

Astro-**W**ise **E**nvironment

User and Development manual

March 25, 2004

Contents

I	User's manual	4
1	Data reduction cookbook	5
1.1	Everything in a nutshell	5
1.1.1	Architecture	5
1.1.2	Python and Object Oriented Programming	5
1.1.3	Processing steps	6
1.2	Ingesting raw data into the database	6
1.3	Database GUI	6
1.4	Data processing	7
1.4.1	Interactive processing	7
1.4.2	Non-parallel processing	8
1.4.3	Parallel processing	9
1.4.4	The bias pipeline	11
1.4.5	The flat-field pipeline	11
1.4.6	The photometric pipeline	11
1.4.7	The image pipeline	12
1.5	Timestamps	13
1.6	Interfaces to other programs	13
1.6.1	SQL interface, interaction with the database	13
1.6.2	Eclipse interface	15
1.6.3	SWARP interface	16
1.6.4	SExtractor interface	16
1.6.5	LDAC interface	16
1.7	A lengthy example	17
1.7.1	Ingesting (skip in case of demo)	17
1.7.2	Image calibration files	17
1.7.3	Photometric calibration files	19
1.7.4	Image pipeline	21
1.7.5	Coaddition	21
1.7.6	Source lists	22
2	Source lists in the Astro-Wise system	23
2.1	Creating simple source lists from science frames	23
2.2	External source lists	24
2.3	Using sourcelists	24
2.4	Associating source lists	25

3	Documentation, manual pages	29
3.1	Online documentation	29
3.2	Inline documentation	29
4	Frequently Asked Questions	34
II	Developer's manual	35
5	Development	36
6	Key concepts	37
6.1	Persistent classes	37
6.1.1	targets, dependencies, make	37
6.2	Verification and quality control	39
7	The astro-wise class hierarchy	41
8	Database Tasks	43
8.1	Setting up the database for general use	43
8.2	Keeping the database synchronized with STABLE sources	44
8.3	Database Type Evolution	45
8.3.1	Database Type Evolution	45
8.3.2	Overview	45
8.3.3	The SQL representation of persistent Python class	46
8.3.4	Finding information about the SQL types, tables and views	46
8.3.5	Adding a persistent class	46
8.3.6	Removing a persistent class	46
8.3.7	Adding persistent attributes to a class	47
8.3.8	Removing persistent attributes from a class	47
8.3.9	Renaming a persistent attribute	47
8.3.10	Changing the type of a persistent attribute	47
8.3.11	Moving a persistent subclass to a different parent class	48
8.3.12	Error messages	48
A	Where to find things	49
A.1	Where to find things	49
B	Quick guide to making a test schema in the database	50
B.1	Summary	50
B.2	Checkout of CVS code base	50
B.3	Creating a data server	51
B.4	Changing relevant environment variables	51
B.5	Bootstrap the test schema	51
B.6	Ingesting data into the database	52

Introduction

This document describes the Astro-Wise data reduction environment. The first part of this document is aimed at the end-user, and shows how to use the environment to process data, and introduces the use of source lists and their associations. This part consists of a reduction cookbook, a short introduction to the creation of source lists, a description of how to browse the large amount of documentation written inside the code, and a list of Frequently Asked Questions. The second part of this document is aimed at the developer, and gives a high-level overview of the concepts and implementation of the Astro-Wise library.

Part I

User's manual

Chapter 1

Data reduction cookbook

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 working system, and has access to the database, a parallel computing cluster and his/her own CVS checkout of the code.

1.1 Everything in a nutshell

1.1.1 Architecture

In the architecture of the Astro-Wise system, three main components can be identified : a file server, a database, and a Beowulf cluster.

The file server holds the actual FITS-files, while the database keeps track of the whereabouts of 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 combination of the file server and the database is referred to as the *dataserver*. The Beowulf cluster is used to process the data. During processing, requests are made to the *dataserver* for the raw science data and for the necessary calibration files.

1.1.2 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. Because of the OOP style in which the pipelines are written, *classes* are associated with the various conventional calibration images. For example, 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 *attributes* and *methods*. 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 the system various recipes have been coded that control the creation of instances of these classes. There may be different ways to create an instance (object) of a class, depending on which attributes are set to what values, and which methods are used. A ColdPixelMap object for example, can be created by using a DomeFlatFrame or a TwilightFlatFrame object as input.

Within the Object Oriented Programming style *inheritance* is an important concept. Classes can inherit attributes and methods from other classes. In the above example, the ColdPixelMap

class has an attribute 'flat', which must be an object of the BaseFlatFrame class; DomeFlatFrames and TwilightFlatFrames (these are master dome- and twilight images) inherit from the BaseFlatFrame class and hence are both allowed as input.

Once you have obtained the code via CVS, an environment variable called **\$PYTHONPATH** should point to the installation directory (opipe).

1.1.3 Processing steps

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

- Ingesting raw data into database/dataserver
- 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 sections 1.4.4, 1.4.5, 1.4.6 and 1.4.7 for more specific information about the pipelines.

1.2 Ingesting raw data into the database

The first step in using the Astro-Wise system is the ingestion of the raw data into the database. This is handled by a *recipe* called `ingest.py`, which can be found in `$PYTHONPATH/Toolbox/`. The *recipe* is invoked from the Unix command line with the following command :

```
awe $PYTHONPATH/Toolbox/ingest.py -i <raw data> -t <type>
-n <number of extensions>
```

with `<raw data>` the list of input data to ingest, and `<type>` the type of the input data (e.g. bias, domeflat). The `-n` switch is optional, and is used to set the number of extensions of the input raw data (default is 8). The input data must be unsplit, and can (for the moment) be of one of the following types : bias, dome, twilight, science.

1.3 Database GUI

The Astro-wise database is a product of Oracle. The package comes with a (somewhat archaic) GUI to manage, view and edit the database and its entries. This is one of the entry points to look at data, because in particular it can give a complete overview of everything that's present in the database. Open the GUI with the following commands:

```
unixprompt>source ~/aworacle/bin/coraenv (followed by enter)
unixprompt>oemapp console &
```

In order to view ingested data, one can now: "Launch standalone", "Add selected databases", then for example: **Databases** -> **AW98.ZERNIKE.NOVA.AW** (Log in...) -> **Schema** -> **AWOPER** -> **Tables**

Alternatively you can open the **Views** in the this schema. You will be presented with an overview of all classes, the names of which should be fairly self-explanatory. Right-click on a table or view in the left window and select **View/Edit Contents...** to open the **table**, or **Display Contents...** to open the **view**. You may be asked to refine the query if there are many entries. For now it does not appear absolutely necessary to do this, but it may still be helpful to narrow the selection from the start. Note that the red/green dot in the bottom status bar indicates whether all entries have been loaded.

If you select a table the right window will show its dependencies. The dependencies of type reference (REF) will not show in the tables, but may have been made visible in the views.

Note that when viewing sources (table SourceList*sources) the GUI may crash due to the high number of entries.

1.4 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.

1.4.1 Interactive processing

An example of interactive processing:

```
awe> from astro.main.BiasFrame import BiasFrame
awe> from astro.main.RawFrame import RawBiasFrame
awe> r = RawBiasFrame.select_for_date(date='2001-01-02', chip_name='ccd50')
awe> for raw in r: raw.retrieve()
...
awe> b = BiasFrame(pathname='bias.fits')
awe> b.raw_bias_frames = r
awe> b.process_params.OVERSCAN_CORRECTION = 1
awe> b.make()
awe> b.store()
awe> b.commit()
```

This will select raw bias frames from the database for the night of January 2nd, 2001, *retrieve* the FITS files from the dataserer, create a master bias image called "bias.fits", then upload the image to the dataserer (*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.

1.4.2 Non-parallel processing

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

```
awe> from DBRecipes.DomeFlat import DomeFlatTask
awe> task = DomeFlatTask(date='2000-04-28', chip_name='ccd50',
filter_name='#842')
awe> task.execute()
```

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

```
awe> help(DomeFlatTask)
Help on class DomeFlatTask in module DBRecipes.DomeFlat:
```

```
class DomeFlatTask
| -----
|
| CATEGORY : data processing, calibration pipeline.
|
| PURPOSE  : making a master dome flat for a specific date, chip
|            and filter.
| -----
|
| This task is used to create a master dome flat.
|
| The task assumes that a dataserver is active, and that it has
| been instantiated with valid values for the 'date', 'chip_name',
| 'filter_name' and (optionally) 'overscan' parameters. The task
| is executed by calling its 'execute' method.
| -----
|
| Example of use from the awe prompt :
|
|     awe> task = DomeFlatTask(date = '2003-02-12', chip_name = 'ccd50',
|     ...                   filter_name = '#843', overscan = 1)
|     awe> task.execute()
|
| Mandatory keywords :
|
|     date (string, yyyy-mm-dd), chip_name (string), filter_name (string)
|
| Optional keyword :
|
|     overscan (integer)
|
| The type of the values assigned to the keywords is given between
| parentheses.
```

```

| -----
|
| Methods defined here:
|
| __init__(self, date='1989-01-01', chip_name='aChip', filter_name='aFilter',
overscan=0)
|
|   configure_process_parameters(self, masterdomeflat)
|
|   construct_target_name(self)
|
|   execute(self)
|       Method intended to allow parallel execution
|
|   make_domeflat(self)
|
|   query_database(self)
|
| -----
| Data and non-method functions defined here:
|
| __doc__ = '\n      -----...-----'
|
| __module__ = 'DBRecipes.DomeFlatTask'
(END)

```

These tasks and their help files are comparable in their usage. The names of these classes are:

- BiasTask
- HotPixelsTask
- DomeFlatTask
- ColdPixelsTask
- TwilightFlatTask
- MasterFlatTask
- FringeFlatTask
- NightSkyFlatTask
- ScienceStareTask
- ScienceDitherTask
- CoaddTask

1.4.3 Parallel processing

In order to reduce large volumes of data, it is possible to use parallel processing. The **\$PYTHON-PATH/parallel/generic_runner.py** recipe is used for this. This recipe in fact calls part of the same recipes that are used for non-parallel processing; the parallel code is a layer on top of the existing recipes.

Table 1.1: The list of tasks available for running the calibration and image pipelines on the cluster.

task	functionality	task	functionality
<code>Bias</code>	derive masterbias	<code>Nightsky_Flat</code>	derive night-sky flat
<code>Hot_Pixels</code>	derive hot-pixel map	<code>Fringe_Flat</code>	derive fringemap
<code>Cold_Pixels</code>	derive cold-pixel map	<code>Master_Flat</code>	derive master flat
<code>Dome_Flat</code>	derive domeflat	<code>Science_Stare</code>	calibrated data ¹
<code>Twilight_Flat</code>	derive twilightflat		

1 : used for calibrating single-pointing science data.

```
awe $PYTHONPATH/parallel/generic_runner.py -task <task> [-log <logfile>]
    -i <instrument> -d <date (yyyy-mm-dd)> [-oc <overscan correction>]
```

See table 1.1 for an overview of the possible values of the "-task" option of generic_runner.py.

So, in order to produce the same master bias as in the previous sections, type:

```
awe $PYTHONPATH/parallel/generic_runner.py -task Bias -log biases.log
    -i WFI -d 2001-04-02 -oc 1
```

Note that (naturally) in this case the master biases for all CCDs in the WFI instrument are created simultaneously.

Parallel processing in Groningen

Log in onto your account at the parallel computing center using ssh. A queuing system is used. Jobs are submitted to the queue using a Python script; **submit-awe.py**, along with a slightly changed Python startup script **awe**, located in the Toolbox directory of your checkout. Copy these files to the directory in your account at the HPC, where you will run the recipes. In order to create master biases like above, give the command:

```
awe submit-awe.py $PYTHONPATH/parallel/generic_runner.py -task Bias
    -log demo-biases.log -i WFI -d 2000-04-28
```

It is possible to check the status of a job (queued or running) with the following Unix command:

```
unixprompt>qstat
```

or

```
unixprompt>qstat | grep omegacam
```

Jobs can be removed from the queue:

```
unixprompt>qdel <job number>
```

1.4.4 The bias pipeline

Recipes used:

- Bias.py
- HotPixels.py

1.4.5 The flat-field pipeline

Recipes used:

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

1.4.6 The photometric pipeline

Recipes used:

- ZeropointSky.py

This recipe, like all other recipes, uses the database for retrieving the information needed for processing. However, contrary to the recipes used by the bias and flatfield pipelines, more data is needed than just frames for the recipe to work. The recipe also needs a standard star catalog, and a standard extinction curve. Before processing can begin, both these items have to be stored in the database first.

The standard extinction curve can be stored in the database as follows:

```
awe> from astro.main.PhotExtinctionCurve import PhotExtinctionCurve
awe> extcurve = PhotExtinctionCurve(pathname = 'ca1564E.dat')
awe> extcurve.commit()
```

The steps to perform for storing the standard star catalog in the database are:

```
awe> from astro.main.PhotRefCatalog import PhotRefCatalog
awe> refcat = PhotRefCatalog(pathname = 'ca1569E.cat')
awe> refcat.store()
awe> refcat.commit()
```

Note the extra `store` command that will put the file on the fileserver.

These actions (ideally) have to be performed only once during the setup of the system. The files that contain the standard extinction curve (`ca1564E.dat`) and the primary standard star catalog (`ca1569E.cat`) can be found in a separate CVS checkout: `/cvsroot/catalog`. The standard extinction curve used is the one for La Palma. The standard star catalog contains all the Landolt equatorial fields and cluster fields. It supports the $UBVR_cI_c$ system, as well as the first four Sloan bands (u' , g' , r' , i').

Photometric calibration for the poor

In contrast to the bias or flatfield pipeline, the photometric pipeline needs a very specific set of data that greatly depends on the calibration plan. It is, therefore, difficult to create simple, generic modules for running the photometric pipeline. This basically forces users to write their own specific recipes to fulfill the requirements of the system.

Given this state of affairs, how does one create calfiles for the photometric calibration in the case that one does not want to write their own photometric recipes? The photometric calfiles can easily be created using the `ingest_photometrics` tool. This tool is located in the `$PYTHONPATH/Toolbox/photometry` directory and is invoked thus:

```
awe $PYTHONPATH/Toolbox/photometry/ingest_photometrics -z <value of
    zeropoint> -c <chip> -f <filter name> -ec cal564E.dat
```

with `-c` the chip name, `-f` the filter name, `-z` the zeropoint for the given combination of filter and chip, and `-ec` an extinction curve. Note that for this tool to work the filters and chips must be known to the database. Also note that the timestamps of the produced calfiles have to be adjusted in the database to suit your own set of raw science data. A file for the standard extinction curve can be found under a separate CVS checkout: `/cvsroot/catalog`.

Important note: in the initialization of the system for image pipeline processing, a set of default photometric calibration files needs to be created in the database for every combination of chip and filter. The same `ingest_photometrics` tool can be used for this initialization.

1.4.7 The image pipeline

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

Recipes used:

- ScienceStare.py
- ScienceDither.py
- ScienceJitter.py

Various modes of observation are supported by OmegaCAM, including *dithering* modes and a *stare* mode. The processing by the image pipeline varies slightly for the different modes. Here, only the reduction of data obtained with the simplest observing mode (*stare*) is described. Starting the image pipeline in single-CCD mode is quite easy. To reduce the data of a given raw science frame:

```
awe> from DBRecipes.ScienceStare import ScienceStareTask
awe> s = ScienceStareTask(raw_filename = <input name>)
awe> s.execute()
```

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

Or:

```
awe> from DBRecipes.ScienceStare import ScienceStareTask
awe> s = ScienceStareTask(date=<date>, object_name=<object name>)
awe> s.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 `Science_Stare`. The command issued to run the pipeline is:

```
awe $PYTHONPATH/parallel/generic_runner.py -i <instrument name>
      -task Science_Stare -d <date> -f <filter name> -o <object name>
```

Raw images that are ingested in 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 described in Sect. 1.4.6.

1.5 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 GUI, see section 1.3.

1.6 Interfaces to other programs

1.6.1 SQL interface, interaction with the database

In order to query data in the database or `commit` data to it, an interface to SQL was written. Through this interface logical operators are interpreted as SQL queries. A "like" method for strings is also available to query using wild cards similar to Unix command line arguments. In this way, the database can be searched from Python for objects with matching attributes etc. For example:

```
awe> from astro.main.MasterFlatFrame import MasterFlatFrame
awe> query = MasterFlatFrame.chip.name == 'ccd50'
```

The variable "query" now contains the result of the query for MasterFlatFrames for which the name of the CCD chip equals 'ccd50'. The result is in the form of a list, which contains the references to the MasterFlatFrame objects.

It is possible to extend queries with other queries, so that in case you want to obtain all MasterFlatFrames for CCD 'ccd50' and filter '#845' you could continue the above statements with:

```
awe> query &= MasterFlatFrame.filter.name == '#845'
```

Alternatively this could have been done in a one-liner:

```
awe> q = (MasterFlatFrame.chip.name == 'ccd50') &  
        (MasterFlatFrame.filter.name == '#845')
```

Note that in this case brackets are necessary. Operators like !=, <=, >=, <, > are also possible. Clearly in the latter cases the queried attributes need to be numerical values.

An example of the "like" method:

```
awe> from astro.main.RawFrame import RawScienceFrame  
awe> s = RawScienceFrame.OBJECT.like('CDF?*')
```

The above examples can be extended with statements such as the following:

```
awe> len(s)  
632  
awe> len(q)  
24  
awe> flat = q[0]  
awe> flat.domeflat.raw_domeflat_frames[0].DATE_OBS  
<DateTime object for '2001-02-13 21:24:08.00' at 84b3d00>  
awe> flat.domeflat.raw_domeflat_frames[0].filename  
'WFI.2001-02-13T21:24:08.199_1.fits'  
awe> for f in q: print f.domeflat.raw_domeflat_frames[0].EXPTIME  
...  
17.773  
17.773  
17.773  
2.991  
2.991  
2.991  
2.991  
2.991  
17.773  
17.773  
2.991  
2.991  
17.773  
17.773  
0.2011  
17.773  
0.2011  
0.2011  
3.7007  
5.7475  
3.762  
3.762  
3.7007  
5.7475  
2.991
```

```
awe> flat.retrieve()
Pinging xxx.xxx.x.xxx port
awe>
```

Note that the last statement retrieves data from the fileserver and saves it to your current working directory.

1.6.2 Eclipse interface

For image arithmetic the C library Eclipse is used. In order to use this library in the Astro-Wise system a Python wrapper/interface was written. There are three main classes 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.

Regions can be cut from images:

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

Also, 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 (\sim) 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()
awe> 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 chapter 3.)

1.6.3 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 SwarpConfig
awe> swarpconfig = SwarpConfig()
awe> swarpconfig.COMBINE = 'N'
awe> swarpconfig.REGRID = 'Y'
awe> files = ['file1.fits', 'file2.fits', 'file3.fits']
awe> Swarp.swarp(files, swarpconfig.get_kw_dict())
```

1.6.4 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> sci = 'sci_1.fits'
awe> Sextractor.sex(sci, param=['FLUX_ISOCOR'], CATALOG_NAME='mycatalog.cat', DETECT_THRESH=2.0)
```

In general the first argument of Sextractor.sex is the detection (and measurement) image, the second is an optional measurement image, the third are possible *extra* output parameters other than those specified in the interface. The final arguments are the configuration options, which are separate arguments in KEYWORD1='value1', KEYWORD2='value2', etc. format.

It is encouraged to use the higher level interface provided in the *Catalog* class:

```
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.scats", which can be overlaid on the FITS image ("sci_1.fits") when using ESO's Skycat viewer.

1.6.5 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. Again, this can be done from the Python prompt.

```
awe> from astro.external import LDAC
awe> incat = 'science.cat'
awe> outcat = 'science.filtered.cat'
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.

1.7 A lengthy example

Note: this example is intended to be run multiple times as a demo of the Astro-wise system. The ingestion step is included for completeness, but is not to be performed when showing the demo.

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.

1.7.1 Ingesting (skip in case of demo)

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 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 dataserer, 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:

```
wfi60835.fits
wfi60836.fits
wfi60867.fits
wfi60868.fits
wfi60869.fits
wfi60870.fits
wfi60871.fits
```

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

```
unixprompt>foreach i ( 'grep .fits bias_files.txt' )
foreach? awe $PYTHONPATH/Toolbox/ingest.py -i $i -t bias -n 8
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 dataserer as RawBiasFrame, RawDomeFlatFrame and RawTwilightFlatFrame instances.

One can check this by using Oracle's database GUI (Section 1.3, look in AWOPER -> Views -> AWRawBias (Display all...)). It is possible to sort/filter etc. (right click on column names). Notice that all data is split; each RawFrame is of a single CCD.

1.7.2 Image calibration files

Note: a special database user has been defined for the purpose of demos, it is required to become this user when running this example. This is done by editing the Environment.cfg file in the

.awe directory. *Database_user* and *database_password* should both be set to **awdemo**.

Note: at the university of Groningen a central computing cluster is used. This cluster uses a queuing system; differences w.r.t. the standard commands used for processing are stated in the box at the end of this section.

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 $PYTHONPATH/parallel/generic_runner.py -task Read_Noise
      -log demo-readnoise.log -i WFI -d 2000-04-28
```

Now use the following command to create master biases for each CCD:

```
awe $PYTHONPATH/parallel/generic_runner.py -task Bias -log demo-biases.log
      -i WFI -d 2000-04-28
```

Once the cluster is processing, a cluster log file is created, assume we named this file "demo-biases.log". All messages (including errors) that are produced on the nodes where the parallel tasks run are written to this file. A command such as the following can be used to keep track of the process while it is running:

```
unixprompt>tail -f demo-biases.log
```

At this point GainLinearity objects are necessary for further processing. These can be made from the Python prompt:

```
awe> from astro.main.GainLinearity import GainLinearity
awe> from astro.main.Chip import Chip
awe> from astro.main.Instrument import Instrument
awe> chip = ['ccd50', 'ccd51', 'ccd52', 'ccd53', 'ccd54', 'ccd55', 'ccd56', 'ccd57']
awe> instrument = 'WFI'
awe> instrument = (Instrument.instrument_name == instrument)[0]
awe> for chip in chip:
...   chip = (Chip.name == chip)[0]
...   gn = GainLinearity()
...   gn.chip = chip
...   gn.instrument = instrument
...   gn.gain = 2.0
...   gn.commit()
```

In addition, the logs for the individual tasks are written into the directory *logs*. Once the biases finish the other necessary calibration steps need to be performed (in this order) as follows:

```
awe $PYTHONPATH/parallel/generic_runner.py -task Hot_Pixels -log demo-hot.log
      -i WFI -d 2000-04-28
```

```
awe $PYTHONPATH/parallel/generic_runner.py -task Dome_Flat -log demo-dome.log
      -i WFI -d 2000-04-28 -f #842
```

```
awe $PYTHONPATH/parallel/generic_runner.py -task Cold_Pixels -log demo-cold.log
      -i WFI -d 2000-04-28 -f #842

awe $PYTHONPATH/parallel/generic_runner.py -task Twilight_Flat
      -log demo-twilight.log -i WFI -d 2000-04-28 -f #842

awe $PYTHONPATH/parallel/generic_runner.py -task Master_Flat
      -log demo-masterflat.log -i WFI -d 2000-04-28 -f #842

[awe $PYTHONPATH/parallel/generic_runner.py -task Fringe_Flat
      -log demo-fringe.log -i WFI -d 2000-04-28 -f #845]
```

I have added the optional step of making a fringe map, but this does not apply in the case of the Johnson B filter taken for this example.

Parallel processing in Groningen

Log in onto your account at the parallel computing center using ssh. A queuing system is used. Jobs are submitted to the queue using a Python script; **submit-awe.py**, along with a slightly changed Python startup script **awe**, located in the Toolbox directory of your checkout. Copy these files to the directory in your account at the HPC, where you will run the recipes. In order to create master biases like above, give the command:

```
awe submit-awe.py $PYTHONPATH/parallel/generic_runner.py -task Bias
      -log demo-biases.log -i WFI -d 2000-04-28
```

It is possible to check the status of a job (queued or running) with the following Unix command:

```
unixprompt>qstat
```

or

```
unixprompt>qstat | grep omegacam
```

Jobs can be removed from the queue:

```
unixprompt>qdel <job number>
```

1.7.3 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.

Standard values (relative photometry)

For relative photometric calibration in the case of WFI observations, standard values for zero-point and extinction are available. The required calibration files to use these are either already present in the database, or can be created as follows (See sect. 1.4.6). Note that you need a file containing data of a standard extinction curve, **cal564E.dat**, which can be obtained by doing the following CVS checkout:

```
cvs -d cvs.astro-wise.org:/cvsroot co catalog
```

You then need to copy cal564E.dat to the directory where you run the ingest_photometrics script.

```
awe $PYTHONPATH/Toolbox/photometry/ingest_photometrics -c ccd50 -f #842 -ec cal564E.dat
awe $PYTHONPATH/Toolbox/photometry/ingest_photometrics -c ccd51 -f #842 -ec cal564E.dat
etc.
```

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 1.7.4.

Using standard star fields (absolute photometry)

Alternatively (and in the case of this demo) standard star fields may be used to obtain an exact photometric solution. For the CDF as used in this demo standard star fields are available. The image data for these fields has to be reduced itself before zeropoints for the night can be derived. This is done by running the image pipeline on the standard star fields. This is identical to section 1.7.4, but using the following command (Science_Stare does not resample the data, and the object name is used to select a standard star field):

```
awe $PYTHONPATH/parallel/generic_runner.py -task Science_Stare
      -log demo-photom.log -i WFI -d 2000-04-28 -f #842 -o 'Land_104_B'
```

Now it is necessary to provide the database with a standard extinction curve and a standard star catalog, both of which can be found in the catalog checkout, (see section about relative photometry).

```
awe> from astro.main.PhotExtinctionCurve import PhotExtinctionCurve
awe> extcurve = PhotExtinctionCurve(pathname='cal564E.dat')
awe> extcurve.commit()
awe> from astro.main.PhotRefCatalog import PhotRefCatalog
awe> refcat = PhotRefCatalog(pathname='cal569E.cat')
awe> refcat.store()
awe> refcat.commit()
```

Use the following to create the final photometry calibration file (PhotometricParameters). Note how first the database is queried to get the filename of the data of Landolt 98. The file names depend on the date and time at which you process the data.

```
awe> from astro.main.ScienceFrame import ScienceFrame
awe> s = ScienceFrame.select_for_object(object_name='Land98_B')
13:11:32 - Querying database for instances of class ScienceFrame
awe> for f in s: print f.filename
...
Sci-DEMO-WFI-----#842-ccd56---Sci-53075.4696174.fits
Sci-DEMO-WFI-----#842-ccd55---Sci-53075.4696211.fits
Sci-DEMO-WFI-----#842-ccd57---Sci-53075.4696297.fits
Sci-DEMO-WFI-----#842-ccd54---Sci-53075.4696318.fits
Sci-DEMO-WFI-----#842-ccd53---Sci-53075.4696483.fits
Sci-DEMO-WFI-----#842-ccd51---Sci-53075.4696595.fits
Sci-DEMO-WFI-----#842-ccd52---Sci-53075.4697611.fits
Sci-DEMO-WFI-----#842-ccd50---Sci-53075.4696771.fits
awe> from DBRecipes.ZeropointSky import ZeropointSkyTask
```

```
awe> z = ZeropointSkyTask(image=
'Sci-DEMO-WFI-----#842-ccd5?---Sci-53075.469*.fits', commit=1)
awe> z.execute()
```

You can now run the image pipeline for this day.

1.7.4 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 $PYTHONPATH/parallel/generic_runner.py -task Science_Dither
      -log science_dither.log -i WFI -d 2000-04-28 -f #842 -o 'CDF4_B_?'
```

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 ScienceFrames that are stored in the database. In addition these science frames are resampled to a new grid. The grid centers for this system are fixed so that pixels in these *RegriddedFrames* can be combined without resampling again first.

After the job completes there should be 80 new ScienceFrames and 80 new RegriddedFrames in the database. One can check this from the awe/python interpreter as follows:

```
awe> from astro.main.ScienceFrame import ScienceFrame
awe> s = ScienceFrame.select_from_db(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.reduced.raw.filename, f.filter.name, f.chip.name,
f.reduced.raw.EXPTIME, f.reduced.raw.OBJECT
...
```

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

1.7.5 Coaddition

The RegriddedFrames created in the previous step can be coadded into a single mozaic, to form the intended contiguous region on the sky. This step can not be done on the computing cluster and must be done on a single machine. (All pixel data needs to be available in one place.)

Note: run the following on a single, local machine, not on the parallel cluster

In order to coadd the data, one can do the following:

```
awe> from DBRecipes.Coadd import CoaddTask
awe> task = CoaddTask(date='2000-04-28', instrument_name='WFI',
filter_name='#842', object_name='CDF4_B_?')
awe> task.execute()
```

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

1.7.6 Source lists

See chapter 2 for more specific information.

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> from Experimental.SourceList import SourceList
awe> from astro.main.RegriddedFrame import CoaddedRegriddedFrame
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()
```

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

Chapter 2

Source lists 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 have to be performed. The way this is handled in the Astro-Wise system is described in the following subsections. Beware that it is assumed that knowledge about some of the basic concepts that underly the system is present. If this is not the case, the reader is advised to read Sect. 6 first.

2.1 Creating simple source lists from science frames

In the Astro-Wise system source lists are represented by `SourceList` objects. To access the `SourceList` class, the user should do the following from the `awe` prompt:

```
awe> from Experimental.SourceList import SourceList
```

In order to derive a source list from a science frame, an instance of the `SourceList` class must be created and the science frame must be assigned to this object as a dependency. 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 :

```
awe> sourcelist = SourceList()
awe> sourcelist.frame = ScienceFrame(pathname='frame.fits')
awe> sourcelist.make()
```

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 :

```
awe> sourcelist = SourceList()
awe> sourcelist.frame = ScienceFrame(pathname='frame.fits')
awe> sourcelist.sexconf.DETECTION_THRESHOLD = 12
awe> sourcelist.sexparam = ['MAG_APER', 'MAGERR_APER']
```



```
awe> sourcelist.make()
```

with the extra output parameters added to the definition of the sources contained in the list.

2.2 External source lists

The description of the use of `SourceList` objects given in Sect. 2.1 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 the external source list given as its pathname, and calling the ‘`make_sourcelist_from_catalog`’ method. As seen from the `awe` prompt :

```
awe> sourcelist = SourceList(pathname='external.fits')
awe> sourcelist.make_sourcelist_from_catalog()
```

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.

2.3 Using 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 `SourceList 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> sourcelist = (SourceList.name == 'MySourcelists')
```

Each source in a `SourceList` has a unique `Source Identifier (SID)`, which is assigned during ingesting 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)
```

The column information is obtained as follows:

```
awe> column_info = sourcelist.sources.get_attributes()
```

`column_info` is a dictionary with columnnames as keys and columntypes as contents. Columndata is for example retrieved with the following statement

```
awe> RA = sourcelist.sources.RA
```

where `RA` is a list containing the values of column ‘`RA`’.

Rowdata is retrieved in a similar way, for example

```
awe> first_source = sourcelist.sources[0]
```

where `first_source` is a dictionary with columnnames as keys and columnvalues as contents.

There are some methods defined for sourcelists. At the moment they 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 a htm depth of 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 keys the SLID's (SourceList IDentifiers) and values lists of SID's (Source IDentifiers).

- `sourcelist.sources.sql_query(dict, query_string)`

`sql_query` performs a SQL query on source attributes.

`query_string` may contain only valid SQL query syntax and the unquoted names of the 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).

2.4 Associating source lists

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`.

- 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.

In python this is done as follows:

```
awe> AL = AssociateList()
awe> SLs = (SourceList.SLID == 0 and SourceList.SLID == 1)
awe> for SL in SLs: AL.sourcelists.append( SL )
awe> AL.set_search_distance(5.0)
awe> AL.make()
```

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
```

Once an `AssociateList` is created the number of pairs can be obtained as follows

```
awe> print AL.get_number_of_pairs()
```

and likewise the number of singles

```
awe> print AL.get_number_of_singles()
```

or the total number of associations

```
awe> print len(AL)
```

The pairs themselves are obtained with

```
awe> ( ( SL1, SID1 ), ( SL2, SID2 ) ) = AL.get_pairs()
awe> for i in range(len(SID1)):
awe> print ' ( %f, %f ) <=> ( %f, %f ) % (' \
awe> SL1.sources[SID1[i]]['RA'], SL1.sources[SID1[i]]['DEC'], \
```

```
awe> SL2.sources[SID1[i]]['RA'], SL2.sources[SID2[i]]['DEC'] )
```

where SL1 and SL2 are instances of the two input sourcelists and SID1 and SID2 lists containing the source ID's of the associated sources.

The singles are obtained with

```
awe> ( ( SL1, SID1 ), ( SL2, SID2 ) ) = AL.get_singles()
awe> for SID in SID1:
awe>   print 'Single from first SourceList: ( %f, %f )' % ( \
awe>     SL1.sources[SID1[i]]['RA'], SL1.sources[SID1[i]]['DEC'] )
awe> for SID in SID2:
awe>   print 'Single from second SourceList: ( %f, %f )' % ( \
awe>     SL2.sources[SID2[i]]['RA'], SL2.sources[SID2[i]]['DEC'] )
```

A much more useful function is the function `get_attributes_of_pairs()` which returns the values of the attributes of the pairs. This makes it possible to for example compare the magnitudes of the members of the pairs, by doing

```
awe> a1 = { 'MAG_ISO' : [] }
awe> a2 = { 'MAG_ISO' : [] }
awe> aids = AL.get_attributes_of_pairs( a1, a2 )
```

The magnitudes of the sources from the first sourcelist are to be found in `a1['MAG_ISO']`. Likewise, the magnitudes from the second sourcelist are in `a2['MAG_ISO']`. The list `aids` contains the association ID's of the associated sources, sorted in ascending order. Any number of (existing) attributes may be given in the dictionary, as for example in

```
awe> a1 = { 'RA' : [], 'DEC' : [] }
awe> a2 = { 'RA' : [], 'DEC' : [] }
awe> aids = AL.get_attributes_of_pairs( a1, a2 )
```

With the above call we can now check whether the distance between the members of the pairs is indeed less than the search distance, i.e.

```
awe> for (r1,r2,d1,d2) in zip(a1['RA'],a2['RA'],a1['DEC'],a2['DEC']):
awe>   print r1, r2, d1, d2
```

Similarly the function `get_attributes_from_singles` exists for obtaining the attributes of single sources from a specified sourcelist, like:

```
awe> a1 = { 'RA' : [], 'DEC' : [] }
```

```
awe> a2 = { 'RA' : [], 'DEC' : [] }  
awe> aids1 = AL.get_attributes_of_singles( 1, a1 )  
awe> aids2 = AL.get_attributes_of_singles( 2, a2 )
```

Chapter 3

Documentation, manual pages

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

3.1 Online documentation

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 opipe checkout. The server is accessible as

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

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

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

3.2 Inline documentation

It is possible to access docstrings from the python/awe prompt. This is often convenient and faster than using a pydoc server. One can enter the Python help environment 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:

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

```

astro.main.Astrom - the world coordinate system of a FITS file
astro.main.AstrometricCatalog
astro.main.AstrometricCorrection
astro.main.AtmosphericExtinction
astro.main.AtmosphericExtinctionCurve
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 - the object indentifying 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
astro.main.DomeFlatFrame - dome flat (req542), master dome flats
astro.main.Filter - the object identifying the filter
astro.main.FilterTransmissionCurve
astro.main.FringeFrame - fringe images (req545)
astro.main.GainLinearity - gain (req523) and linearity (req533)
astro.main.HotPixelMap - hot pixel maps (req522)
astro.main.IlluminationCorrectionFrame - illumination correction frames (req548)
astro.main.Imstat - image statistics
astro.main.Instrument - the object identifying the instrument
astro.main.LinearityMap - non-linear pixels (req533)
astro.main.LongAstrom
astro.main.MasterFlatFrame - defines the final flat-field to be used (req546)
astro.main.NightSkyFlatFrame - module NightSkyFlatFrame
astro.main.ObservingBlock - module ObservingBlock
astro.main.PhotExtinctionCurve
astro.main.PhotRefCatalog
astro.main.PhotSkyBrightness
astro.main.PhotSrcCatalog
astro.main.PhotTransformation
astro.main.PhotometricExtinctionReport
astro.main.PhotometricParameters
astro.main.PhotometricSkyReport
astro.main.PixelMap - defines a class for pixel maps
astro.main.ProcessTarget - processable objects with quality control flags
astro.main.QuickDetectorResponsivity - quick check (req547)
astro.main.RawFitsData - raw (un-split, multi-extension FITS) data
astro.main.RawFrame - classes for the different kinds of raw frames
astro.main.ReadNoise - read noise (req521)
astro.main.ReducedScienceFrame - de-bias & flat-field (seq632), apply astrometry (seq634/req555)
astro.main.RegriddedFrame - the support classes for regridding and coaddition
astro.main.SaturatedPixelMap - maps of saturated pixels
astro.main.ScienceFrame - apply photometry (seq635)
astro.main.ShutterCheck - shutter timing (req561)
astro.main.Template
astro.main.TwilightFlatFrame - twilight flat (req543), master twilight flats
astro.main.WeightFrame - individual weights (seq633)
astro.main (package) - The Persistent Object Hierarchy

```

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
  /zer1/users/helmich/opipe/astro/main/BiasFrame.py

DESCRIPTION
  This module contains class definitions for BiasFrameParameters
  and BiasFrame.

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

  BiasFrame is the class that defines a master bias object.

CLASSES
  astro.main.BaseFrame.BaseFrame(astro.main.DataObject.DataObject, astro.main.ProcessTarget.ProcessTarget)
  BiasFrame
  astro.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 frame
  |                 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
  |   astro.database.DBMain.DBObject
  |   astro.main.ProcessTarget.ProcessTarget
  |   astro.database.DBMeta.DBMixin
  |   __builtin__.object
  |
  | Methods defined here:
  |
  | QC_sigttest(self)
  |     Quality control tool for Bias
  |
  |     The algorithm splits the input frame in a number of subwindows and then
  |     counts, for each window, the excess of elements greater than the median
  |     (respect to those smaller); if this number is not compatible with the
  |     binomial distribution, then the subwindow is flagged. Finally the
  |     number of bad windows is compared with the theoretical number of bad
  |     windows expected; the quality index, then is defined as
  |     (NBAD-NEXPECTED)/sigma.
  |
  | __init__(self, pathname='')
  |
  | check_preconditions(self)
  |
  | compare(self)
  |     Compare the results with a previous version.

```



```

    Requires:
        prev -- A BiasFrame object for the previous Bias measurement

    The following flags may be set in the status attribute:
        MEAN_DIFFER -- (mean of bias-mean of previous) > MAXIMUM_BIAS_DIFFERENCE

copy_observing_block(self)

derive_timestamp(self)
    Assign the default period for which this calibration frame is valid.
    For a BiasFrame the start of this period is noon of the day preceding
    a night of observing. The end of the period is noon of the day
    following a night of observing. Note that the default period for which
    a BiasFrame is valid is linked to observing frequency, which is
    described in the OmegaCAM URD and Calibration Plan.

inspect(self)
    Optional visual inspection for quality control TBD displays image or
    safe image for later display.

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
        bias exposures.

    Averages the input frames only.

verify(self)
    Verify the results.

    The following flags may be set in the status attribute:
        MEAN_LOW -- mean of bias < MINIMUM_BIAS_LEVEL
        MEAN_HIGH -- mean of bias > MAXIMUM_BIAS_LEVEL
    -----
    Properties 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. The exact same page can be called with the following commands (i.e. at the normal Python prompt, rather than the help environment):

```

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)

```

These pages contain exactly the same information as the HTML documentation.

Chapter 4

Frequently Asked Questions

FAQ 1: 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 ?

Answer: make sure that you are not importing something from within the opipe directory structure. Due to some strange python things concerning packages, this can lead to completely intractable error messages.

FAQ 2: I am installing the Astro-Wise system on my box, but when I try to ‘install’ e.g. eclipse I get the comment that some file or directory is missing.

Answer: 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’.

FAQ 3: I want to run Sextractor on my images. How do I do that ?

Answer: 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, i.e. 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

FAQ 4: I have seen the software of the photometric pipeline, and I am blown away by its sheer brilliance. What can I do to pay homage to the author ?

Answer: Transfer large amounts of money to the bank account of the author (account number available on request).

Part II

Developer's manual

Chapter 5

Development

In the next few 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 (`for...in`, `if...elif...else`), 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 `opipe`) is in your `PYTHONPATH`, then you should see a link, near the top of the page, to the `astro` package containing the library modules, and the `Recipes` package containing recipes (surprise).

Chapter 6

Key concepts

6.1 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 ??? for further details)

6.1.1 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 sourcecode 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 `RawBias` 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 useally called from the `make()` method. The value of the special attribute `process_status` is updated to record this.

`get_depedencies()` 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 `Recipes` of the library. Have a look at these recipes, including the Bias recipe, to see that all look extremely similar to this example.

6.2 Verification and quality control

In order to verify the results of the data processing, makeable 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 makeable objects:

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

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

`is_inspected()` returns 1 if `inspect()` has been successfully 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
```

Chapter 7

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 7.1 gives an overview of this class hierarchy. 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 `DBObject`s 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. 7.1) 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)

¹A ‘mixin’ class is a class that adds behaviour to another class in a derived class using multiple inheritance

Chapter 8

Database Tasks

8.1 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 howto) and that the environment can be started with the `awe` command.

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

```
database_engine: oracle_9i
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 astro.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 and an `AWUSER` role

```
awe opipe/Toolbox/dbawoper.py
```

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

```
awe opipe/Toolbox/dbimportall.py
```

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

```
awe opipe/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 opipe/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 opipe/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`.

8.2 Keeping the 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 successfully 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

```
cv$ tag -r revision_number source_file
```

8.3 Database Type Evolution

8.3.1 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 to 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;
```

8.3.2 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*.

8.3.3 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>*.

8.3.4 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.

8.3.5 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.

8.3.6 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 8.3.11.

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

¹Use `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$';
```

8.3.7 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;
```

8.3.8 Removing persistent 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 section 8.3.3.

8.3.9 Renaming a persistent attribute

To rename a persistent attribute the procedures described in 8.3.7 and 8.3.8 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;
```

8.3.10 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 section 8.3.7.
- 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 section 8.3.8.

- Before the dummy attribute can be removed, the attribute whose type is changed needs to be added with its final type, as explained in section 8.3.7.
- 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 section 8.3.8.

8.3.11 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.

8.3.12 Error messages

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

Appendix A

Where to find things

A.1 Where to find things

The astro-wise website

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

The CVS repository

`cvs.astro-wise.org:/cvsroot`

The CVS webservice

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

Data location in Groningen containing both OmegaCAM data from the lab, data taken specifically for OmegaCAM calibration and other wide-field data.

`/zer1/users/omegacam`

Appendix B

Quick guide to making a test schema in the database

Because the OmegaCAM database is a multi-user environment and any changes to the Python code base can have nasty consequences for those other users, it is advisable to try out changes to the code in a test environment. This chapter intends to give a quick overview of how to do this.

B.1 Summary

It is assumed all necessary software components are installed and a working database is in operation. To make a test schema in this database and start processing, contact your database administrator and have him/her create a new user directory (schema) on the existing database. The following actions should then be performed:

- Do a new checkout of the **opipe** directory
- Create your own data server and start the server daemon
- Change your environment variables for the new schema/operator combination
- Bootstrap the test schema
- Ingest data into the database
- Perform and test your changes

B.2 Checkout of CVS code base

A new checkout of the CVS code base can be made with the following command:

```
cv$ -d cvs.astro-wise.org:/cvsroot co opipe
```

Make sure that the environment variable PYTHONPATH points to this checkout rather than any old one. Change this either in your **.cshrc** file or by

```
setenv PYTHONPATH /the/path/to/my/disk/opipe
```

B.3 Creating a data server

One must find a disk with enough free space. Create a directory **data** containing a subdirectory **cache**. This will be the place data is *stored* to and *retrieved* from. Go to this something/data/ directory, and start the data server daemon:

```
awe -u /disk/user/opipe/parallel/dataserver_server.py 129.125.6.201 8157 &
```

Where the numbers are the IP address and port number of the dataserver. Note that four consecutive ports are claimed, starting with the given port. Something like following lines will be printed:

```
omegacen01>Starting dataserver peer process at Thu Oct 23 12:03:50 2003
Serving
Server running
Server running
```

B.4 Changing relevant environment variables

Aside from the abovementioned PYTHONPATH variable, the information for the access to your database must be specified. This is done in Environment.cfg located in the **.awe** directory in your home directory. This file contains lines such as the following:

```
[astro_rug_n1]
testdata_dir      : /zer1/users/helmich/unittestdata

data_server       : 129.125.6.201
data_port         : 8157

database_engine   : oracle_9i
database_name     : aw98.zernike.nova.aw
database_user     : ops$helmich
database_password : password
```

B.5 Bootstrap the test schema

Some recipes require data to be present in the database. MasterFlat (req546) requires information about hot and cold pixels. BiasFrames and DomeFlatFrames or TwilightFlatFrames (the sources of information for hot and cold pixels) can be made without this information though.

A GainLinearity object as well as a PhotometricParameters object are necessary for the photometry recipes. There is a tool to create the latter but a few standard catalogs need to be obtained from CVS:

```
cvs -d cvs.astro-wise.org:/cvsroot co catalog
```

followed by:

```
awe $PYTHONPATH/Toolbox/ingest_photometrics.py <options>
```

where one of the options is a catalog located in the catalog checkout. (Refer to the usage of ingest_photometrics.py for syntax.) A GainLinearity object can be made by hand using the Oracle Enterprise Manager GUI. Open Databases → AW98 → Schema → OPS\$YOURNAME → Tables. Then right click on GainLinearity and select "View/Edit Contents...". Fill in the timestamp attributes as well as the gain, and 0 as quality_flag (i.e. data is valid).

B.6 Ingesting data into the database

Ingesting data into the database is handled by **ingest.py** located in **Toolbox**. Note however that you should create links to the relevant files, rather than specify a full pathname. One approach could be to first create ascii files that contain the names of the fits files that should be ingested. These files can be per type (bias, dome flat, twilight flat, science) of raw image data. Consider for example the following case where INT-WFC (4 CCDs) twilight flats are ingested:

```
omegacen01>foreach i ( 'grep .fits intTWILIGHTFLAT.contents' )
foreach? ln -s /ome01/users/data/INT/2003/02/$i $i
foreach? end
```

```
awe $PYTHONPATH/Toolbox/ingest.py -i *.fits -t twilight -n 4
```