiaMstackConfig,DiaGetpsfConfig,
     DiaAgaConfig,DiaGetvarConfig,DiaPhotConfig

[select science frames]

awe> q = ( RegriddedFrame.OBJECT.like('projectname')) &\
         ( RegriddedFrame.filter.name == '#842' ) &\
         ( RegriddedFrame.chip.name == 'ccd50')

awe> regs = [reg for reg in q]

[creates variability object]

awe> v = VariabilityFrame()

[put our list of images as input]

awe> v.regridded_frames = regs

[create the list of images to produce the reference (optional)]
awe> q = ( RegriddedFrame.OBJECT.like('projectname')) & \
         ( RegriddedFrame.filter.name == '#842' ) &\
         ( RegriddedFrame.chip.name == 'ccd50') &\
         ( RegriddedFrame.psf_radius < 0.6 )
awe> v.regridded_frames_for_reference=list(q)

[run VODIA (all programs automatically)]

awe> v.make()

[to interactively check the results]

awe> v.inspect()

[Example of changing one parameter (C_MIN=detection threshold) and run again
getvar and phot]

awe> print v.getvar_config.C_MIN
awe> v.getvar_config.C_MIN = 0.9
awe> v.make_getvar()
awe> v.make_phot()

[check again new results]

awe> v.inspect()
```

```
[storing the file with the light curves]

awe> v.store()

[commit the results to database]

awe> v.commit()
```

The output consists of one ASCII file with N lightcurves (N being the number of sources analyzed).

### 22.7.5 Documentation

A manual with a detailed description of VODIA is available at the Astro-WISE web site.

## 22.8 HOW-TO use Galactic extinction in Astro-WISE

We have implemented in Astro-WISE two Galactic extinction maps:

SFD: Schlegel, D., Finkbeiner, D., & Davis, M., ApJ, 1998, 500, 525

Arenou: Arenou F., Grenon M., Gomez A., Astron. Astrophys. 1992, 258, 104

In the case of SFD map we used an original IDL program rewritten in python http://astro.berkeley.edu/ marc/dust/data/data.html

### 22.8.1 SFD extinction map: for extragalactic sources

To find the Galactic extinction towards an extragalactic object one can use the SFD map for which you have to provide galactic coordinates:

```
awe>longvec=45.0
awe>latvec=45.0
awe>from astro.util.extinction import extinction
awe>ret=extinction(longvec,latvec)
awe>print ret
[0.0439861752093]
```

The returned value is an excess ratio ($E_{B-V}$) in the selected direction. You can use as well vectors for input coordinates:

```
awe>longvec=[0.0,45.0,90.0]
awe>latvec=[0.0,45.0,90.0]
awe>from astro.util.extinction import extinction
awe>ret=extinction(longvec,latvec)
awe>print ret
[101.313850403, 0.0439861752093, 0.012209450826]
```

The Galactic extinction can be calculated with an interpolation between pixels closest to the desired direction:

```
awe>longvec=[0.0,45.0,90.0]
awe>latvec=[0.0,45.0,90.0]
awe>from astro.util.extinction import extinction
awe>ret=extinction(longvec,latvec,interp=True)
awe>print ret
[99.697704474, 0.0443909994508, 0.0119094799738]
```

Note: The number precision used in the IDL code is lower than in the python implementation. This can cause differences in derived $E_{B-V}$ between the two implementations in areas of highly varying extinction. Differences are $< 0.1\%$ for 99.8% of the sky as table 22.5 illustrates:

### 22.8.2 Arenou extinction map: inside the Galaxy

Arenou extinction model based on Hipparcos data and provides an extinction inside the Galaxy, i.e., for a selected distance. The user can provide a distance (in kpc), if the distance is omitted, an extinction for 15 kpc will be returned (according to the model).

Table 22.5: Differences in derived $E_{B-V}$ between the IDL and python implementation for SFD

| Absolute difference | sky area fraction |
|---|---|
| $< 0.1\%$ | 99.8 % |
| $< 1\%$ | 99.8 % |
| $> 5\%$ | 00.02 % |
| $> 10\%$ | 00.004 % |
| $> 50\%$ | 00.0004 % |

```
awe>longvec=[0.0,45.0,90.0]
awe>latvec=[0.0,45.0,90.0]
awe>from astro.util.extinction import extinction
awe>ret=extinction(longvec,latvec,source='Arenou')
awe>print ret
[0.51390040827009664, 0.017030418212218647, 0.032073867112540198]
```

or, for 100 pc distance,

```
awe>longvec=[0.0,45.0,90.0]
awe>latvec=[0.0,45.0,90.0]
awe>d=[0.1,0.1,0.1]
awe>from astro.util.extinction import extinction
awe>ret=extinction(longvec,latvec,source='Arenou',dist=d)
awe>print ret
[0.08085090032154342, 0.017030418212218647, 0.0032773954983922873]
```

## 22.9 Coordinate transformation

A number of functions for coordinate transformations are available in Astro-WISE (which are based on the IDL astro library).

1. **glactc** Convert between celestial and Galactic (or Supergalactic) coordinates.

   Input parameters:

   ra right ascension, hours (or degrees if degree=True is set), scalar

   dec declination, degrees, scalar

   year equinox of ra and dec, scalar

   degree if degree=True, both coordinates are in degree (overwise ra is in hours), degree=False by default

   fk4 if fk4=True, then coordinates are assumed to be in FK4, if fk4=False (default), FK5 is assumed. By B1950 coordinates use year=1950 and fk4=True

   SuperGalactic SuperGalactic=False by default, if SuperGalactic=True, SuperGalactic coordinates are returned (deVaucouleurs et al. 1976), to account for the local supercluster. The North pole in SuperGalactic coordinates has Galactic coordinates l = 47.47, b = 6.32, and the origin is at Galactic coordinates l = 137.37, b= 0 )

   eqtogal direction of conversion, eqtogal=True by default, if eqtogal=False, the input coordinates (ra, dec) are galactic coordinates and returned coordinates are celestial ones

Example: Convert coordinates (0.0,0.0) to galactic coordinates

```
awe>from astro.util.idllib import glactc
awe>glactc(0.0,0.0,2008.0)
(96.112413056666824, -60.188305254568284)
```

Convert galactic coordinates (0.0,0.0) to FK4 coordinates for epoch 2008.0, in degrees

```
awe> from astro.util.idllib import glactc
awe> glactc(0.0,0.0,2008.0,degree=True,fk4=True,eqtogal=False)
(266.53170097124888, -28.938911978654406)
```

2. **precess** Precess coordinates between two epochs

   Input parameters:

| | |
|---|---|
| ra | right ascension, degrees (or radians if radian=True is set), scalar |
| dec | declination, degrees (or radians if radian=True is set), scalar |
| equinox1 | original equinox of coordinates, numeric scalar |
| equinox2 | equinox of precessed coordinate, numeric scalar |
| fk4 | if fk4=True, then coordinates are assumed to be in FK4, if fk4=False (default), FK5 is assumed |
| radian | if radian=True, coordinates must be in radians, by default radians=False |

   Example:

```
awe> from astro.util.idllib import precess
awe> ra=329.887720833
awe> dec=-56.9925147222
awe> precess(ra, dec, 1950.0,1975.0,fk4=True)
(330.3144305415542, -56.871861264857067)
```

## 22.10 HOW-TO use SourceCollections in Astro-WISE

The SourceCollection classes extend the concepts of data lineage and data pulling to catalog, for example sample selection and parameter derivation.

This HOW-TO is not yet complete and should be considered a draft. For background information and details see the thesis of Hugo Buddelmeijer.

### 22.10.1 Overview

A SourceCollection is a ProcessTarget for source catalogs. A SourceCollection represents both a catalog of sources with attributes and the operation to create the data of this catalog. Sources are identified by their `SLID-SID` combination and attributes by their name. The operations range from selection of sources to calculation of attributes and are applied to other persistent objects, often other SourceCollections. Each operator corresponds to a separate persistent class that is derived from the base SourceCollection class.

### 22.10.2 An Astro-WISE Session

Using the SourceCollection classes is demonstrated by showing an example Astro-WISE session.

#### Bootstrapping

A session would usually start with retrieving an existing SourceCollection. This demo session starts with creating a new SourceCollection because this makes the pulling examples more predictable.

For now it is required to wrap a SourceList in a SourceCollectionWrapper class in order to use it with in the SourceCollections. Ultimately, the SourceCollection classes will be better integrated with the SourceList and other *List classes.

```
from astro.main.SourceList import SourceList
from astro.main.sourcecollection.SourceCollection \
  import SourceCollection
from astro.main.sourcecollection.SourceListWrapper \
  import SourceListWrapper

# Fetch a SourceList
sl = (SourceList.SLID == 1575051)[0]

# Create a SourceCollection from the SourceList
slw = SourceListWrapper()
slw.set_sourcelist(sl)

# The SourceCollection can be made persistent if wanted.
slw.commit()
print slw.SCID
# 100511
```

#### Pulling One Sample

SourceLists created from an image only contain photometric attributes. The SourceCollection classes allow attributes derived from those to be created by pulling them. This example shows how to pull a subset of the sources with newly derived attributes.

```
# Continue with 'slw' from above.

# Define a criterion to select the required sources.
query = ' "DEC" < 11 '

# Define the requested attributes.
# In this case a comoving distance and the inverse concentration index.
attributes = ['R', 'iC']

# Pull a SourceCollection that has the requested data.
#
# The information system will search for existing SourceCollections
# that can be used to fulfill the request. New SourceCollections
# are created if no suitable ones are found.
scnew = slw.derive(attributes=attributes, query=query)

# New SourceCollections are created to be as general as possible
# in order to facilitate reuse. In this case, the SourceCollection
# calculating 'R' will be created to calculate the attribute for all the
# sources of slw, not only for the sources with a declination below 11.

# Check whether the SourceCollection has been processed.
scnew.is_made()

# If False is returned, the SourceCollection has to be processed.
scnew.make()

# The is_made() and make() functions only apply to the parts of the
# dependency tree that is required to build the target SourceColloction.
# In this case, 'R' will only be calculated for the sources with DEC < 11.
```

### Loading Catalog Data

The catalog data of a SourceCollection can be accessed by loading it into a TableConverter object or by sending it over SAMP.

```
# Load the catalog data into a TableConverter.
scnew.load_data()

# Interface with the catalog data.
print scnew.data.attribute_order
# ['SLID', 'SID', 'R', 'iC']
print scnew.data.attributes['R']
# {'length': 1, 'ucd': '', 'null': '', 'name': 'R', 'format': 'float64'}
print scnew.data.data['R']
# [  562.54038472  1905.82573128   397.30116968 ...,    12.93537333

# Or send the catalog data over Samp.
from astro.services.samp.Samp import Samp
s = Samp()
s.broadcast(scnew)
```

```
# Highlight or select and broadcast some sources in Topcat or Aladin.
# Retrieve the SLIDs/SIDs or the row in the TableConverter.
s.highlightedSource(scnew)
# (1575051L, 812L)
s.selectedSources(scnew)
# [(1575051L, 843L), (1575051L, 847L), (1575051L, 848L)]
s.highlighted(scnew)
# 812
```

### Pulling More Samples

An important feature of the SourceCollections is that only the parts of a dependency tree are processed that are required to build the final target SourceCollection. This is demonstrated with the following example.

```
# Continue with 'slw' from before.

# Pull attributes for a subset of the sources.
scnew1 = slw.derive(query=' "DEC" < 11 ', attributes=['R'])
scnew1.is_made()
# Should return True because this part of the SourceCollection has already
# been processed in the previous example, if not, make it:
scnew1.make()

# Pull attributes for a smaller subset of the sources
scnew2 = slw.derive(query=' "DEC" < 10 ', attributes=['R'])
scnew2.is_made()
# True, because this has already been processed.

# Pull data for a larger subset of the sources
scnew3 = slw.derive(query=' "DEC" < 12 ', attributes=['R'])
scnew3.is_made()
# False, because this part of the SourceCollection
# has not been processed completely yet.
```

### Storing Catalog Data

New catalog data is created in the examples above. Storing the attribute values prevents them to be calculated again.

```
# Continue with the datasets from above.

# Commit the SourceCollection that need to be saved. This will recursively
# commit the dependency tree as well.
slnew2.commit()

# Store the source data that has been calculated. This will store the catalog data
# in the most optimal way. Although slnew2 only represents a subset of the
# sources, all the calculated attributes are stored: The 'R' value for the sources
# in scnew1 will be stored as well.
slnew2.store_data()
```

```
# The demo session ends here.
```

### 22.10.3 Pushing SourceCollections

The `derive()` function above creates a hierarchy of SourceCollections. This section shows how this hierarchy can be created manually.

It is preferable to use the data pulling functions and let the information system determine what needs to be created automatically. This is less work and facilitates reuse.

Nonetheless, not all SourceCollections can be created through pulling data, therefore it is useful to know how to create them manually.

#### Import

First import all relevant classes.

```
# astro.main classes
from astro.main.SourceList import SourceList
# Virtual base SourceCollection
from astro.main.sourcecollection.SourceCollection \
  import SourceCollection
# All derived operator SourceCollections
from astro.main.sourcecollection.SourceListWrapper \
  import SourceListWrapper
from astro.main.sourcecollection.FilterSources \
  import FilterSources
from astro.main.sourcecollection.SelectSources \
  import SelectSources
from astro.main.sourcecollection.SelectAttributes \
  import SelectAttributes
from astro.main.sourcecollection.ConcatenateAttributes \
  import ConcatenateAttributes
from astro.main.sourcecollection.AttributeCalculator \
  import AttributeCalculator, AttributeCalculatorDefinition
```

#### Bootstrap

Start with a new SourceCollection.

```
# Fetch a SourceList
sl = (SourceList.SLID == 1575051)[0]

# Create a SourceCollection from the SourceList
slw = SourceListWrapper()
slw.set_sourcelist(sl)
```

#### Calculate Attributes

An AttributeCalculator SourceCollection is used for the derivation of new source attributes from existing attributes. The calculation that is performed by an AttributeCalculator SourceCollection is given by an AttributeCalculatorDefinition object. The creation of AttributeCalculator SourceCollections in a pushing way requires some handwork:

```
# Find all AttributeCalculatorDefinition objects that can be used to
# calculate comoving distances.
acdsR = AttributeCalculatorDefinition.get_acds_by_attribute('R')

# Pick the first one.
acdR = acdsR[0]
acdR.name
'Comoving Distance Calculator'

# See which attributes are required by the ACD.
acdR.input_attribute_names
['redshift', 'RA', 'DEC', 'HTM']

# Which are al available in 'slw':
[a in slw.get_attribute_names() for a in acdR.input_attribute_names]
[True, True, True, True]

# The required attributes have to be selected with a SelectAttributes:
sa1 = SelectAttributes()
sa1.parent_collection = slw
sa1.selected_attributes = ['redshift', 'RA', 'DEC', 'HTM']

# And the AttributeCalculator can be initialized. The 'AC' property of an ACD
# object is class derived from the AttributeCalculator class which used this
# definition.
ac1 = acdR.AC()
ac1.parent_collection = sa1
```

Similarly for the inverse concentration:

```
acdsiC = AttributeCalculatorDefinition.get_acds_by_attribute('iC')
acdiC = acdsiC[0]
all([a in slw.get_attribute_names() for a in acdiC.input_attribute_names])

sa2 = SelectAttributes()
sa2.parent_collection = slw
sa2.selected_attributes = [a for a in acdiC.input_attribute_names]

ac2 = acdiC.AC()
ac2.parent_collection = sa2
```

### Selecting Sources

Selecting sources is either done with a FilterSources SourceCollection or with a SelectSources SourceCollection.

```
# A FilterSources represents a subset of the parent SourceCollection
# by evaluation of a selection criterion.
fs = FilterSources()
fs.parent_collection = slw
fs.set_query(' "DEC" < 11 ')
```

```
# Alternatively, a SelectSources SourceCollection can be used to select
# a subset that is explicitly listed by another SourceCollection.
# First create a SourceCollection with only source identifiers.
sa3 = SelectAttributes()
sa3.parent_collection = fs
# Use this to specify the selected sources of a SelectSources SourceCollection
ss = SelectSources()
ss.parent_collection = slw
ss.selected_sources = fs
```

### Combining All Attributes

A ConcatenateAttributes SourceCollection is used to combine the attributes fro the AttributeCalculators for the sources in the FilterSources.

```
# First select no attributes from the FilterSources
sa3 = SelectAttributes()
sa3.parent_collection = fs
# Select the comoving distance
sa4 = SelectAttributes()
sa4.parent_collection = ac1
sa4.selected_attributes = ['R']
# Select the inverse concentration
sa5 = SelectAttributes()
sa5.parent_collection = ac2
sa5.selected_attributes = ['iC']

# Combine all the attributes
ca = ConcatenateAttributes()
ca.parent_collections = [sa3, sa4, sa5]

# And we're done
scnew = ca
scnew.make()
scnew.load_data()
```

### Other Operators

SourceCollection classes not yet shown in this HOW-TO are

- ConcatenateSources, to combine the source of different SourceCollections.
- RenameAttributes, to rename the attributes of the parent SourceCollection.
- RelabelSources, to give sources of the parent SourceCollection a new `SLID-SID` combination by specifying an AssociateList.

```
from astro.main.AssociateList import AssociateList
from astro.main.sourcecollection.SourceCollection \
  import SourceCollection
from astro.main.sourcecollection.RelabelSources \
  import RelabelSources
from astro.main.sourcecollection.ConcatenateSources \
```

```
  import ConcatenateSources
from astro.main.sourcecollection.RenameAttributes \
  import RenameAttributes

# Fetch a SourceCollection and AssociateList
sc = (SourceCollection.SCID == 100161)[0]
al = (AssociateList.ALID == 472781)[0]

# Create a RelabelSources
rs = RelabelSources()
rs.parent_collection = sc
rs.associatelist = al

# 'rs' and 'sc' now represent 'different' sources, which
# can be concatenated. (This is not really useful though.)
cs = ConcatenateSources()
cs.parent_collections = [sc, rs]

# And the attributes can be renamed.
ra = RenameAttributes()
ra.parent_collection = cs
ra.attributes_old = ['MAG_ISO', 'MAGERR_ISO']
ra.attributes_new = ['MAG_B', 'MAGERR_B']
```

### 22.10.4 The SourceCollectionTree in the Background

The automatic creation of new SourceCollections is managed by non-persistent SourceCollectionTree objects. For example, the `derive()` function, but also the `is_made`, `make` and `load_data` functions use a SourceCollectionTree. It is instructive to discuss the SourceCollectionTree class, even though it is not often required to interface with one directly.

#### Derive

The `derive()` function above uses a SourceCollectionTree to pull SourceCollections. This example shows what happens inside this function.

```
from astro.main.sourcecollection.SourceCollection \
  import SourceCollection
from astro.main.sourcecollection.SourceCollectionTree \
  import SourceCollectionTree

# Same start as above.
slw = (SourceCollection.SCID == 100511)[0]
query = ' "DEC" < 11 '
attributes = ['R', 'iC']

# Create a SourceCollectionTree from the SourceCollection. The SCT traverses
# the dependency tree and keeps the progenitors of the given SC in memory.
# A Pass SC is created with the given SC as parent, which is used as the end
# node of the tree. Later functions of the SCT will replace SCs in the tree,
# but never this end node.
```

```
sct = SourceCollectionTree(slw)

# Apply the selection criterion. There are two things that can happen:
# 1) The SCT discovers an existing SC that represents the requested sources
#    and will create a SelectSources SC to select the requested sources.
# 2) The SCT cannot find suitable SCs and creates a new FilterSources SC with
#    the end node as parent and the given selection criterion as query. The
#    query is not evaluated; the exact composition of sources is unknown.
# In both cases, the created SC is placed between the Pass node at the end of
# the tree and its parent.
sct.apply_filter(query=query)

# Select the attributes. For every attribute the SCT will search for existing
# SCs that represent the attribute for the requested set of sources. A
# hierarchy of SelectAttributes and ConcatenateAttributes SCs is created that
# provides the right attributes. The SCT will try to instantiate new
# AttributeCalculator SCs if there are no existing SCs that contain a
# requested attribute.
sct.apply_attribute_selection(attributes)

# The end node of the tree now represents the requested catalog. This is
# a Pass SC, so its parent is returned by the .derive() function.
scnew = sct.sourcecollection.parent_collection
```

### Visualizing a SourceCollectionTree

The dependency tree of a SourceCollection can be visualized with a SourceCollectionTree (figure 22.1).

```
sct = SourceCollectionTree(scnew)
sct.make_dot_graph('howtotree1')
```

### Making Catalog Data

The functions of the SourceCollection class that handle the catalog data, `is_made`, `make` and `load_data`, can be called either with optimization or without. With optimization (`optimize=True` parameter, the default) a SourceCollectionTree is created and the respective function of this object is called to perform the required action in the optimal way.

```
# Continuing with 'scnew' from earlier

# Calling is_made without optimization will return True because scnew is a
# ConcatenateAttributes SC, which does not have to be processed.
scnew.is_made(optimize=False)

# Calling is_made with optimization is equal to the following:
# Create a SourceCollectionTree
sct = SourceCollectionTree(scnew)
# Optimize the tree for loading catalog data, by placing the selection of
# sources and attribute early in the tree.
sct.optimize_for_load()
```

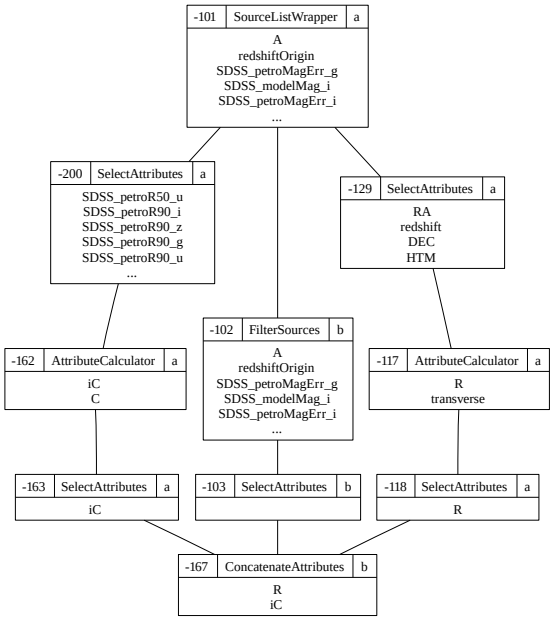



Figure 22.1: Dependency tree generated by the SourceCollectionTree.

```
# Now the entire optimized tree can be checked.
sct.is_made()
# This will recursively call 'is_made(optimize=False)' on the SCs in the
# optimized tree.

# Similarly, calling 'scnew.make()' is identical to:
sct = SourceCollectionTree(scnew)
sct.optimize_for_load()
sct.make()
# This will recursively call 'make(optimize=False)' on the SCs in the
# optimized tree that are not yet made.

# Finally, 'scnew.load_data()' is identical to:
sct = SourceCollectionTree(scnew)
sct.optimize_for_load()
sct.load_data()
# This will load the catalog data of the end node of the tree in the
# most optimal way. It will not necessarily load the catalog data
# for all the other SCs in the tree.
```

### Store Catalog Data

Storing catalog data with `store_data()` in an optimized way works differently from the functions above. In practice, all the attributes of the AttributeCalculators in the dependency tree of the SourceCollection will be stored. No catalog data will be stored as part of the SourceCollection itself, unless it is an AttributeCalculator too.

```
# Store the catalog data in an optimized way.
```

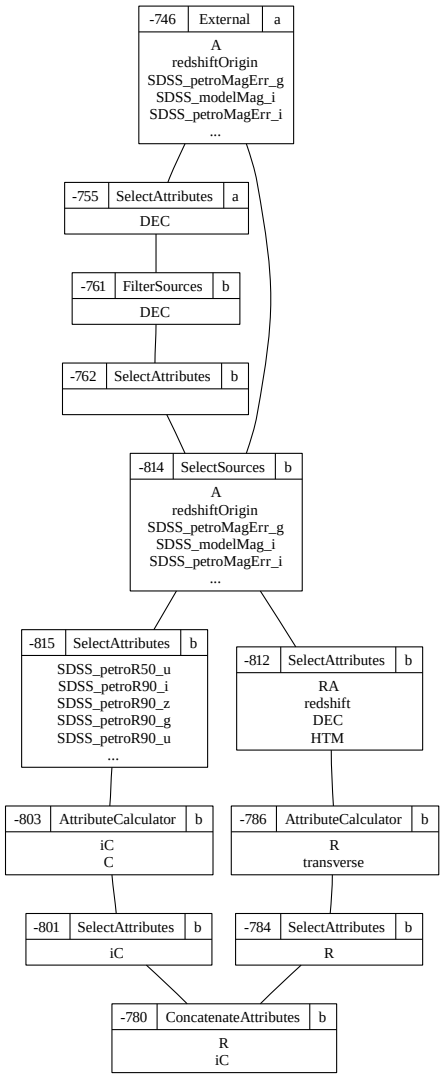



Figure 22.2: Optimized dependency tree of figure 22.1, generated by the SourceCollectionTree. The exact shape of the tree will differ depending on what part of the SourceCollections has already been processed.

```
slnew.store_data()

# This is equivalent to
sct = SourceCollectionTree(slnew)
sct.store_data()
```

### Finding SourceCollections

There are several moments when the SourceCollectionTree will search for existing SourceCollections in order to fulfill a data pulling request. The SourceCollectionTree will create a list of all SourceCollections that could be used for a particular purpose. This list is subsequently

ranked according to a key function and the SourceCollection with the highest rank is selected. Any SourceCollection with a positive key value would be suitable. The `key_functions` class property of the SourceCollectionTree is a dictionary that holds these key functions. The keys of the dictionary are:

- `find_selection`: Used to rank SourceCollections that represent the sources selected by a given selection criterion.
- `find_attribute`: Used to rank SourceCollections that provide a given attribute for a specific set of sources.
- `find_attribute_new_calculators`: Used to rank new AttributeCalculator SourceCollections that provide a given attribute.
- `find_sources`: Used to rank SourceCollections that represent the source identifiers of a given SourceCollection.

The selection process can be influenced by overloading these functions:

```
# Start with a SourceCollection.
slw = (SourceCollection.SCID == 100511)[0]

# Create an AttributeCalculator to calculate 'R' without lamdba.
acd = AttributeCalculatorDefinition.get_acds_by_attribute('R')[0]
ac2 = acd.AC()
ac2.parent_collection = slw
ac2.set_process_parameter('omega_m', 1.0)
ac2.set_process_parameter('omega_l', 0.0)

# Create an SCT and manually ensure that the relevant SCs are tracked.
sct = SourceCollectionTree(slw)
sct.track_children_auto(cache=True)
sct.track_tree(ac2)

# Search for the comoving distance. An AC with the wrong omega_m is selected.
sc1=sct.find_attribute('R')
print "#sc1 omega_m", sc1.get_process_parameter('omega_m')
#sc1 omega_m 0.3

# Define a new key function to find the correct AC.
from astro.main.sourcecollection.SourceCollectionTree \
  import key_find_attribute
def mykey(scd):
    # First retrieve the default ranking.
    tkey = key_find_attribute(scd)
    # The 'scd' is a dictionary, the key 'sc' points to the actual SC.
    sc = scd['sc']
    # Reduce the key value of SCs that are not an AttributeCalculator
    if not isinstance(sc, AttributeCalculator):
        tkey -= 10**9
    # and reduce the key value of the ACs that use another omega_m.
    elif sc.get_process_parameter('omega_m') != 1.0:
        tkey -= 10**9
```

```
    return tkey

# Set the new key function.
SourceCollectionTree.key_functions['find_attribute'] = mykey

# Search for the comoving distance again, the preferred AC is found.
sc2=sct.find_attribute('R')
print "#sc2 omega_m", sc2.get_process_parameter('omega_m')
#sc2 omega_m 1.0
```

### 22.10.5 AttributeCalculatorDefinitions

The calculation that is performed by an AttributeCalculator SourceCollection is described by an AttributeCalculatorDefinition object. These objects can be created by any scientist and shared with others.

The code of an AttributeCalculatorDefinition is stored in a file on the dataserver. This (python) file contains a new AttributeCalculator class that is derived from the one in code base. The `create_from_file()` method of the AttributeCalculatorDefinition class can be used to create a new definition from this class. Auxiliary files can be used by wrapping everything in a tarball.

The procedure for this is too long to to list in this document, see demo 17 for an example.

```
cd $AWEPIPE/astro/experimental/SourceCollection/demos/demo17
```

### 22.10.6 SAMP Interaction and Query Driven Visualization

A design goal of the SourceCollection classes was to be able to use them interactively over SAMP. New SAMP messages are designed to to allow query driven visualization in a more declarative way than is possible with other information systems.

The SAMP interaction is described in the HOW-TOon SAMP (section 23.8) The Query Driven Visualization is described in the HOW-TOon Query Driven Visualization (section 23.9).

# Chapter 23

# Visualization

## 23.1 HOW-TO Inspect

This HOW-TO deals primarily with *image* inspection methods. Other inspection routines are described in their relevent sections (see, e.g., sections 18.4 and 19.4.2). These and other inspection routines may eventually be linked from this HOW-TO formally, but they will never be described in any detail here.

### 23.1.1 Image Inspect Plot

Image inspection takes place primarily a MatPlotLib (PyLab) window and is illustrated in figure 23.1. The plot contains a representation of the image with a title containing the filename of the frame being inspected. There are also pixel coordinate indicators for convenience. As this plot is within a Pylab window, all the familiar manupulation routines are available (e.g., panning, zooming, etc.) In addition to these, there are some single-key commands to create new plots that illustrate specific details around the cursor position. The next section gives details on these.

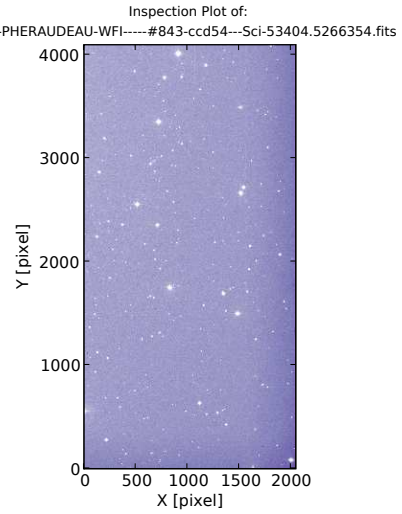



Figure 23.1: A typical image inspection window without the PyLab controls.

### 23.1.2 Image Inspect Method

Image inspection can be called from *any* `BaseFrame`-derived frame, and is called by executing the frame's `inspect()` method:

```
awe> frame = BaseFrame(pathname='filename.fits')
awe> frame.inspect()
```

There are also a large number of options to control how the plot looks and even what is plotted. A summary of the options is given below:

pixels: optional list or array representing the image to be inspected (can be MxN for greyscale, or MxNx3 for RGB)

zone: tuple of (x0, y0, x1, y1) representing the image coordinates of the two oposing corners of the sub image to consider

kappa: the factor by which the dynamic range is increased in units of sigma (0 gives full range)

iterations: number of iterations in the kappa-sigma range clipping

cmap: PyLab color map instance

vmin: lower display range in native units (e.g. ADU)

vmax: upper display range in native units (e.g. ADU)

interpolation: type of interpolation the PyLab viewer uses (nearest, bilinear, etc.)

width: width of the PyLab figure window (in inches)

ratio: ratio by which to scale the figure height (default: x_dim/y_dim)

viewer: external viewer to use in case the image is too large

force_viewer: always use the viewer

subplot_size: width and height in pixels of region of interest

contour_levels: number of contour levels for the contour plot of the region of interest

num_bins: number of bins in the histogram plot

extension: extension of the filetype to save plot to (png, ps, or eps) None disables saving

compare: compare this frame to its previous version using difference imaging (current-previous), pixels is ignored

level: depth of query for previous version (0 goes as deep as possible) when compare is True

other: a second of the same type of Frame object to replace previous when compare is True (if color is True, other can be a list of two images)

clip: kappa-sigma clip each image prior to subtraction when compare is True

color: use color combining (RGB) instead of differencing when compare is True (kappa, vmin/vmax only honored when clip is True), this image is R, other is B if single, other is [G, B] if it is a list (EXPERIMENTAL)

Also, there are a number of commands that work on an area around the current cursor position. A summary of these commands are given below:

**q** closes the most recent plot window when pressed in the main window

**SPACE** displays the X and Y coordinate (FITS standard unit indexed) and the count level

**a** performs aperture photometry on brightest feature in the region of interest (NOT YET IMPLEMENTED)

**c** displays a contour plot of the region of interest (see contour_levels)

**h** displays a histogram of the pixel values of the region of interest (see num_bins)

**r** displays a radial plot of the brightest feature in the region of interest

**w** displays a wireframe plot of the region of interest

**p** displays profile plots in both X and Y dimensions versus intensity (count level)

Please see the `inspect` docstring for more details and current details on both of these:

```
awe> help(frame.inspect)
```

### 23.1.3 Image Display Method

A *display* method is available for all frames:

```
awe> frame.display()
```

This will open up the frame in *skycat*, by default.

## 23.2 HOW-TO View the Contents of a Photometric Association Catalog

The first processing step in the photometric pipeline always consists of extracting source catalogs from the standard field data, and associating the results with a standard star catalog. These particular 'association' catalogs are represented in the system by `PhotSrcCatalog` objects. To view the content of a `PhotSrcCatalog` object, simply invoke its *inspect* method:

```
awe> photcat.inspect()
```

which will result in an output to screen that looks like the one shown in Figure 23.2. The *inspect* plot shows the magnitudes of the individual standard stars as known to the standard star catalog on the x-axis, and their associated raw zeropoints on the y-axis.

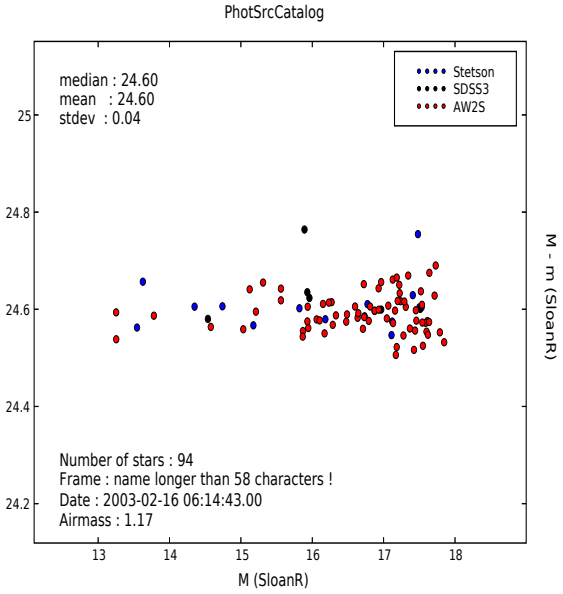



Figure 23.2: The result of invoking the *inspect* method of a `PhotSrcCatalog` object.

## 23.3 HOW-TO View Processing Results as a mosaic (MEF file)

It is sometimes useful to look at the results of a processing step not chip-by-chip, but at all chips simultanesouly. One such processing result could be the overall illumination correction in FITS-format (see Fig. 23.3).

To facilitate this, a little tool has been created that can merge the separate results for every chip back into a Multi-Extentsion FITS (MEF) file[1]. This tool can be found in CVS under `$AWEPIPE/astro/util/Image.py`. Given below is an example of how to merge the BiasFrames for a given day back into an MEF file using this tool:

```
Example 1.
    awe> from astro.main.BiasFrame import BiasFrame
    awe> from astro.util.Image import Image
    awe> biases = []
    awe> for i in range(8):
    ...   bias = BiasFrame.select(instrument='WFI', date='2000-04-28', chip='ccd5%s' % i)
    ...   bias.retrieve()
    ...   biases.append(bias)
    awe> len(biases)
    8
    awe> MEF = Image('output_MEF.fits')
    awe> MEF.frames = biases
    awe> MEF.make()
```

The frames used MUST first exist on disk in your local directory (cf. retrieve() method) and be a consistent set for the instrument (e.g., one frame for each of the 8 chips of the WFI instrument). The output MEF with the name "output_MEF.fits" can then be viewed in eg. skycat. Note that the possibility of viewing a complete MEF file depends on the viewer you use. Be aware, that for instruments with larges numbers of chips, the memory requirements are correspondingly larger. For OmegaCAM, a 32 CCD instrument, the output file will be greater than 1GB on disk.

In the next example, extended syntax is shown in addition to optimal command-lines for different viewers. In this case, if frames are used, they will be retrieved by default unless the retrieve option is explicitly set to False.

```
Example 2.
    awe> # normally imported automatically
    awe> from astro.util.Image import Image
    awe>
    awe> # using instantiated frames
    awe> img = Image('output.fits', frames=list_of_frames)
    awe> # img.frames can also be set directly
    awe>
    awe> # using filenames
    awe> img = Image('output.fits', filenames=list_of_filenames)
    awe> # img.filenames can also be set directly
    awe>
    awe> img.make()

    awe> # view with skycat allowing greater zoom range
```

[1]The RawFrames were made from a MEF associated with their RawFitsData object.

```
awe> os.system('skycat -min_scale -20 output.fits')
awe> # you must select "Display as one Image"
awe>
awe> # view with ds9 in mosaic image mode
awe> os.system('ds9 -mosaicimage wcs output.fits')
awe> # you can select Zoom -> Zoom to Fit Frame and
awe> # Scale -> Scope -> Global to scale like skycat
```

All the descriptions above assume a non-RawFrame type is used. If a RawFrame type is used (i.e., with trim regions still intact), the frame_type option MUST be set to 'raw' for a proper chip-to-chip alignment. Using RawFrame types is generally not done as the RawFitsData is already available as a multi-extension FITS image.

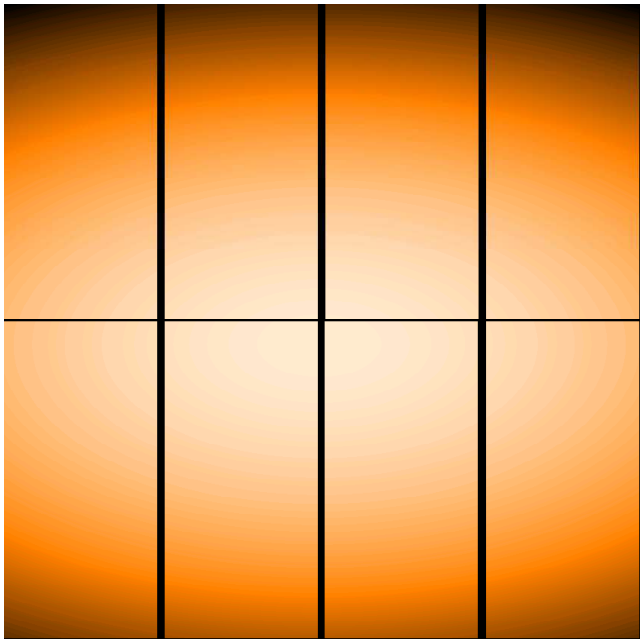


Figure 23.3: The eight illumination correction frames derived for the R filter (#844) of the Wide-Field Imager combined into one multi-extension FITS file.

## 23.4 HOW-TO: Image Services

This HOW-TO shows how to view FITS images in multiple formats from within the DBView web-service and from the `awe`-prompt using catalogs (`SourceLists`) for the source specification.

### 23.4.1 Visualizing and Navigating the Database with DBviewer

To visualize the data in table-views and tree-views, the DBviewer in this Astro-WISE portal offers HTML controls to enter data, send forms and instructions to the the server interacting with the database. Narrow the search by entering conditions to SQL-queries, set context to focus the search on projects by filling in input-fields of forms. Bookmark results, and query-forms as well for repeated searches. Use the Oracle-SQL as shown as basis to modify or build into new SQL-queries. Navigate by links between tableviews, treeviews of a single object and its dependencies,and back from the tree to a row in the table to search for similar objects. Refer to DBviewer's own help-functions for more information.

### 23.4.2 Visualizing FITS Images

#### Directly From Dbview

In DBview, controls are present to view whole FITS images via the web-browser after being processed on-demand into png format by the Astro-WISE image-server. Large images are binned down to a manageable size. Simple controls allow inverting colors and basic adjustment of contrast and size. For SourceList data, overplotting of positions on these images is being developed.

#### Exporting FITS Images to VO Applets and Image Browsers

Next to providing downloaded FITS data to resident programs, the DB viewer supports features to interact with VO-table-aware astronomical programs imported as web-applets. This can be a useful feature for clients without suitable software installed or as a way to be sure to have the latest version available.

The Aladin Sky Atlas java-applet will read downloaded FITS image data or VO-XML source data. For some science-tables, there is an option to load FITS-data directly from links without exporting to local disk. The applet can access a large number external catalogs and do a number of operations, including interactive plotting.

The graphical table viewer-applet TOPCAT reads downloaded tabular data from the DB-viewer in VO-XML format. TOPCAT has statistical and interactive (regression, 3-D) plotting abilities, and can access a few catalogs. The program needs a one-time configuration step of the java webstart manager on the client's machine.

> NOTE: Refer to DBviewer's own help-functions for more information on downloading and interaction with external programs, see VO work-bench for additional web-client software.

FITS images from any table of the database can be loaded via the clients machine into astronomical Image Browsers. In the Dbviewer, activate the HTML-links to FITS files to download via the web browser dialog menu. **NB: FITS-images may be large (> 500 Mb), and in a compressed format.**

Alternatively, FITS file downloads may be done from the `awe`-prompt rather than from the web-interface, using the object-method retrieve(), for example:

```
awe> bias=BiasFrame.filename==’2000-04-21cal541_ccd57.fits’
awe> bias.retrieve()
```

The FITS file is saved in the current working directory of the `AWE` session.

**FITS Viewing Programs**

- FV software tool for viewing and editing any FITS format image or table
- SAOImage DS9 astronomical image and data visualization application
- SkyCat astronomical image and catalog browser

### 23.4.3 Visualizing FITS Cut-out Images

Working from the `awe`-prompt, SourceList and AssociateList objects can export FITS filenames and positional data. These are sent to the image-server's cut-out service via an image-client script to provide either sub-image FITS files, or an HTML page and table with links to PNG representations of the sub-images. In addition, the HTML page enables access to the original FITS images and the sub-image FITS files, and downloading in FITS format. Magnified PNG representations can be viewed in private windows to be moved around the screen and juxtaposed for comparison. For full control over the image-client, the user can add to the data extra parameters to specify size of sub-images (per item or per series), designate pixel coordinates, add FITS headers, or determine the name of the output file. For more info on how to use the image-client see the code examples below.

### 23.4.4 `awe`-prompt: Python Code Examples to Access the Cut-out Services of Image-server

Importing the interface to the image-server:

```
awe> from astro.services.imageview import imgclient
```

**Dedicated SourceList or AssociateList object methods**

Object-Methods sourcelist.sources.make_image_dict(sids, mode=’sky’) or al.associates.make_image_dict(aids, mode=’sky’) return dictionaries to get cut-out FITS images from the image server.

- **sids/aids** may contain one or a list of SID/AID indexes.
- **mode** can specify sky (default) or grid coordinates.

Example to obtain cutouts of SourceList SID number 1..10 for SLID #54052:

```
awe> from astro.main.SourceList import SourceList
     # instantiate a sourcelist object by any query
awe> sourcelist = (SourceList.SLID == 54052)[0]
     # make the dictionary for a given list of SIDs
awe> imgdict=sourcelist.sources.make_image_dict([1,2,3,4,5,6,7,8,9,10],
...  mode=’grid’)
```

Send the dictionary to the image-server with an optional instruction for size:

```
     # have dictionary checked
awe> ic = imgclient.imgclient(imgdict, wide_high=[150, 150])
     # receive zipped cut-out fitsfiles or:
awe> ic.getzipfile()
     # receive an HTML-page with image-links to png representations
awe> ic.getimg()
     # obtain the name of the zipfile that has been retrieved with .getzipfile()
awe> print ic.zipfilename
```

Example to obtain cutouts of Associatelist ALID #7854:

```
awe> from astro.main.AssociateList import AssociateList
     # select or make an AssociateList object by any query
awe> AL = (AssociateList.ALID == 7854)[0]
     # check the number of associated Sourcelists
awe> print len(AL.sourcelists)
2
     # make the dictionary for a given list of AIDs
awe> imgdict = AL.associates.make_image_dict([1,2],mode='grid')
awe> print imgdict
{'Sci-EVALENTYN-WFI-------#844---Coadd---Sci-53874.5477624.fits':
 [{'PIX_Y': 61.8842964172363,   'PIX_X': 3229.7326660156305},
  {'PIX_Y': 50.037860870361293, 'PIX_X': 9802.1259765625}],
 'Sci-EVALENTYN-WFI-------#844---Coadd---Sci-53876.4051544.fits':
 [{'PIX_Y': 94.668701171875, 'PIX_X': 3188.1870117187505},
  {'PIX_Y': 83.101295471191392, 'PIX_X': 9760.25}]}
```

Send to image-server as shown above for SourceLists. When requested as HTML image-table, cutouts from images will be arranged column-wise. Current limits of of this representation is 16 x 60 columns.

**Generating dictionaries with data and control parameters for cut-out services**

To construct a dictionary for the image-server without the help of a function, the following calls and data-structures are available to program data input and output. Invoke the image-client and -services in the usual way:

```
awe> from astro.services.imageview import imgclient
     # check dictionary and convert into url
awe> x=imgclient.imgclient(dictionary,wide_high=[100,100])
     # retrieve zipfile of FITS-image cut-out files or
awe> x.getzipfile()
     # retrieve HTML-pages showing cut-outs and original images
awe> x.getimg()
     # obtain the name of the zipfile that has been retrieved with .getzipfile()
awe> print x.zipfilename
```

The user-dictionary should contain filename-data items:

```
{filename1: [pdata], filename2: [pdata], etc}
```

The [pdata] list may contain simple lists with [RA,DEC] (and optionally width, height data) or dictionaries with additional parameters. The wide_high parameter is optional to set generic width and height for all sub-images. Special dictionary-keys are:

- **PIX_X, PIX_Y** instead of RA,DEC to enter pixel-coordinates
- **width, height** individual sub-FITS image dimensions in pixels (see wide_high parameter above)
- **FTAG** for a user-defined tag added to the output-filenames
- Additional items to be placed into the FITS-headers.

**Examples of [pdata]**

Lists with optional height and width parameters per individual sub-image:

```
[[11.5557568095, -29.3887976732],
 [11.4245138174, -29.3883174776, 200, 200],
 [11.5253142382, -29.3874586034, 300, 300]]
```

Dictionaries specifying position, size and additional items:

```
[{'RA': 11.5557568095, 'DEC': -29.3887976732, 'extra1': 123, 'extra2': 321},
 {'RA': 11.4245138174, 'DEC': -29.3883174776, 'width': 200, 'height': 200},
 {'RA': 11.5253142382, 'DEC': -29.3874586034, 'width': 300, 'height': 300,
  'extra1': 123, 'extra2': 321, 'extra3': 567, 'extra4': 8910}]
```

NOTE: Numeric values in [pdata] can also be in string format (e.g. 200 == '200' == "200").

## 23.5 HOW-TO Obtain PSF Information of Science Images

A simple recipe is available to make several plots of SExtractor parameters relevant to the PSF. The recipe is $AWEPIPE/astro/filerecipes/PSF_Anistropy.py and is used as follows:

```
------------------------------------------------------------------------------

    CATEGORY : Health check

    PURPOSE : Detect PSF Anisotropy

    FULFILLS : Requirement 554 (Category III)

------------------------------------------------------------------------------

    This recipe makes 5 plots of SExtractor parameters:
    1) Bars of length FWHM_IMAGE (times a scale factor) and angle w.r.t.
       x-axis Theta. Average FWHM_IMAGE and THETA are evaluated per area
       in a grid of Xpos/Ypos.
    2) MAG_ISO vs. FLUX_RADIUS (half-light radius)
    3) FWHM_IMAGE vs. THETA (position angle)
    4) FWHM_IMAGE vs. Xpos-CRPIX1
    5) FWHM_IMAGE vs. Ypos-CRPIX2

------------------------------------------------------------------------------

    Mandatory inputs :

        -l : list of input split raw or processed science images

    Configuration parameters (min - [default] - max) :

        -s  : switch to not run SExtractor (if catalogs already exist)
        -d  : SExtractor detection threshold
              range: 0.0 - [10.0] - 100.0
        -f  : maximum FLAG of sources
              range: 0 - [0] - 255
        -m  : maximum FLUX_MAX of sources
              range: 0 - [60000] - 70000
        -r  : minimum FLUX_RADIUS of sources
              range: 0.0 - [0.0] - 50.0
        -rm : maximum FLUX_RADIUS of sources
              range: 0.0 - [50.0] - 50.0

    Switches :

        -in : interactive mode (if omitted a postscript file is created)

    Example of use :

        awe PSF_Anisotropy.py -l image12345_?.fits [-s] [-d 5.0] [-f 1]
```

```
                                [-m 50000] [-r 2.0] [-rm 5.0] [-in]

-------------------------------------------------------------------------
```

Note that the recipe can be used on a single coadded image by giving it as input.
Below is an example of an output plot

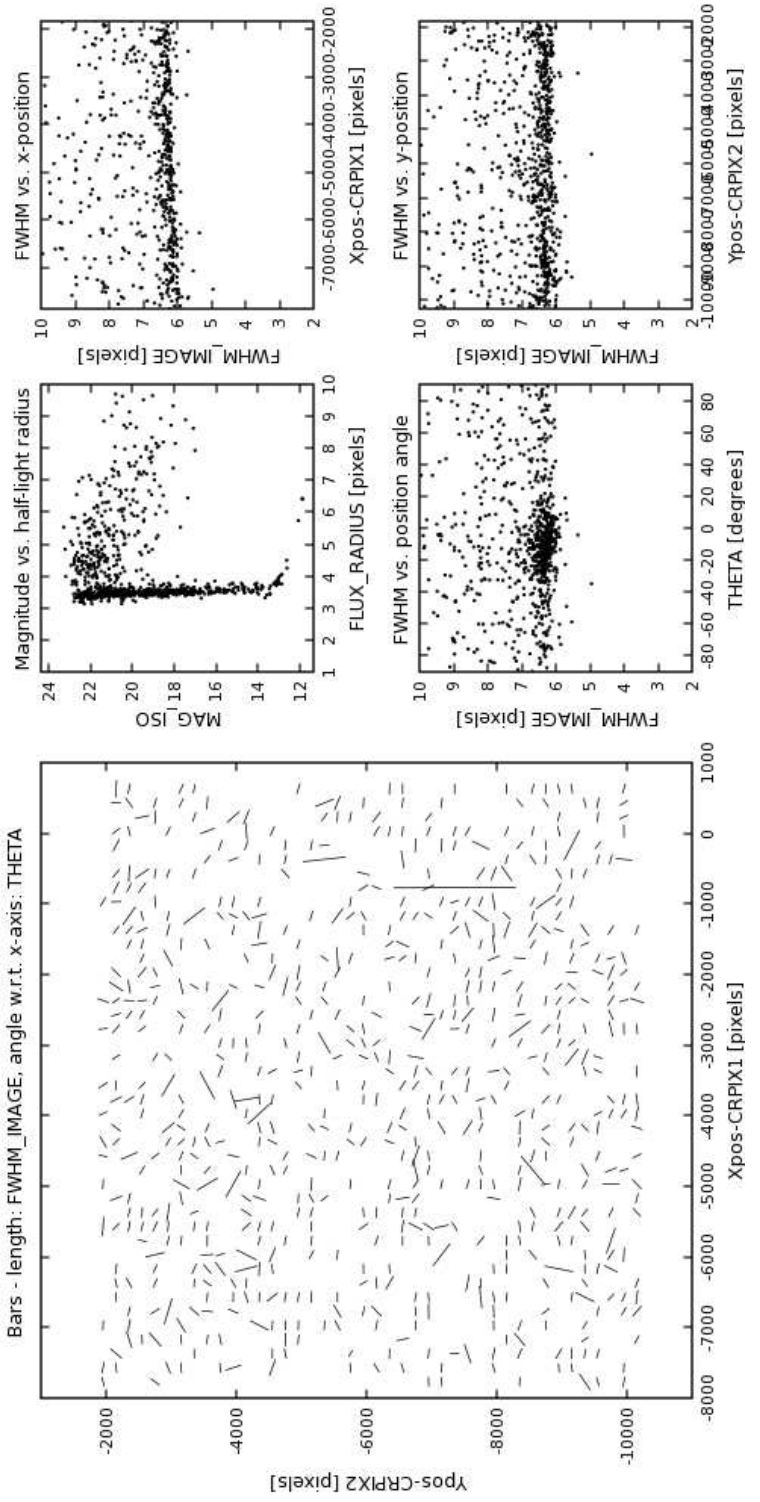



Figure 23.4: Example output plot of PSF_Anistropy

## 23.6 HOW-TO Visualize observational pointings with ObsViewer

`ObsViewer` can query the database for RawScienceFrames or ReducedScienceFrames for a given rectangular region of the sky, wavelength range and range in exposure time. It can output the results in a figure and ascii files.

NOTE1: EXPERIMENTAL VERSION. Bug reports and other feedback welcome:
Gijs Verdoes Kleijn (verdoes@astro.rug.nl)

NOTE2: ONLY WFI OBSERVATIONAL POINTINGS ARE QUERIED.

EXAMPLE1 of use at the `awe`-prompt:

1. `awe> from astro.experimental.ObsViewer import ObsViewer`
2. `awe> o=ObsViewer()`
3. `awe> o.query_database(rastart=174, raend=178, decstart=-5, decend=-5,`
   `... central_lambdastart=4000.0, central_lambdaend=5000.0,`
   `... expomin=0.0, expomax=200.0,raw='yes')`
4. `awe> o.preview(newplot=1,plotcolor='b',plotsize=20)`
5. `awe> o.plot(plotfov=1,fillfov=1,newplot=1,plotcolor='b',plotsize=20,sec_per_dot=1.0)`

The command lines have the following effects:

1. IMPORTS MODULE INTO AWE ENVIRONMENT.
2. MAKES A CLASS INSTANTIATION.
3. QUERIES THE DATABASE.
   Input parameters:
   - rastart/end: (degrees) start and end right ascension of rectangular region in the sky.
   - decstart/end: (degrees) start and end declination of rectangular region in the sky.
   - central_lambdastart/end: (Angstrom) start and end of wavelength region in which the central wavelength of the exposure filter has to be.
   - expomin/max: (sec) minimum/maximum exposure time for exposure.
   - raw: query for raw images (raw='yes') or reduced images (raw='no').

   Result:
   a query which can be used by o.preview and o.plot.
4. PLOTS AND SAVES A FIGURE CONTAINING ONLY THE POINTING CENTERS OF THE QUERY RESULTS.
   Input parameters:
   - newplot: make a new plot (newplot=1) or overplot on an existing one (newplot=0)?
   - plotcolor: color of plotting symbol: color='r' means red, and b:blue, g:green, c:cyan, m:magenta, y:yellow, k:black, w:white.
   - plotsize: size of the plotting symbol to plot.

Output:
a figure (png format) named ObsViewer_ra_<rastart>_<raend>_dec_<decstart>_<decend>-_lambda_<lambdastart>_<lambdaend>_expo_<expomin>_<expomax>_preview.png (see Figure 23.5).

NOTE: the tool buttons on the bottom of the graphical display window can be used to zoom in on parts of the figure.

5. PLOTS AND SAVES A FIGURE VISUALIZING THE QUERY RESULT AND WRITES AN OUTPUT FILE WITH THE RESULTS.
Input parameters:

- plotfov: plot the field of view (FOV) around the pointing centers (0=no, 1=yes)?
- fillfov: fill the FOV with random poisson points (to emphasize overlapping exposures) (0=no, 1=yes)?
- sec_per_dot: the number seconds exposure time per dot plot by fillfov. For example, sec_per_dot=10.0 plots a dot for each ten seconds of exposure time.
- plotcolor: color of plotting symbol: color=ŕḿeans red, and b:blue, g:green, c:cyan, m:magenta, y:yellow, k:black, w:white.
- plotsize: size of the plotting symbol to plot.

Outputs:
A graphical display window with the result.
A figure (png format) named ObsViewer_ra_<rastart>_<raend>_dec_<decstart>_<decend>-_lambda_<lambdastart>_<lambdaend>_expo_<expomin>_<expomax>.png (see Figure 23.6).
A text file (ascii format) named ObsViewer_ra_<rastart>_<raend>_dec_<decstart>_<decend>-_lambda_<lambdastart>_<lambdaend>_expo_<expomin>_<expomax>.txt.

NOTE: the tool buttons on the bottom of the graphical display window can be used to zoom in on parts of the figure.

EXAMPLE2:

1. `awe> from astro.experimental.ObsViewer import ObsViewer`
2. `awe> o=ObsViewer()`
3. `awe> o.query_database(rastart=170, raend=180, decstart=-20, decend=-20,`
   `... central_lambdastart=4000.0, central_lambdaend=5000.0,`
   `... expomin=0.0, expomax=200.0,raw='yes')`
4. `awe> o.plot(plotfov=1,fillfov=1,newplot=1,plotcolor='b',plotsize=20,sec_per_dot=1.0)`
5. `awe> o.query_database(rastart=170, raend=180, decstart=-20, decend=-20,`
   `... central_lambdastart=5000.0, central_lambdaend=6000.0,`
   `... expomin=0.0, expomax=200.0,raw='yes')`
6. `awe> o.plot(plotfov=1,fillfov=1,newplot=0,plotcolor='g',plotsize=20,sec_per_dot=1.0)`

The command lines have the following effects:

1. -.4: similar to steps in example 1.
5. Now the query is made for frames with a `central_lambda different` from step 3
6. The result from the query in step 5 is overplotted in green on the existing plot from step 4 (see Figure 23.7).

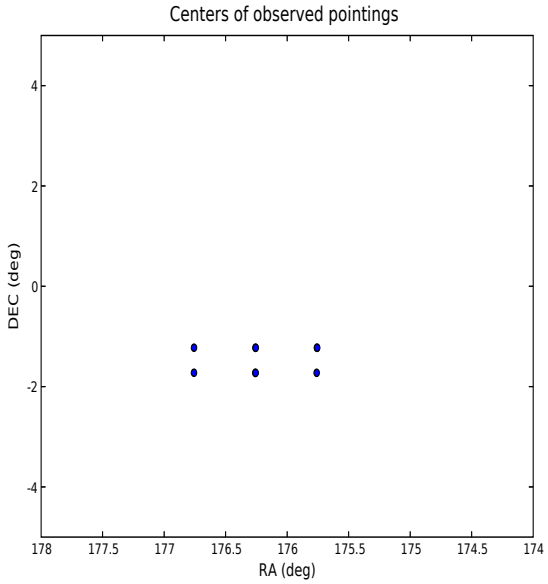



Figure 23.5: Example output figure from `awe> o.preview()`. The circles indicate the centers of the pointings.

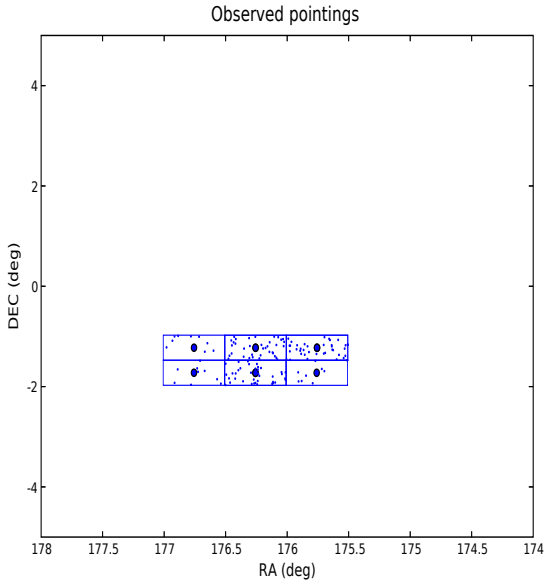



Figure 23.6: Example output figure from `awe> o.plot(plotfov=1,fillfov=1)`. The rectangles represent the FOV of the exposure to scale. The random dots (by setting fillfov=1) reflect the exposure time as explained in example 1.

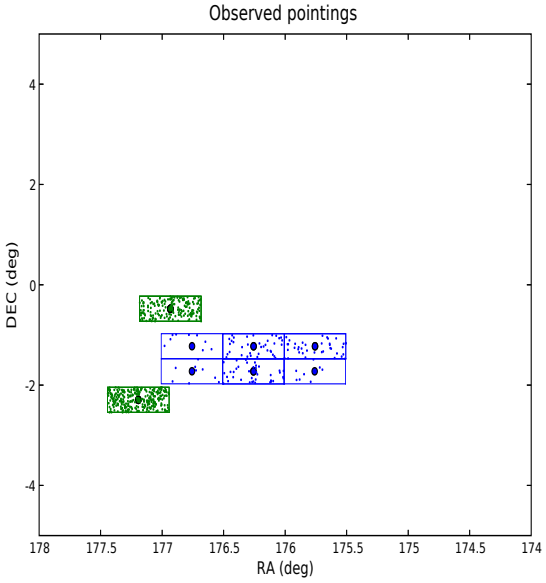



Figure 23.7: Example output figure from two queries as in example 2. The rectangles represent the FOV of the exposure to scale. The random dots (by setting fillfov=1) reflect the exposure time as explained in example 1.

## 23.7 HOW-TO perform a trend analysis and to find outliers

### 23.7.1 Summary

This HOW-TO shows how one can do a simple database trend analysis from the `awe`-prompt. In general to get to a result you need to do the following steps:

- Which quantities do you want to do a trendanalysis on? Determine which classes and attributes of objects in `AWE` are needed to get the desired information for those quantities.
- Construct the database query/queries required to get the desired information.
- Make a plot of the desired information to graphically detect outliers.
- Refine the constraints in the query to encompass only outliers.
- Retrieve the outlying objects and inspect them.

### 23.7.2 Examples

*Question 1:* Make a plot of the bias level of all raw biases of a CCD as a function of modified julian date of observation.
*Answer 1:*

```
awe> q = (RawBiasFrame.chip.name == 'ccd50')
awe> biases = list(q)
awe> x = [b.MJD_OBS for b in biases]
awe> y = [b.imstat.median for b in biases]
awe> pylab.scatter(x,y,s=0.5)
```

This results in the plot in figure 23.7.2 (zoomed, labels added).

*Question 2:* Look for raw biases for ccd50 (WFI) in 2004 for which the level of the trim section differs significantly from the level of the overscan.
*Answer 2:*

```
awe> q = (RawBiasFrame.filename.like('WFI.2004*_1.fits'))
awe> len(q)
419
awe> biases = list(q)
awe> x = [b.MJD_OBS for b in biases]
awe> y = [b.imstat.median-b.overscan_x_stat.median for b in biases]
awe> pylab.scatter(x,y,s=0.5)
```

This produces a plot as in figure 23.7.2.You can see that there seems to be one case where the difference is 5 ADU. This image will be interesting to look at. We can select it as follows:

```
awe> frames = [b for b in biases if b.imstat.median-b.overscan_x_stat.median > 4]
awe> len(frames)
2
awe> for f in frames: print f.filename
...
WFI.2004-10-15T15:10:02.248_1.fits
WFI.2004-10-15T15:11:52.384_1.fits
awe> for f in frames: f.retrieve()
...
```

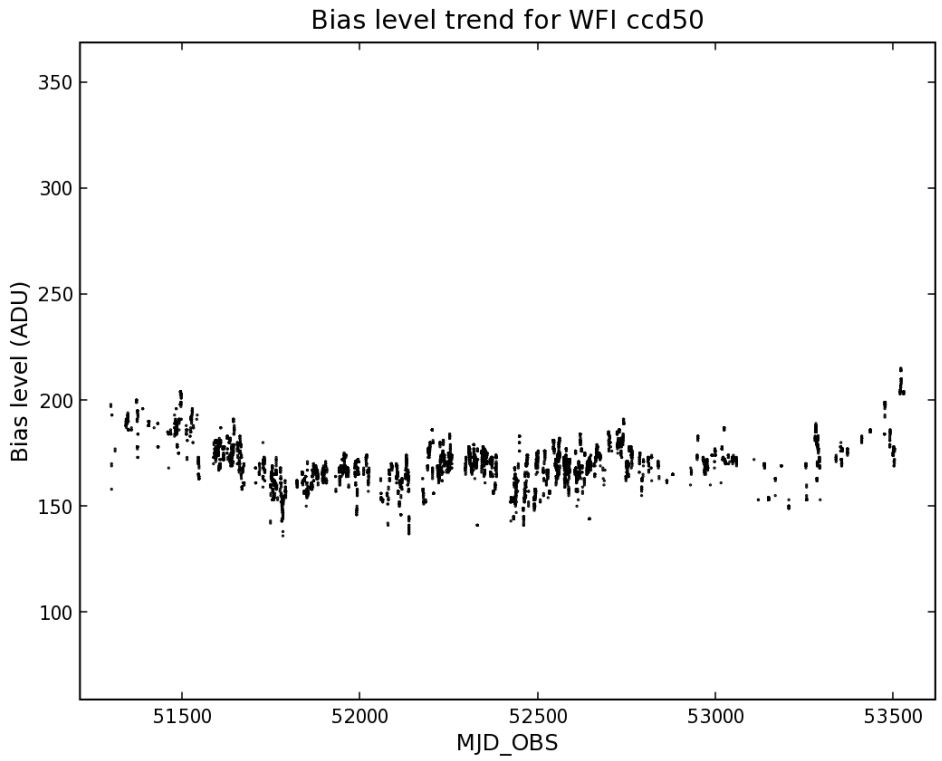



Figure 23.8: Trend analysis: bias level for all raw biases of CCD against modified julian date of observation

It turns out there are in fact two frames of this kind. The images seem to have an uncharacteristic bright region in them; something was obviously wrong during these observations.

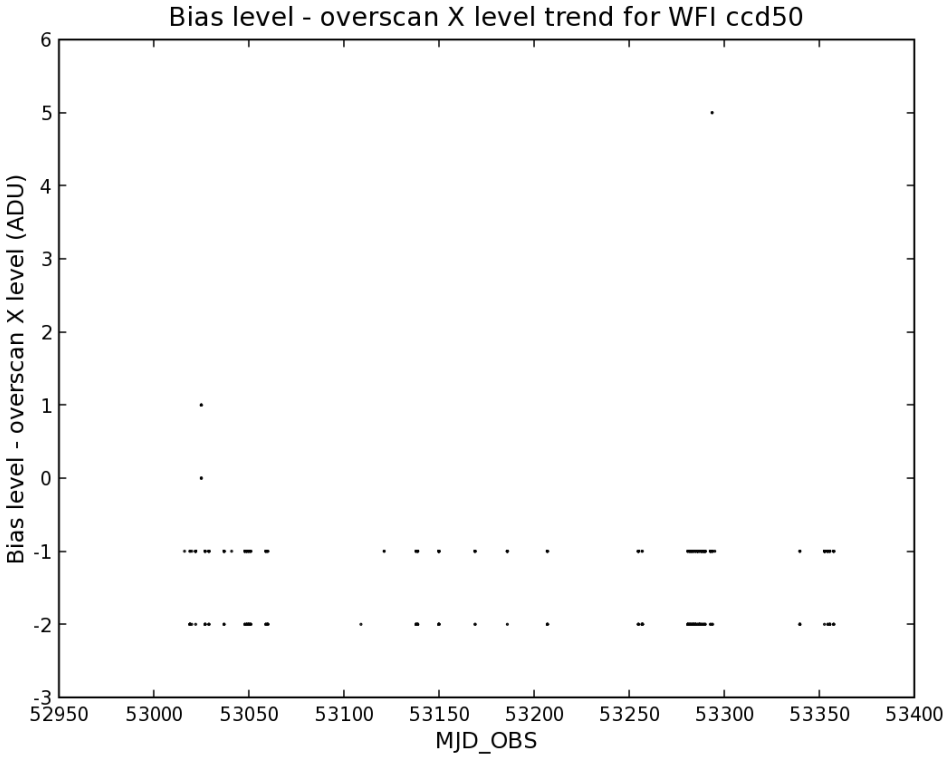



Figure 23.9: Trend analysis: trend of raw bias level in trim section minus the level in the overscan X region. The difference is plotted as a function of modified julian date of observation.

## 23.8 HOW-TO use SAMP in Astro-WISE

The Simple Application Messaging Protocol (SAMP) is a protocol for astronomical applications to collaborate. A SAMP client is available from the `awe`-prompt which allows interaction with visualization software such as Topcat and Aladin. Furthermore, a set of new SAMP messages has been designed to allow interactive query driven visualization through data pulling.

The idea behind SAMP is akin to the UNIX-philosophy that tools should do one thing, should do that thing well and communicate with other programs for things they cannot do. E.g.: A specific piece of software is responsible for retrieving the data from a data source, another program computes parameters which then get visualized with a third program.

The general principle in SAMP is that there is a central HUB, to which clients connect. The clients send messages through the HUB to other clients, which (can) respond with the result of the requested actions. There is a lot of freedom in what messages can be send, and the protocol is designed to be language agnostic. SAMP can be seen as the successor to PLASTIC (PLatform of AStronomical Tool Interaction) which had a similar goal, but never became a VO standard.

### 23.8.1 SAMP HUB and Clients

Figure 23.10 shows a diagram of the interoperability between Astro-WISE and SAMP.

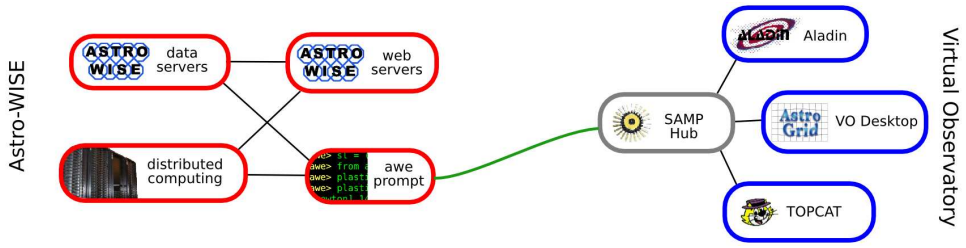



Figure 23.10: The connectivity between Astro-WISE and SAMP. On the left in red the Astro-WISE system and on the right in blue SAMP enabled applications. The SAMP HUB in the center is gray, because usually it is not a separate application itself but embedded in one of the clients. The green line is the connections between Astro-WISE and SAMP.

#### SAMP HUB

The center of the SAMP protocol is the SAMP HUB. The principal function of the HUB is to register connected applications and to relay messages inbetween them. The HUB itself does not have to have any human interface at all, let alone a GUI. The HUB is often integrated in one of the clients, e.g. Aladin and Topcat, but standalone HUBs exist as well.

#### Topcat

Topcat[2] is a table viewer/manipulator written in java. It can read a large variety of tabular data, from FITS and VOTable to common csv files. It is now developed by AstroGrid. Topcat has some nice visualization options, such as 2D or 3D scatter plots and primitive density plots. The power of the visualizations lies in the interactivity. Select points in one scatter plot, and they become automatically highlighted in another plot which might show entirely different parameters of the data points.

[2] http://andromeda.star.bris.ac.uk/~mbt/topcat/

### Aladin

Aladin[3] is an FITS image viewer developed in France by the Centre de Donn'ees astronomiques de Strasbourgh (CDS). It can view local files, connect directly to several image repositories, by the VO and otherwise. Furthermore it can receive images from other applications such as the VO Desktop through SAMP. Aladinis developed in java and can load images up to 50K by 50K pixels.

## 23.8.2 SAMP Astro-WISE integration

The awe-prompt includes a SAMP client as a Python module. This allows an astronomer to combine the large scale data handling from Astro-WISE with the visualization tools from other SAMP applications.

### Example SAMP usage in AWE

A SAMP HUB needs to be started first in order to use SAMP at all, we recommend the one in Aladin or Topcat. Besides a HUB, other SAMP enabled applications (such as Topcat or the VO Desktop) should be started as well. Start SAMP from the awe-prompt as follows:

```
awe> from astro.services.samp.Samp import Samp
awe> samp = Samp()
```

One of the most common usages of SAMP is to broadcast the data of objects to inspect them with other applications:

```
awe> sls = (SourceList.OBJECT == '2df_V_18') & \
.... (SourceList.creation_date > datetime.datetime(2007, 7, 1))
awe> sl = sls.min('creation_date')
awe> print sl
Name of SourceList : SL-HBUDDELMEIJER-0000135591
SourceList ID      : 135591
Sources in list    : 3226
Parameters in list : 35

awe> samp.broadcast(sl.frame)
[newton] 12:33:28 - Retrieving Sci-GSIKKEMA-WFI-------#843--...
awe> samp.broadcast(sl)
```

This will show the frame in Aladin with the source catalog overlaid. Topcat will have loaded the SourceList as a table.

The visualization software can be used to to select interesting sources for further investigation in the awe-prompt. With Aladin this is done by either hoovering over a source (highlighting it), or by dragging a rectangle around a group of sources (selecting them). Other software such as Topcat have similar mechanisms. The highlighted or selected sources can be requested on the awe-prompt.

```
awe> samp.highlightedSource(sl)
(135591L, 1635L)
awe> samp.selectedSources(sl)
[(135591L, 1710L), (135591L, 1773L), (135591L, 1787L)]
```

These SIDs can now be used for further processing, e.g. with GalFit.

[3] http://aladin.u-strasbg.fr/aladin.gml

```
awe> sids = [ids[1] for ids in samp.selectedSources(sl)]
awe> dpu.run('GalFit', slid=sl.SLID, sids=sids)
```

### Communication with HUB

Connecting and registering with the HUB is done automatically when the Samp class is instantiated. This can be suppressed with the `register=False` parameter. The following procedure registers manually:

```
awe> from astro.services.samp.Samp import Samp
awe> # instantiate the class, this also starts an XML-RPC server
awe> # in the background to receive messages from the HUB
awe> samp = Samp(register=False)
awe> # read the ~/.samp file for settings of the hub
awe> settingsHub = samp.getSettingsHub()
awe> for k in settingsHub:
....     print '%-20s %1s'%(k,settingsHub[k])
....
samp.hub.xmlrpc.url  http://127.0.0.1:59102/xmlrpc
samp.secret          7ce936d9b5fea215
hub.start.date       Fri Jun 19 15:16:21 CEST 2009
samp.profile.version 1.1
hub.impl             org.astrogrid.samp.hub.BasicHubService
awe> # connect to the XML-RPC server provided by the HUB
awe> XmlRpcHub = samp.connect()
awe> # register us with the HUB and declare our message subscriptions
awe> infoRegistration = samp.register()
awe> for k in infoRegistration:
....     print '%-20s %1s'%(k,infoRegistration[k])
....
samp.self-id         c3
samp.private-key     k:4_wavkccristijjefj
samp.hub-id          hub
```

Information about (other) registered clients can be requested with `getClients`. Information other clients can see from the Astro-WISE class is stored in `metadata`.

```
awe> clients = samp.getClients()
awe> for cid in clients:
....     print cid,clients[cid]['samp.name']
....
c2 topcat
c1 Aladin
hub Hub
awe> for k,v in clients['c1'].iteritems():
....     print '%-24s %s'%(k,v)
....
aladin.version           v5.926
author.email             fernique@astro.u-strasbg.fr
author.name              Pierre Fernique, Thomas Boch
home.page                http://aladin.u-strasbg.fr/
author.affiliation       CDS, Observatoire astronomique de Strasbourg
```

```
samp.documentation.url   http://aladin.u-strasbg.fr/java/FAQ.htx
samp.icon.url            http://aladin.u-strasbg.fr/aladin_large.gif
samp.description.text    The Aladin sky atlas and VO Portal
samp.name                Aladin
awe> for k,v in samp.metadata.iteritems():
....     print '%-24s %s'%(k,v)
....
author.email             buddel@astro.rug.nl
author.name              Hugo Buddelmeijer
home.page                http://www.astro-wise.org
author.affiliation       Kapteyn Astronomical Institute, Groningen
samp.name                Astro-WISE
samp.description.html    <p>Astro-WISE</p>
samp.documentation.url   http://www.astro-wise.org
samp.icon.url            http://www.astro-wise.org/pics/logo-samp-astrowise.png
samp.description.text    Astro-WISE.
```

### Sending Tables and Images

Sending data is most easily achieved with one of the `broadcast` functions. The `broadcast` functions send data to all clients (that can handle that datatype).

- `broadcastSourceList(sourcelist, filename=None, tableid=None, name=None)`: converts sourcelist to an VOTable and uses `broadcastVOTable` to send it.
- `broadcastSourceCollection(sourcecollection, filename=None, ..)`: converts sourcecollection to an VOTable and uses `broadcastVOTable` to send it.
- `broadcastCatalog(catalog, filename=None, tableid=None, ..)`: converts catalog (a PhotSrcCatalog) to an VOTable and uses `broadcastVOTable` to send it.
- `broadcastTableConverter(tableconverter, filename=None, tableid=None, name=None)`: converts tableconverter to an VOTable and uses `broadcastVOTable` to send it.
- `broadcastVOTable(filename, tableid=None, name=None)`: Sends a VOTable using the `table.load.votable` message.
- `broadcastFrame(frame, filename=None)`: Downloads the frame (as FITS) and sends it using `broadcastImage`.
- `broadcastImage(filename)`: Broadcasts a FITS image.
- `broadcast`: This function accepts an object that is either a SourceList, a SourceCollection, a Frame, a PhotSrcCatalog or a TableConverter and dispatches it to the relevant function.

The `filename`, `tableid` and `name` keywords are respectively the filename where the (intermediate) VOTable or FITS file is saved, an identifier for the table and a name to display in the other programs.

At the time of writing, the `table.load.fits` message is not fully supported in all SAMP applications, therefore all `broadcast` functions send only VOTables. The (tabular) data is converted to a 'TABLEDATA' votable. All broadcasted files (VOTables and fits images) are stored locally on disk, of which the location is broadcast through SAMP.

### Sending Interaction Messages

Interactivity between different SAMP programs for tabular data can be achieved with the `highlight` function (which sends the `table.highlight.row` message) and the `select` function (which sends the `table.select.rowList` message). Their first argument is a 'table key' (see section 23.8.2), the second one SID or a list of SIDs respectively.

```
awe> samp.highlight(sl, 1234)
awe> samp.select(sl, range(1000,1100))
```

The `pointAtSky` function uses the `coord.pointAt.sky` message to let other applications point to a certain position on the sky. E.g., Aladin will center the active image plane on the position.

```
awe> samp.pointAtSky(123.0, -20)
```

### Sending General Messages

SAMP describes 4 methods to send messages (of any type) which are mapped to similar function names which can be used to send messages manually. This is usefull to send messages that are not supported by the class (yet). Since the message types are agreed upon between individual applications, any application developer (you) can create their own messages. All the above commands use the `callAll` method to send their requests.

- `notify(receiverid, message)`: Sends the message to one specific receiver, it is not possible to reply.
- `call(receiverid, message)`: Sends the message to one specific receiver. The receiver is expected to reply, but the program (the `awe`-prompt) continues.
- `callAll(message)`: Sends the message to all clients that have registered for this MType. The receivers are expected to reply, but the program (the `awe`-prompt) continues.
- `callAndWait(receiverid, message, timeout=20)`: Sends the message to one specific receiver. the program (the `awe`-prompt) waits at most timeout seconds on the recipient to reply.

The example below sends an Aladin script with several methods (it is assumed client 'c1' is Aladin). In the case of `call` it is likely that only Aladin will be registered to this `mtype` and thus all example will achieve the same thing. (Sending scripts through SAMP is an undocumented feature of Aladin).

```
awe> message = {
....   'samp.mtype': 'script.aladin.send',
....   'samp.params':{'script':'zoom 16x'}
.... }
awe> samp.callAll(message)
awe> samp.call('c1',message)
awe> samp.callAndWait('aladin',message,10)
```

The `receiverid` parameter is the id of the receiver. These can be found with the `getClients` method. For the Aladin hub these are just the letter c followed by a number. For convenience all above function use the `fixReceiverID` which allow the name of the client (as found through `getClients`) as the receiverid. For Topcat and Aladin it is also possible to use the shortcuts `topcat` and `aladin` as `receiverid`.

The `notify` command does not allow replies, but it is implemented in the same way as the `call` method, so replies that are send anyway are still stored. The `callAndWait` functionality is mainly available for non multithreading applications. It can also be useful if it is essential to wait for the answer, so it is implemented in the Python class as well. Some applications (such as Aladin) do not always reply, so make sure to set a timeout.

### Receiving Messages

Tables can be send to the `awe`-prompt with the `table.load.votable` message. In Topcat this can be done by broadcasting a table or a subset. In Aladin by broadcasting the relevant plane. The table itself is only stored if the `load_tables` member of the SAMP instance is set to `True`.

For interactivity for tables, the Python class can receive the `table.highlight.row` and `table.select.rowList`. The chosen sources are stored in the `tables` member, see section 23.8.2.

In Aladin highlighting a source is done by hoovering over a source, in Topcat this can be done by setting 'Activation Action' to 'Transmit Row' and selecting a row. Selecting multiple sources in Aladin is done by dragging a green rectangle around the sources, this is automatically broadcast with the `table.select.rowlist` message. In Topcat it is possible to send subsets as selections either when creating the subset or later from the 'subsets' window.

Retrieving highlighted and selected sources can be done with similarly named functions

- `highlighted(tableid)`: Returns the latest highlighted source from table tableid as an integer row number (SID in the case of a SourceList).
- `selected(tableid)`: Returns the latest selected sources from table tableid as a list of integer row numbers (SIDs in the case of a SourceList).
- `highlightedSource(tableid)`: Returns the `SLID-SID` combination of the latest highlighted source.
- `selectedSources(tableid)`: Returns the `SLID-SID` combinations of the latest selected sources.

### Storing Information

Of every table that is send through SAMP, the class records its properties in the `table` member.

```
awe> for k,v in samp.tables[sl].iteritems():
....     print "%-12s %s"%(k,v)
....
name         SourceList-135591
url          file://localhost/Users/users/buddel/SL-135591-2df_V_18.votable
tableid      SourceList-135591
type         votable
selected     [1710, 1773, 1787]
highlighted  1635
```

The `url`, `tableid` and `name` are described in the `table.load.votable` message type. They are respectively the location of the intermediary votable, the id to be used within SAMP and the name to be displayed in the applications. The `type` denotes the type, at this moment only `votable` is supported.

The `highlighted` key stores the id of the last source that has been highlighted with the `table.highlight.row` message as an integer. The `selected` key stores the ids of the last set

of sources that have been selected through the `table.select.rowList` message as a list of integers. In the case of SourceLists, the ids are the same as the SIDs.

The primary key of the `tables` dictionary is the SAMP table-id which is stored in the `tableid` key. For Python objects (SourceList, PhotSrcCatatalog, TableConverter), the object itself can be used as key as well. For SourceLists, the SLID (as integer) is also a key.

All messages that are send are stored in the `messagesSend` member, including with possible replies. All received messages are stored in `messagesReceived`. The messages are stored as a dictionary, of which the actual message is stored as an item. An example of a message in `messagesSend`:

```
{
    'messageTag': 2,
    'messageId': {'c1': 'c3_A_16a8_2'},
    'type': 'callAll'}
    'receiverId': 'all',
    'time': 1245661731.7381041,
    'message': {
        'samp.params': {
            'url': 'file://localhost/[..]/SL-135591-2df_V_18.votable',
            'table-id': 'file://localhost/[..]/SL-135591-2df_V_18.votable',
            'name': 'SourceList-135591'
         },
        'samp.mtype': 'table.load.votable'
    },
    'replies': {
        'c1': {'samp.status': 'samp.ok', 'samp.result': {}},
        'Aladin': {'samp.status': 'samp.ok', 'samp.result': {}},
        'aladin': {'samp.status': 'samp.ok', 'samp.result': {}}
    },
}
```

`messageTag` is an unique identifier for our client (consecutive integers starting from 1), `messageId` is the unique id of the message given to it by the HUB (it the return value for the `call` methods). The `type` denotes how the message is send, broadcasted to all clients in this case, `receiverId` denotes who got the message, everyone in this case. `time` is the value returned by `time.time()` on sending the message ("current time in seconds since the Epoch"). The actual message is stored with key `message`. The replies are stored in `replies` which is a dictionary with as keys the SAMP client-id (`c1`), the client name (`'Aladin'`) and in some cases shortcut names (`'aladin'`). Note that only the client-id is always unique.

### 23.8.3 Query Driven Visualization through SAMP

The `awe`-prompt SAMP client accepts several new message types to pull data and to inspect and influence its derivation over SAMP. The paradigm of Target Processing includes data pulling with full data lineage, which can be abstracted well. These messages could therefore easily be used in other information systems as well and are described here in a general form, highlighting the Astro-WISE specifics.

Within Astro-WISE, the messages are currently only applicable to SourceCollections. In the SourceCollection HOW-TO (22.10) a set of prototype applications that use these messages are shown.

### 23.8.4 Data Pulling Messages

There are two new messages to pull catalog data out of the information system.

- `catalog.pull`: Pull a catalog and send it over SAMP using one of the `table.load.*` messages. This message requires the following parameters, detailed below: an identifier of a catalog to select the sources from, a selection criterion and a list of requested attributes of the sources. The `awe`-prompt SAMP cliet will create a dependency tree of SourceCollections whose end node contains the requested catalog.
- `catalog.derive`: Perform the same action as `catalog.pull`, but without sending the catalog data over SAMP.

The `.derive` message is useful when it is necessary to inspect or modify the derivation of the catalog—using the messages in section 23.8.5—before visualization. These two messages require three parameters which we should elaborate on:

- `catalog-id`: An identifier of the base catalog to select the sources from. For the Astro-WISE SAMP client this has to be the SCID of a SourceCollection. This could be extended in the future, for example by referring to an observation.
- `query`: A selection criterion to specify which sources of the original catalog are requested. This should be a logical expression referencing the `attributes` below. For the Astro-WISE SAMP client this should be a string that is suitable for use in a FilterSources SourceCollection.
- `attributes`: A list of requested attributes (parameters) of the sources. It is not required that the catalog corresponding to the `table-id` contains these attributes. The attributes should be specified as a comma-separated list of attribute names for the Astro-WISE SAMP client.

### 23.8.5 Object Messages

Several SAMP message types are defined for interaction with an information system that store data through persistent objects. These messages allow the visualization software to gain information about the objects and inspect or even influence its processing. Although the messages are designed to be applicable for any object, they are currently only supported for SourceCollections. The persistent object related message are:

- `object.highlight`: Highlight an object.
- `object.info`: Return information about an object, see below.
- `object.change`: Change the value of a property of an object such as a process parameter or a dependency.
- `object.action`: Perform an action related to an object or property. Possible actions are retrieved using the `object.info` message.

The `object.highlight` message can be send to any application, the others are supposed to be send to the information system only.

#### SAMP `object.info` Data Structure

A specific SAMP map is defined as a return value for the `object.info` message, containing information about the object and its properties. This is generated by the `get_export()` function of the Astro-WISE classes, which is currently only implemented for SourceCollections. For the object itself it includes information about its processing status, whether the object can be modified and what properties it has.

The properties of an object include process parameters and references to the progenitors of the object. The returned information of a property include its name, current value and optionally other values it can be set to. Furthermore the information system can define actions that can be performed on the object or its properties.

### 23.8.6 More and future features

In the `subscriptions` member it is stored which SAMP message type gets mapped to which function. By updating this dictionary before registering with the HUB, it is possible to hook your own functions to specific types. Perhaps a better hooking mechanism would be useful.

The class can also be used standalone, then it can act as a proxy for non-interactive clients. These clients can connect to the XML-RPC server to request which sources are highlighted by other applications and such. There is only preliminary support for this now.

Currently, only one highlighted row and one selected row list is stored. Perhaps in the future highlighted and selected sources will be stored per SAMP client.

### 23.8.7 SAMP Protocol

SAMP is a simple but extensible protocol, using a client-server model based on application defined messages. We give a short description of the protocol to the extend that suffices how the protocol works from the user perspective. For full details, refer to the official documentation [4] or the official wiki [5].

Clients register with the SAMP hub and register for certain types of messages. Clients can than send messages to individual clients, or to any client that has registered for that kind of message. If necessary, receiving clients can send a response to the sender of a message they receive.

SAMP is in principle language agnostic and is based on abstract interfaces. That is, it specifies which functions the HUB and the clients must have in order to send and receive messages, but not the exact protocol that the HUB and client use to call those functions. The rules which describe how SAMP functions are mapped to the internally used protocol is described in a SAMP 'Profile'. One standard profile based on XML-RPC is described in the official documentation, and this is what is used in Astro-WISE.

#### SAMP datatypes

Because SAMP is language and even communication protocol agnostic, the number of datatypes that are supported is very limited. The only three data types are

1. `string` — a scalar value consisting of a sequence of ASCII-characters.
2. `list` — an ordered array of data items.
3. `map` — an unordered associative array with a string as key.

[4] http://www.ivoa.net/Documents/latest/SAMP.html
[5] http://www.ivoa.net/cgi-bin/twiki/bin/view/IVOA/SampInfo

Other scalar types have to be mapped to strings, and there is a specification to represent integers, floats and booleans as strings. These data types can be nested to any level, e.g. it is possible to have a map with lists as values.

### SAMP messages

There is a small set of predefined message types, most (astronomically useful) message types are defined between application developers themselves. More or less agreed upon message types can be found on the SAMP wiki [6]. SAMP message types are abbreviated to MTypes.

An MType looks syntactically like a dot-separated string such as `samp.hub.event.shutdown` or `table.load.votable`. MTypes that start with `samp.` are administrative messages defined by the protocol, others are defined by application authors. Most messages have arguments such as the name of the file to be loaded. Messages can require the receiving end to send a return value, but so far no widely accepted MType does this. However, clients are supposed to give a general reply with success or failure of a requested operation even if no response is required.

### SAMP administrative messages

Some important administrative messages are

- `samp.hub.event.register`
  The hub broadcasts this message every time a client successfully registers.
  Arguments:
  - `id` (string) Public ID of newly registered client
- `samp.hub.event.unregister`
  The hub broadcasts this message every time a client unregisters.
  Arguments:
  - `id` (string) public ID of unregistered client
- `samp.hub.event.subscriptions`
  The hub broadcasts this message every time a client declares its subscriptions.
  Arguments:
  - `id` (string) public ID of subscribing client
  - `subscriptions` (map) new subscriptions declared by client

### SAMP application messages

The relevant application defined messages for our purposes are

- `table.load.votable`
  Loads a table in VOTable format.
  Arguments:
  - `url` (string) URL of the VOTable document to load
  - `table-id` (string) (OPTIONAL) identifier which may be used to refer to the loaded table in subsequent messages
  - `name` (string) (OPTIONAL) name which may be used to label the loaded table in the application GUI

[6] http://www.ivoa.net/cgi-bin/twiki/bin/view/IVOA/SampMTypes

- `table.load.fits`
  Loads a table in FITS format.
  Arguments:
  - `url` (string) URL of the FITS file to load
  - `table-id` (string) (OPTIONAL) identifier which may be used to refer to the loaded table in subsequent messages
  - `name` (string) (OPTIONAL) name which may be used to label the loaded table in the application GUI
- `table.highlight.row`
  Highlights a single row of an identified table by row index. The table to operate on is identified by one or both of the table-id or url arguments. At least one of these must be supplied; if both are given they should refer to the same thing. Exactly what highlighting means is left to the receiving application.
  Arguments:
  - `table-id` (string) identifier associated with a table by a previous message (e.g. table.load.*)
  - `url` (string) URL of a table
  - `row` (SAMP int) Row index (zero-based) of the row to highlight.
- `table.select.rowList`
  Selects a list of rows of an identified table by row index. The table to operate on is identified by one or both of the table-id or url arguments. At least one of these must be supplied; if both are given they should refer to the same thing. Exactly what selection means is left to the receiving application.
  Arguments:
  - `table-id` (string) identifier associated with a table by a previous message (e.g. table.load.*)
  - `url` (string) URL of a table
  - `row-list` (list of SAMP int) list of row indices (zero-based) defining which table rows are to form the selection
- `image.load.fits`
  Loads a 2-dimensional FITS image.
  Arguments:
  - `url` (string) URL of the FITS image to load
  - `image-id` (string) (OPTIONAL) Identifier which may be used to refer to the loaded image in subsequent messages
  - `name` (string) (OPTIONAL) name which may be used to label the loaded image in the application GUI
- `coord.pointAt.sky`
  Directs attention (e.g. by moving a cursor or shifting the field of view) to a given point on the celestial sphere.
  Arguments:
  - `ra` (SAMP float) right ascension in degrees
  - `dec` (SAMP float) declination in degrees

### 23.8.8 Query Driven Visualization Message Details

We designed new SAMP messages and data structures to enable query driven visualization through data pulling mechanisms. The `object.*` messages assume that the information system uses an object oriented model for science products such as catalogs.

#### QDV SAMP: Message Types

The proposed messages are:

- `catalog.derive`: Create a catalog through data pulling. Arguments:
  - `catalog-id` (string): Identifier of the catalog to select the sources from.
  - `query` (string): Selection criterion for the sources.
  - `attributes` (list of strings): Names of the attributes.
- `catalog.pull`: Perform the same action as `catalog.derive` and send the data over SAMP. Arguments:
  - `catalog-id` (string): Identifier of the catalog to select the sources from.
  - `query` (string): Selection criterion for the sources.
  - `attributes` (list of strings): Names of the attributes.
- `object.highlight`: Highlight an object. Arguments:
  - `class` (string): Class of the object.
  - `object-id` (string): Identifier of the object.
- `object.info`: Returns a SAMP map with information about an object as described below. Arguments:
  - `class` (string): Class of the object.
  - `object-id` (string): Identifier of the object.
- `object.change`: Change a property of an object. Arguments:
  - `class` (string): Class of the object.
  - `object-id` (string): Identifier of the object.
  - `property-id` (string): Identifier of a property of the object.
  - `value` (string): New value of the property.
- `object.action`: Perform an action related to an object. Arguments:
  - `class` (string): Class of the object.
  - `object-id` (string): Identifier of the object.
  - `property-id` (string): Identifier of a property of the object.
  - `action-id` (string): Identifier of the action.

### QDV SAMP: Data Format

SAMP data structures are defined to send information about objects between applications. The structures are designed to be generic enough that they could be used for any information system.

Information about an object itself, e.g. the response to an `object.info` message, is communicated through a map with the following keys:

- `class` (string): The class of the object. A client that has knowledge about the used classes could handle known classes in a special way.
- `id` (string): Identifier this object, unique in combination with the class.
- `status` (string): Indication the processing status of this object (see below).
- `properties` (list of maps): Properties of this object (see below).
- `actions` (list of maps): Actions that can be performed on this object (see below).
- `readonly` (boolean): Flag to indicate that the object cannot be modified.

Properties of an object, for example process parameters, are described with a map with the following keys:

- `name` (string): Name of the property, as used in the object.
- `class` (string): The class that the value of the property should have, or a primitive such as 'int'.
- `description` (string): A human readable description of the property.
- `value` (string): The used value for the property. This is the `id` of the object if the property references to a persistent class.
- `options` (list of maps): Possible values for the property, if applicable (see below).
- `actions` (list of maps): Actions that can be performed on the property (see below).
- `readonly` (boolean): Flag to indicate that the property cannot be modified.

An action that can be performed on an object or property is defined by a map with the following keys:

- `id` (string): A unique identifier for this action.
- `name` (string): A human presentable name for this action.

### QDV Samp: Object Status

The status value of an object refers to the processing status of the object. It can have the following values:

- `ok`: The object has been processed, or can be processed while retrieving the result.
- `automatic`: The object has to be processed before the can be retrieved. This can be done without user interaction.
- `new`: This is a non persistent object, which can be processed without user interaction.

- `depends`: This is a new object, which can be processed only after human intervention. For example to set a process parameter that has no proper default.
- `not`: As it is, this object cannot be processed, e.g. because a dependency cannot be fulfilled. The scientist might be able to solve the problem, but whether this is the case is not clear to the information system.
- `unknown`: The status is unknown.

The `awe`-prompt SAMP client currently does not support the `status` property, it returns `unknown` on all objects.

### QDV Actions

The actions value of the dictionaries refer to actions that can be performed through the `object.-action` message.

## 23.9 HOW-TO use Query Driven Visualization in Astro-WISE

Query Driven Visualization is an extension of the request driven Target Processing to Visualization. Instead of *pushing* data into the visualization it is *pulled* from within the visualization.

Query Driven Visualization for SourceCollections (secion 22.10 in Astro-WISE can be done over SAMP (section 23.8).

A design goal of the SourceCollection classes was to be able to use them interactively over SAMP. New SAMP messages are designed to to allow query driven visualization in a more declarative way than is possible with other information systems. Support for these SAMP messages can easily be implemented in a visualization tool, here we use simple standalone programs to perform query driven visualization of catalogs. The exploration of the catalogs is deferred to other applications, such as TOPCAT, by sending the catalogs through SAMP.

### 23.9.1 Bootstrapping SAMP

A SAMP HUB and the `awe`-prompt SAMP client should be started to perform Query Driven Visualization. TOPCAT can be used as a SAMP HUB, which has the added benefit that TOPCAT is a useful tool to explore the catalogs itself. TOPCAT can be started through java webstart:

```
javaws http://andromeda.star.bris.ac.uk/~mbt/topcat/topcat-full.jnlp &
```

The `awe`-prompt adds command line interaction to the query driven visualization. Run the following on the `awe`-prompt:

```
# Import the SAMP module.
from astro.services.samp.Samp import Samp

# Start a SAMP client.
s = Samp()
```

The programs below reside in the Astro-WISE code base, even though they have no dependencies on Astro-WISE-code. It is assumed that the Astro-WISE code base can be found in `$AWEPIPE`. Set `$AWEPIPE` to the value that is returned by the following command, in case it has not been set already:

```
awe -c "import os; print(os.environ['AWEPIPE'])"
```

### 23.9.2 Simple Puller

The *Simple Puller* (figure 23.11) is a SAMP application with a web based frontend for pulling catalog data. This application shows the most basic way to pull catalog data over SAMP, which can easily be added to other SAMP applications. It can be found on the CVS:

```
cd $AWEPIPE/astro/services/qdvsamp/simplepuller/
python simplepuller.py
```

Browse to http://localhost:8080 to use it.

Provide the following pieces of information:

- Starting Catalog: 100511
- Selection Criterion: "R" ¡ 300
- Attributes: absMag_u, absMag_g, iC

And press the 'Pull' button.

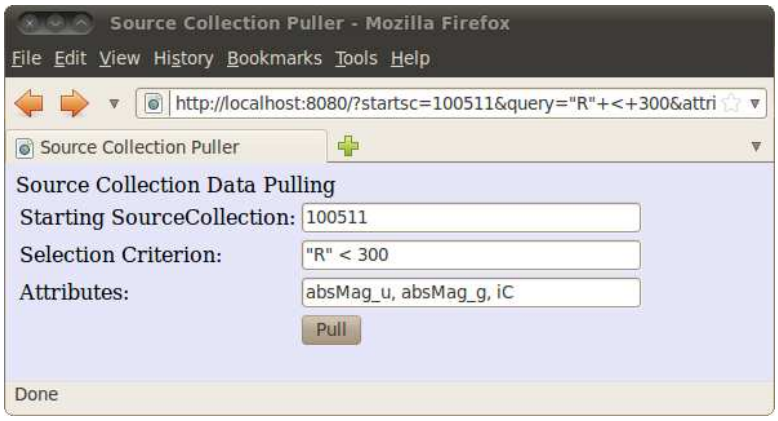



Figure 23.11: SAMP application for pulling catalog data.

### 23.9.3 Tree Explorer

The *Tree Explorer* is a simple program that allows exploration of the dependency tree held by the Astro-WISE SAMP client (Figure 23.12). Highlighting a node by clicking on it sets the `highlighted_sourcecollection` property of the Astro-WISE client. This application shows how a SAMP application can use the data lineage to give the user more information about the derivation of a particular dataset. The Tree Explorer can be found in the Astro-WISE code base:

```
cd $AWEPIPE/astro/services/qdvsamp/treeexplorer
python treeexplorer.py
```

### 23.9.4 Object Viewer

The *Object Viewer* (figure 23.11) is a SAMP application with a web based frontend for viewing and modifying details of individual Process Targets. Currently only SourceCollections are supported. This application shows how an application can use SAMP to influence processing details, without requiring it to know details of Astro-WISE.

```
cd $AWEPIPE/astro/services/qdvsamp/objectviewer
python objectviewer.py
```

Browse to `http://localhost:8084/objectviewer.html` to use it.

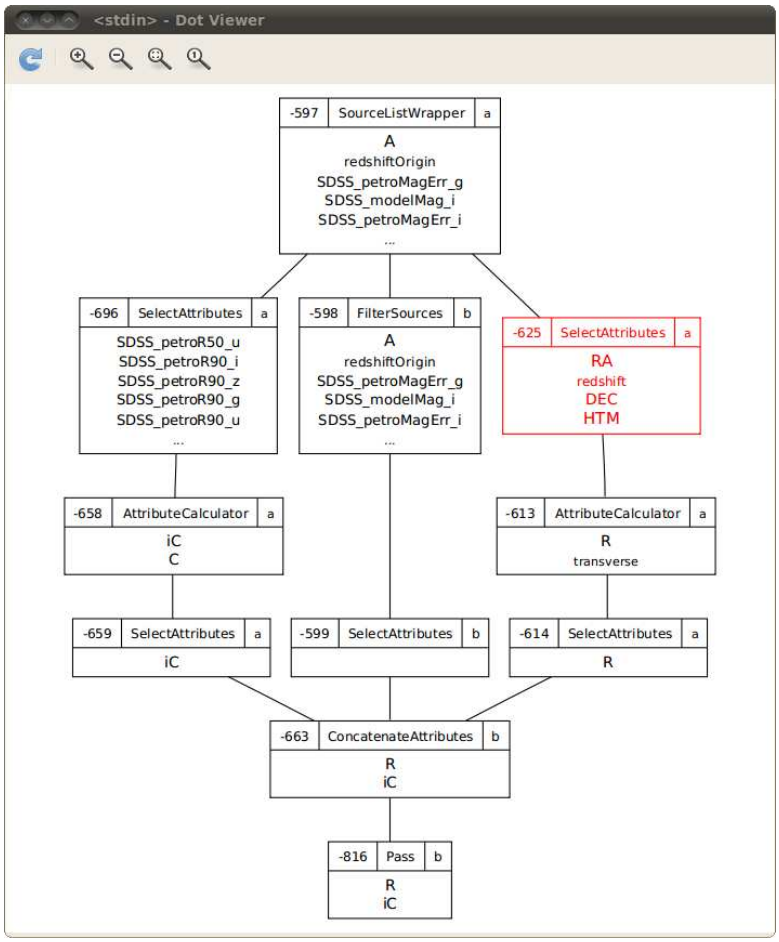



Figure 23.12: SAMP application for exploring trees. Every node shows the SCID on the top left, the operator in the top center and an identifier for the set of sources on the top right. The attributes are shown in the rest of the box.

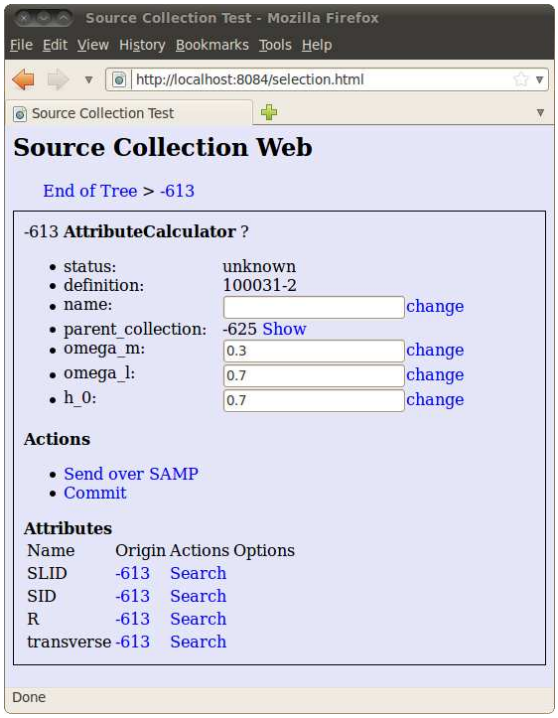



Figure 23.13: SAMP application to view and modify details of individual Process Targets. The highlighted SourceCollection from figure 23.12 is shown. This figure shows a prototype of the application.

# Chapter 24

# Development

## 24.1 HOW-TO Define a new instrument in the Astro-WISE System (to enable ingesting data from it)

### 24.1.1 Summary

Three steps are required:
(1) Check format of FITS data to be ingested.
(2) Put information about the new instrument into a template file.
(3) Send the file to Danny Boxhoorn (danny@astro.rug.nl) to discuss finishing steps.

### 24.1.2 Defining a New Instrument

Step (1): the data to be ingested (so-called "RawFitsData") should be preferentially multi-extension FITS data. Furthermore, it is required that all data of an instrument have the same size (i.e., same NAXIS1 and NAXIS2). If the above is not the case please contact Danny Boxhoorn.

Step (2): requires making a new file called `HeaderTranslator<instrumentname>.py` in `$AWEPIPE/astro/instrument`. This file gives a description of the instrument properties which Astro-WISE can read. This is mostly done by listing which FITS header keyword from the new instrument corresponds to which keyword used by Astro-WISE. The file `HeaderTranslatorOCAM.py` for the OmegaCAM instrument can serve as template. Additional information can be obtained as follows. For general information read the descriptive text in HeaderTranslator.py. For general information on chip specifications read the descriptive text in `$AWEPIPE/astro/main/Chip.py`. In the HeaderTranslator the `obs_type_id_keys` refer to the 5 frame categories- "Bias", "Dome", "Twilight", "Dark" en "Science"- which are used during ingestion. Please see the HOW-TO on ingestion (at §7.5 for further details. Please do not adjust the `HeaderTranslator<instrument>.py` file either for a new or existing instrument in the CVS version of the code before contacting Danny Boxhoorn.

Step (3): send the file `HeaderTranslator<instrumentname>.py` to Danny Boxhoorn (danny@astro.rug.nl)

Finally, see the HOW-TO section "Ingesting" (at §7.5) for information on how to ingest data into the Astro-WISE system for the now defined instrument.

# Chapter 25

# Frequently Asked Questions

## 25.1 General

### 25.1.1 Introductory Material

- Q: What is Astro-WISE?

  A: Astro-WISE for Astronomical Wide-field Imaging System Europe, is an E.U. consortium and associated information system dedicated to the handling of wide-field image data resulting from large astronomical surveys. Please see the main page for a complete description.

- Q: What is the Astro-WISE Environment?

  A: The Astro-WISE Environment, or simply `AWE`, is the information system built to handle this flood of astronomical data. Start here to learn more about it.

- Q: What language is the Astro-WISE Environment written in?

  A: The Astro-WISE Environmentis written mainly in the Python programming/scripting language. There are also Python utility modules written in C (see the HOW-TO Introduction), but these are wrapped in Python to interface with the system.

- Q: What kind of web-services does `AWE` offer?

  A: See the Introduction HOW-TO for a complete list of what `AWE` offers.

### 25.1.2 Getting Started

- Q: I can't access anything but the basics in the `AWE`, nor can I access the database. What's wrong?

  A: Do you have an `AWE` account? If not, please contact an Astro-WISE representative.

- Q: I try to access the anonymous CVS server for a checkout, but it appears to need a password. What is it?

  A: Anonymous CVS requires a special password and is not available to just anybody. Contact an Astro-WISE representative for help. Anonymous CVS is read-only access to the software. When ready to contribute to the code-base, you will need a full account. Again, see an Astro-WISE representative.

- Q: I keep seeing references to `$AWEPIPE`. What is it?

  A: `$AWEPIPE` is the environmental variable that should point to your "awe" checkout, the code-base for `AWE` (e.g., ∼/awe/).

- Q: How do I start `AWE`?

  A: There are many options, but the typical one is to simply use `AWE` on a properly configured system:

```
...]$ awe
Python 2.3.5 (#4, Aug 15 2005, 11:45:46)
[GCC 3.4.3 20050227 (Red Hat 3.4.3-22.1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.


                Welcome to the Astro-WISE Environment
.
.
.
awe>
```

  You can also temporarily modify environmental variables that `AWE` uses so that `AWE` starts with a different configuration:

```
...]$ env AWEPIPE=~/test-awe/ awe

or

...]$ env database_engine= awe
```

  The first example causes `AWE` to use a code-base located in a specified directory (∼/test-awe/ here) and the second disables use of the database (if you need to work offline for some reason).

### 25.1.3 Documentation

- Q: I'm a new user. Where should I look for documentation?

  A: The first place any new user should look for documentation is the HOW-TOs

- Q: What are docstrings?

  A: Docstrings are the inline comments added to Python code that allow for an automated documentation system called PyDoc. Docstrings are contained within *triple quotes*:

```
'''this is a docstring'''
"""this is also a docstring"""
# this is just a comment
```

### 25.1.4 Concurrent Versions System (CVS)

- Q: What is the address of the Astro-WISE CVS server?

  A: cvs.astro-wise.org

- Q: What is the proper way to fully update a code-base?

  A: First, make sure you are in the top-level directory (or whatever directory you want fully updated) and issue the following command:

  ```
  ...]$ cvs -q update -dPA
  ```

  where:

  **-q** suppresses excessive verboseness (option to **cvs**)
  **-d** create any missing directories (option to **update**)
  **-P** prune empty directories (option to **update**)
  **-A** reset any "sticky" tags (option to **update**)

  If you chose a specific tagged version, that tag becomes sticky. The last option overrides this stickiness. See the cvs(1) man page for complete information on CVS.

- Q: I am installing the Astro-WISE system on my local machine, but when I try to install <*insert program name here*>, I get the comment that some file or directory is missing.

  A: are you sure that you retrieved every single sub-directory of the Astro-WISE system during checkout? Using a simple `cvs co` can sometimes result in some sub-directories not being retrieved. If you suspect that something like this might be the problem, go to the directory where the specific sub-directory should have been located and type `cvs update -d`.

- Q: How do I switch between the latest `$AWEPIPE` version and the AWBASE version?

  A: To switch to the most recent check out use

  ```
  ... awe]$ cvs -q update -dPA
  ```

  and to switch back to the AWBASE version, use

  ```
  ... awe]$ cvs -q update -r AWBASE -dP
  ```

  in the 'awe' directory.

### 25.1.5 Data Preparation

- Q: I am about to go observing with an `AWE`-supported instrument. Is there anything special that I should do?

  A: Please look at the Observations Scheduling HOW-TO for guidelines.

- Q: I want to process the data I took on <*insert favorite instrument*> with `AWE`. What do I need to do?

  A: `AWE` was created specifically for OmegaCAM and will handle OmegaCAM data "out-of-the-box". There is also support for the instruments listed on the Supported Data Sources page. If your instrument is not there, contact an `AWE` representative for verification and have a look at the New Instrument HOW-TO to get started.

### 25.1.6 Ingesting

- Q: What is data ingestion and how do I do it?

  A: In order for raw data to be processed in the Astro-WISE system, it must first be *ingested*, or imported into the system. Please see the Data Ingestion HOW-TO for a complete description.

- Q: How can the data be categorized for ingestion in `AWE`?

  A: Data in `AWE` is catagorized by purpose:

**readnoise** bias frames designated to determine the instrument read noise

**bias** a raw bias or zero second exposure frames

**dark** a frame taken to measure the dark current of the instrument

**gain** a specific series of frames designed to determine the gain of the instrument

**dome** a raw flat frame taken from a screen mounted within the dome enclosure

**twilight** a raw flat frame taken at twilight on the sky

**science** any frame taken for the purpose of science (excluding photometric calibration)

**photom** a science frame taken with the specific purpose of photometric calibration

- Q: After I ingested my data, all the filenames were different. What happened?

  A: The filenames were converted to `AWE` cannonical names of the form:

  ```
  <instrument>.<date_obs>.fits

  or

  <instrument>.<date_obs>_n.fits
  ```

  The first from is for multi-extension FITS images, the second is for single-extension FITS images where **n** is the extension number.

### 25.1.7 Dates and Times

- Q: How are observation dates defined in `AWE`?

  A: An observation night is based on the local date at sunset and not on UTC. See the Dates and Times HOW-TO for more information.

- Q: How are dates stored in the database?

  A: In UTC only.

### 25.1.8 Parallel Processing

- Q: What is the parallel processing interface in `AWE`?

  A: The DPU (Distruibuted Processing Unit) is used to run tasks on the parallel compute cluster. It is initialized when `AWE` starts:

```
...]$ awe
.
.
.
Importing Astro-WISE packages. Please wait...

Initializing Distributed Processing Unit...

Current profile:
.
.
.
awe>
```

To initiallize it manually, use this method:

```
awe> from astro.recipes.mods.dpu import Processor
awe> my_dpu = Processor(Env['dpu_name'])
```

- Q: How do I find the instruments, task identifiers, and options supported by the DPU's run method: dpu.run()?

  A: This information is somewhat hidden. The supported instrument list can be seen by querying the `HeaderTranslatorFactory`:

  ```
  awe> from astro.instrument.HeaderTranslatorFactory import supported_instrument_list
  awe> print supported_instrument_list
  ```

  Task identifiers can be found like this:

  ```
  awe> from astro.recipes.mods import Pipeline
  awe> for id in Pipeline.get_available_sequence_identifiers(): print id
  ```

  And the options can be printed in this way:

  ```
  awe> from astro.recipes.util.ArgumentParser import ArgumentParser
  awe> for opt in ArgumentParser().opts: print opt
  ```

  This gives you a list of tuples of the form (short_opt, long_opt, definition).

### 25.1.9 awe-prompt

- Q: What is namespace and how can I see it?

  A: Simply speaking, namespace is all modules, classes, attributes, and methods available in the current scope (i.e., level). If the module, class, attribute, or method you need is not visible in the current namespace, then it cannot be used. It is either loaded within the namespace at a different scope, or it is not there at all. The builtins `dir()` and `help()` allow you to explore the namespace.

  `dir()` without any arguments gives the namespace of the current scope. It typically will give Python builtins and any modules pre-loaded at startup or during the current session. Calling `dir()` with a module, class, attribute, or method as the argument will give you the namespace at that level or an arbitrary level:

```
awe> dir()
['__builtin__', '__builtins__', '__doc__', '__file__', '__name__', '__version_
_', 'astro', 'atexit', 'os', 'pydoc', 'readline', 'rlcompleter', 'sys', 'users
tartup']
awe> dir(os)
['EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST', 'EX_NOINP
UT', 'EX_NOPERM', 'EX_NOUSER', 'EX_OK', 'EX_OSERR', 'EX_OSFILE', 'EX_PROTOCOL'
.
.
.
ttyname', 'umask', 'uname', 'unlink', 'unsetenv', 'utime', 'wait', 'waitpid',
'walk', 'write']
awe> dir(os.write)
['__call__', '__class__', '__cmp__', '__delattr__', '__doc__', '__getattribute
__', '__hash__', '__init__', '__module__', '__name__', '__new__', '__reduce__'
, '__reduce_ex__', '__repr__', '__self__', '__setattr__', '__str__']
```

`help()` can be used in a similar way, but it gives *all* the docstrings recursively for the module, class, or method, or for the parent class of the attribute that was given as the argument.

- Q: When importing something from the `awe`-prompt, I get strange error messages through my screen. I know that the software is okay. What is the deal?

  A: Make sure that you are not importing something from within the awe directory structure. Due to some strange Python things concerning packages, this can lead to completely intractable error messages.

- Q: One of the Python recipes generates an error and I can not figure out what is wrong. What should I do?

  A: Sometimes problems arise because of old compiled Python files (*.pyc); there may be leftovers of Python files that no longer exist. Remove all these files and try again.

- Q: What are some usefull packages and modules available at the `awe`-prompt?

  A: See the awe-prompt HOW-TO.

### 25.1.10 Queries

- Q: What are queries and how do I use them?

  A: Queries in `AWE` are queries to the database from Python with the purpose of returning some needed information. The Queries HOW-TO containes a complete explaination of the use of queries in `AWE`.

- Q: What's the deal with the `select()` method?

  A: The `select()` method is simply a convenience method to find the most recent, valid version of a given frame. Use the Python builtin `help()` on a particular select method to find out exactly how it makes this choice.

### 25.1.11 Process Parameters

- Q: Which process parameters can be configured and where?

  A: Generally speaking, all process parameters can be configured from the `awe`-prompt prior to running a `make` or executing a task. The Configuration HOW-TO gives a complete description of the various methods for process parameter configuration.

### 25.1.12 Context

- Q:

  A:

- Q:

  A:

- Q:

  A:

- Q:

  A:

- Q:

  A:

## 25.2 Astro-WISE Environment

- Q: I want to run SExtractor on my images. How do I do that ?

  A: There are basically three ways to run SExtractor in the system. The most direct one would be to import 'Sextractor' from 'astro/external' and run 'Sextractor.sex(image)'. You can also set the SExtractor configuration through this interface, and modify the list of output parameters. A second way to run SExtractor would be using the 'sex' method of BaseFrame or children thereof. To be concrete, instantiate a ScienceFrame object and call the 'sex' method. The third, and in our paradigm the most correct, way to run SExtractor, is to invoke the 'make' method of a Catalog object. The frame you want to extract sources from is given as a dependency to the Catalog object. For the configuration of SExtractor and to modify its outputs, you need to provide the Catalog object with two additional dependencies: a SextractorConfig object, and a list of parameters. Note that this last way of creating a SExtractor catalog uses the first method. Also note, that the output of running SExtractor is in LDAC fits format.

- Q:

  A:

- Q:

  A:

- Q:

  A:

## 25.3 AW Tutorials

- Q:
  A:
- Q:
  A:
- Q:
  A:

## 25.4 Calibration

- Q:
  A:
- Q:
  A:
- Q:
  A:

## 25.5 Image Pipeline

- Q:
  A:
- Q:
  A:
- Q:
  A:

## 25.6 Visualization

- Q:
  A:
- Q:
  A:
- Q:
  A:

## 25.7 Development

- Q:
  A:
- Q:
  A:
- Q:
  A:

# Part III

# Appendix

# Appendix A

# Installing the basic Astro-WISE Environment

1. Checkout `awe` from CVS. The password required to log in can be obtained on request:

   ```
   > cvs -d :pserver:anoncvs@cvs.astro-wise.org:/cvsroot login
   > cvs -d :pserver:anoncvs@cvs.astro-wise.org:/cvsroot checkout -r AWBASE awe
   ```

2. Follow the instructions given in `awe/INSTALL`.

3. Compile the Python code. This is done automatically by step 4 below. It can be done directly by calling the following scripts:

   ```
   /path/to/awe /path/to/AWBASE/common/toolbox/compile_pipeline.py
   /path/to/awe /path/to/current/common/toolbox/compile_pipeline.py
   ```

4. Let the Python code be updated automatically from CVS.

   This can be done by using *crontab*. Example:

   Add two *crontab* jobs to do automatic updates from CVS for the "AWBASE" and "current" checkouts.

   A list of current crontab jobs can be obtained with this Linux command:

   ```
   > crontab -l
   ```

   Edit the contents of the crontab configuration file and install with:

   ```
   > crontab -e
   ```

   Add the following lines which update the checkouts every 10 minutes (adapt as necessary):

   ```
   5,15,25,35,45,55 * * * * /path/to/awe /path/to/AWBASE/common/toolbox/update_pipeline.py AWBASE
   5,15,25,35,45,55 * * * * /path/to/awe /path/to/current/common/toolbox/update_pipeline.py current
   ```

   The crontab file can be uninstalled by using:

   ```
   > crontab -r
   ```

# Appendix B

# Installation of database software

For the moment, no special instructions except that you should read the installation instructions for Oracle 10g.

# Appendix C

# Installation of various Astro-WISE Servers

This part describes how to setup and configure the Astro-WISE Servers.

## C.1 The Database Viewer

The database viewer accepts on the command line a host name and (after switch `-p`) a port number. At the moment it can only listen on one ethernet interface and on one port number. The default hostname is `localhost` and the default port number is `8879`. To start it from the command line one could type:

```
> awe $AWEPIPE/Services/dbview/dbview.py dbv.aoc.astro-wise.org -p 8879
```

You can stop the process by sending it the `INT` signal. It can be started automatically via the following script:

```
#!/bin/sh
AWEPIPE=/home/astro-wise/awehome/opipe/current
awe=/home/astro-wise/local/bin/awe
host=dbv.aoc.astro-wise.org
port=8879
#
case $1 in
start)
    p=${AWEPIPE}/Services/dbview/dbview.py
    if [ -f ${p} ] ; then
        ${awe} -u ${p} -p ${port} ${host} > ${HOME}/dbview.log 2>&1 &
    fi
    ;;
status)
    pid=‘ps -def | grep awe | awk ’{if{$1=="’dbview’"&&$3=="1")print $2}’‘
    if [ "X${pid}" = "X" ] ; then
        echo "Down"
    else
        echo "Up (pid = ${pid})"
```

```
    fi
    ;;
stop)
    pid=‘ps -def | grep awe | awk ’{if($1==”’dbview’”&&$3=="1")print $2}’‘
    if [ ! "X${pid}" = "X" ] ; then
        kill ${pid}
    fi
    ;;
*)
    echo "Usage: $0 (start|stop|status)"
    ;;
esac
```

See the last section in this chapter for a sample script to start the database viewer at system start.

## C.2 The Dataservers

The dataservers usually listen on port 8000 and can listen to any ethernet device. The example startup script below searches for all available ethernet devices and passes them on to the dataserver. The dataservers react on signal HUP (stop) and USR1 (rescan disk). Usually the fully qualified domainnames of the dataservers are put in the Astro-WISE DNS and point to all the local dataservers (like for example `ds.astro.rug.astro-wise.org`).
The following script will start a dataserver listening on all possible ethernet devices. It is assumed that the data server directory is in a subdirectory data of a directory which starts with the first three letters of the hostname, and the last two letters of the hostname (which is a sequence number in Groningen).

```
#!/bin/sh -f
AWEPIPE=/home/astro-wise/AWEHOME/current
awe=/home/astro-wise/AWEHOME/x86_64/local/bin/awe
mp=’kgb@astro-wise.org’
log=’ds.log’
pid=’ds.pid’
dir=‘hostname | awk -F. ’{printf("%s%s\n",substr($1,1,3),substr($1,length($1)-1,2))}’‘
port=8000
if [ ! -d ${dir}/data ] ; then
    echo "Directory ${dir}/data missing"
    exit 1
fi
args=‘/sbin/ifconfig | awk ’
BEGIN{e="";}
{
    if (substr($0,1,3)=="eth") {
        e=$1
    } else {
        if (e!=""&&$1=="inet"&&substr($2,1,5)=="addr:") {
            printf("%s:’${port}’\n",substr($2,6,length($2)-5));
            e=""
```

```
        }
    }
}
'‘
cd ${dir}/data
case "$1" in
    start)
        if [ -f ${log} ] ; then
            if [ -f ${log}.save ] ; then
                cat ${log} >> ${log}.save
                rm -f ${log}
            else
                mv ${log} ${log}.save
            fi
        fi
        ${awe} -u ${AWEPIPE}/common/net/dataserver_server.py ${args} \
            -p ${pid} -m ${mp} > ${log} 2>&1 &
        sleep 5
        while [ ! -e ${pid} ] ; do
            sleep 1
        done
        ;;
    stop)
        if [ -e ${pid} ] ; then
            kill -HUP ‘cat ${pid}‘
            while [ -e ${pid} ] ; do
                sleep 1
            done
        fi
        ;;
    reload)
        if [ -e ${pid} ] ; then
            kill -USR1 ‘cat ${pid}‘
        fi
        ;;
    status)
        ${awe} -u ${AWEPIPE}/common/net/dataserver_server.py ${args} -status
        ;;
    *)
        echo "Usage: $0 (start|stop|status|reload)"
        ;;
esac
exit 0
```

## C.3 The Distributed Processing Server

The distributed processing server can run on an openpbs managed cluster or on a unmanaged cluster. A script to start the DPU server could look like this:

```
#!/bin/sh
#
```

```
# dpu-server    This shell script starts and stops the dpu server
#
# dpu-server:   345 98 90
#
# description:  Astro-WISE dpu server.
#
# probe:        true

port=9000
host=hpcibm1.service.rug.nl
tunnel=astro-wise@omegadlt.astro.rug.nl
infofile=.`basename $0`.info
logfile=`basename $0`.log

AWEHOME=${HOME}/awehome
AWEPIPE=${AWEHOME}/opipe/current
cd ${HOME}/dpu
export AWEHOME
export AWEPIPE

case "$1" in
    start)
        if [ "X${tunnel}" != "X" -a \
            -z "`ps -def | grep ${port}:${host}:${port} | grep ssh`" ] ; then
            ssh ${tunnel} -fN -R ${port}:${host}:${port}
        fi
        if [ ! -e ${infofile} ] ; then
            awe -u ${AWEPIPE}/common/net/dpu_server.py hpcibm1.cluster:${port} \
                ${host}:${port} -pbs short,200 -ppn 2 > ${logfile} 2>&1 &
        fi
        ;;
    stop)
        if [ -e ${infofile} ] ; then
            pid=`awk '{if (NR == 3){print $0}}' ${infofile}`
            kill -HUP ${pid}
            while [ -e ${infofile} ] ; do
                sleep 1
            done
        fi
        ;;
    status)
        if [ -e ${infofile} ] ; then
            cat ${infofile}
        else
            echo Down
        fi
        ;;
    restart)
        if [ -e ${infofile} ] ; then
            pid=`awk '{if (NR == 3){print $0}}' ${infofile}`
            kill -INT ${pid}
```

```
            while [ -e ${infofile} ] ; do
                sleep 1
            done
            awe -u ${AWEPIPE}/common/net/dpu_server.py hpcibm1.cluster:${port} \
                ${host}:${port} -pbs short,200 -ppn 2 > ${logfile} 2>&1 &
        fi
        ;;
    kill)
        if [ -e ${infofile} ] ; then
            pid=‘awk ’{if (NR == 3){print $0}}’ ${infofile}‘
            kill -KILL ${pid}
            while [ -e ${infofile} ] ; do
                sleep 1
            done
        fi
        ;;
    *)
        echo $"Usage: $0 (start|status|stop)"
        ;;
esac
exit 0
```

This DPU server is running on a 200 node 2 processors per node cluster. When the DPU server starts up it generates a unique key based on the hostname and current directory. This key is needed for the clients to be able to get a secure connection to the DPU server. With the script `dbdpu.py` in the Toolbox directory an entry in the `DPU_SERVER_TABLE` can be made.
At the moment the DPU server cannot run on an unmanaged cluster, work is still in progress.

## C.4 Sample startup script

For some services it will be necessary that they are started at system startup and stopped at system shutdown. Below is a sample script which could for example be used on a RedHat Linux distribution to startup an Astro-WISE service.

```
#!/bin/sh
#
# purpose:        This shell script takes care of starting and stopping
#                 an Astro-WISE service. It is assumed that there
#                 is a script available somewhere in the searchpath of the
#                 specified user account with same name as this script
#                 which is capable of stopping and starting the service.
#
# chkconfig:      345 98 90
# description:    Astro-WISE server starter/stopper
#
# probe:          true
#
usr=astro-wise
exe=‘basename $0‘
case "$1" in
start)
```

```
    su -l ${usr} -c "${exe} start"
    sleep 2
    ;;
stop)
    su -l ${usr} -c "${exe} stop"
    sleep 2
    ;;
status)
    su -l ${usr} -c "${exe} status"
    sleep 2
    ;;
*)
    echo "Usage: $0 (start|status|stop)"
    ;;
esac
exit 0
```

Note that this is a secure script, it is NOT starting the service as root!

# Appendix D

# Adding a node to the Astro-WISE federation

Adding a complete node to the Astro-WISE federation requires the installation of the Astro-WISE software, a database, one or more dataservers. The database and dataservers then have to be connected to the other nodes in the federation. Here we describe both the local requirements (such as hardware, database layout, diskspace), the requirements for the connections between the nodes (such as firewall settings, bandwidth) and the installation and configuration of the components.

## D.1 Firewall setup

Nodes that participate in a federation have to be accessed by the other nodes. This means that for the database and dataservers some network ports need to be opened for all participating nodes.
For the database host, port `1521/tcp` needs to be open to

```
db.astro.rug.astro-wise.org
db.astro.uni-bonn.astro-wise.org
db.na.astro.astro-wise.org
db.ocam.mpe.astro-wise.org
...
```

For the dataserver host, port `8000/tcp` needs to be open to

```
ds.astro.rug.astro-wise.org
ds.astro.uni-bonn.astro-wise.org
ds.na.astro.astro-wise.org
ds.ocam.mpe.astro-wise.org
...
```

If a cluster of dataservers is available, it is sufficient to open the necessary port for only one of these.

## D.2 Database creation and configuration

It is assumed that the most recent version of Oracle has been installed and that no database has yet been created. The database has to be created with the following settings.

- The default blocksize has to be `32k`
- ASM—Automatic Storage Management—has to be used for the device. This gives maximum flexibility, performance and relialibility.
  - Initially, one ASM diskgroup should be sufficient
  - The first diskgroup should be called `AWDISKGROUP1`
  - New harddisks can be added to an ASM diskgroup in a working database. Existing harddisks can be taken off-line in a working database. In both cases the data will be rebalanced automatically.
- The following tablespaces should be defined.
  - `AWLISTS`, add eight datafiles, each having autoextend with 100MB to maxsize. This allows for 1056GB of data, before additional datafiles have to be added.
  - `AWINDX`, add four datafiles, each having autoextend with 100MB to maxsize. This allows for 528GB of data, before additional datafiles have to be added.
  - `UNDOTBS1` UNDO tablespace ¡= 4GB autoextend=off
  - `TEMP` temporary tablespace ¡= 2GB autoextend=on maxsize=2GB
  - `USERS`, make sure that the datafile has autoextend with 100MB to maxsize.

  `USERS` has to be the default tablespace and `TEMP` has to be the default temporary tablespace. For certain operations, the `TEMP` and `UNDOTBS1` may be too small. For `TEMP` you can increase the maxsize. For the UNDO tablespace you should create an `UNDOTBS2` tablespace which can grow indefinitely. After that you should make `UNDOTBS2` the default UNDO tablespace, perform your operation, make `UNDOTBS1` the default tablespace and drop `UNDOTBS2`. The reason to do that is that otherwise the UNDO tablespace can grow very big and allow runaway transactions to run for hours before failing. 4GB corresponds to a transaction that takes about an hour, which should be sufficient for operations on tables of sizes up to 300GB.
- Archive logging has to be set to on. This allows for online backups to be made. A backup of a 300GB database typically takes several hours, during which the database would not be available if it had to be backed up off-line.
- Set the service name of the database equal to its fqdn.

  ```
  ALTER SYSTEM SET SERVICE_NAMES = 'db.?.astro-wise.org' SCOPE=BOTH;
  ```

  Replace the question mark appropriately for your domain. Using the fqdn of the database host as the database service name will make it possible to connect to the database without any client configuration. No `tnsnames.ora` needs to be present in that case on the client.
- Enable the usage of global_names.

  ```
  ALTER SYSTEM SET GLOBAL_NAMES = TRUE SCOPE=BOTH; ALTER DATABASE RENAME
                  GLOBAL_NAME TO DB.?.ASTROWISE.ORG;
  ```

  If the database name contains dashes they have to be removed when renaming the `GLOBAL_NAME` of the database.
- Make sure that your listener runs in shared mode and execute

```
ALTER SYSTEM SET DISPATCHERS = '(PROTOCOL=TCP) (DISPATCHERS=7)';
```

If each client would make a so-called "dedicated" connection, the database host would be saturated quickly when parallel processing takes place. This starts to become noticeable when you have only a few hundred MB left on your database host and you have one hundred or more parallel connections. With `lsnrctl services` you can discover how and how often there has been a connection to your database. Ideally, the number of established `DEDICATED REMOTE SERVER` connections should be zero.

- You have to explicitly enable checking of resource limits. You do this with

```
ALTER SYSTEM SET RESOURCE_LIMIT = TRUE;
```

The database default is that resource limits that are set in database profiles will not be enforced.

- There should be ten redo logs of 2GB, where each redo log has only one member. For typical workloads this is more than enough, but can be required for some long running maintenance transactions.

## D.3 Streams configuration

Oracle Streams is set up at a node by taking the following steps.

1. Change the `STRMADMIN` password and datafile for the `LOGMNRTS` tablespace in

   `common/toolbox/dbmakestrmadmin.sql`

   and run the script as `SYS` in `sqlplus`.

2. Make database links to other nodes as `STRMADMIN`, given the following name for the database link and the fqdn of the node.

   ```
   <linkname> = DB.ASTRO.RUG.ASTROWISE.ORG
   <nodename> = DB.ASTRO.RUG.ASTRO-WISE.ORG
   ```

   Note that a dash is not allowed in the linkname. Connect as `STRMADMIN` before making the link with the following command

   ```
   CREATE DATABASE LINK <linkname>
   CONNECT TO STRMADMIN
   IDENTIFIED BY <strmadmin password>
   USING '<nodename>';
   ```

   and repeat this for all remote nodes that one wants to connect to.

3. Create a datapump directory and grant access to it by the STRMADMIN user.

   ```
   CREATE OR REPLACE DIRECTORY AWDATAPUMP AS '/your/awdatapump/directory';
   GRANT READ, WRITE ON DIRECTORY AWDATAPUMP TO STRMADMIN;
   ```

## D.4 Maintenance

### D.4.1 Cleaning up deleted files and database objects

The `common/toolbox/dbvacuum.py` script is used to put aside files on the dataservers and delete corresponding objects from the database, which users have deleted. The files will be moved to the `ddata` directory on the dataservers from which they can be removed for all eternity.

### D.4.2 Archivelog backup

The archivelog should be backed up daily using a cron job similar to

```
TAG=‘date +BACKUP_AW01.ARCH_%y%m%d%H%M%S‘

$ORACLE_HOME/bin/rman target / log=$HOME/Logs/$TAG.log <<EOF
backup device type disk tag ’$TAG’ archivelog all not backed up delete all input;
EOF
```