# AstroData Users Manual

***Release 0.9.0***

**Kathleen Labrie**

April 04, 2014

# CONTENTS

Contents:

CHAPTER

ONE

# INTRODUCTION

## 1.1 What is AstroData?

(use language a scientist will comprehend. then either refer to the programmer's reference or add a section clearly advertised to programmers. the idea is that we don't want to lose the scientists, but if some readers are more technically oriented we also want to make sure they get the info they are after.)

## 1.2 Installing AstroData

(pre-requisites, build, install, configure)

## 1.3 AstroData Support

(online webpage, gemini helpdesk)

CHAPTER

TWO

# TUTORIAL

## 2.1 Open and Access MEF Files

from astrodata import AstroData

## 2.2 Operate on MEF Files

## 2.3 Create and Update MEF Files

## 2.4 Writing a Python Function using AstroData

CHAPTER

THREE

# INPUT AND OUTPUT OPERATIONS

## 3.1 Open Existing MEF Files

An AstroData object can be created from the name of the file on disk or from PyFITS HDUList. An existing MEF file can be open as an AstroData object in `readonly`, `update`, or `append` mode. The default is `readonly`.

*(KL: why would anyone want to create an AD from another AD??!!) (KL: what's the deal with store and storeClobber? Incomprehensible.)*

Here is a very simple example on how to open a file in `readonly` mode, check the structure, and then close it:

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')
ad.info()
ad.close()
```

To open the file in a mode other than `readonly`, specify the value of the `mode` argument:

```
ad = AstroData('N20111124S0203.fits', mode='update')
```

## 3.2 Update Existing MEF Files

To update an existing MEF file, it must have been opened in the `update` mode. Then a collection of methods can be applied to the AstroData object. Here we give examples on how to append an extension, how to insert an extension, how to remove an extension, and how to replace an extension. Then we show how do basic arithmetics on the pixel data and the headers in a loop. Manipulations of the pixel data and of the headers are covered in more details in later sections (?? and ??, respectively). Finally we show how to write the updated AstroData object to disk as MEF file.

```
from astrodata import AstroData

# Open the file to update
ad = AstroData('N20110313S0188.fits', mode='update')
ad.info()

# Get an already formed extension from another file (just for the
# sake of keeping the example simple)
adread = AstroData('N20110316S0321.fits', mode='readonly')
new_extension = adread["SCI",2]

# Append an extension.
# WARNING: new_extension has EXTNAME=SCI and EXTVER=2
```

```
#          ad already has an extension SCI,2.
#          To avoid conflict, the appended extension needs
#          to be renumbered to SCI,4. auto_number=True takes
#          care of that.
# WARNING: renumbering the appended extension will affect
#          adread as new_extension is just a pointer to that
#          extension in adread.  To avoid the modification of
#          adread, one either does the deepcopy before the
#          call to append, or set the do_deepcopy argument
#          to True, as we do here.
ad.append(new_extension,auto_number=True,do_deepcopy=True)
ad.info()

# Insert an extension between two already existing extensions.
#
# Let's first rename the new_extension to make it stand out once
# inserted.
new_extension = adread['SCI',1]
new_extension.rename_ext('VAR')
new_extension.info()

#   Here we insert the extension between the PHU and the first
#   extension.
#   WARNING: An AstroData object is a PHU with a list of HDU, the
#            extensions. In AstroData, the extension numbering is zero-based.
#            Eg. in IRAF myMEF[1] -> in AstroData ad[0]
ad.insert(0, new_extension)
ad.info()

# Note that because the extension was named ('VAR',1) and that did not
# conflict with any of the extensions already present, we did not have
# to use auto_number=True.

#   Here we insert the extension between the third and the fourth
#   extensions.  Again, remember that the extension numbering is
#   zero-based.
ad.insert(3, new_extension, auto_number=True, do_deepcopy=True)

# A ('VAR',1) extension already exists in ad, therefore auto_number must
# be set to True.  Since we are insert the same new_extension, if we don't
# deepcopy it, the EXTVER of the previous insert will also change.
# Remember in Python, you might change the name of a variable, but both
# will continue pointing to the same data: change one and the other will
# change too.

# Here we insert the extension between [SCI,3] and [SCI,4]
# Note that the position we use for the index is ('SCI',4)
# This is because we effectively asking for the new extension
# to push ('SCI',4) and take its place in the sequence.
#
new_extension = adread['SCI',3]
new_extension.rename_ext('VAR')
ad.insert(('SCI',4), new_extension)
ad.info()

# Now that we have made a nice mess of ad, let's remove some extensions
# Removing AstroData extension 4 (0-based array).
ad.remove(4)
```

```
ad.info()

# Removing extension ['VAR',5]
ad.remove(('VAR',5))
ad.info()

# Here is how to replace an extension.
# Let's replace extension ('SCI',2) with the ('SCI',2) extension from adread.

##### .replace() is broken.  Will add example when it's fixed.


# Finally, let's write this modified AstroData object to disk as a MEF file.
# The input MEF was open in update mode.  If no file name is provide to the
# write command, the file will be overwritten.  To write to a new file,
# specify a filename.
ad.filename
ad.write('newfile.fits')
ad.filename

# Note that any further write() would now write to 'newfile.fits' if no filename
# is specified.

# The pixel data and header data obviously can be accessed and modified.
# More on pixel data manipulation in ???.  More on header manipulation in ???

import numpy as np

for extension in ad:
   # Obtain a numpy.ndarray. Then any ndarray operations are valid.
   data = ext.data
   type(data)
   np.average(data)

   # Obtain a pyfits header.
   hdr = ext.header
   print hdr.get('NAXIS2')

# the numpy.ndarray can also be extracted this way.
data = ad[('SCI',1)].data

# To close an AstroData object.  It is recommended to properly close the object
# when it will no longer be used.
ad.close()
adread.close()
```

## 3.3 Create New MEF Files

The method `write` is use to write to disk a new MEF file from an AstroData object. Here we show two ways to build that new AstroData object and create a MEF file, in memory or on disk, from that AstroData object.

### 3.3.1 Create New Copy of MEF Files

Let us consider the case where you already have a MEF file on disk and you want to work on it and write the modified MEF to a new file.

Here we open a file, make a copy, and write a new MEF file on disk:

```
from astrodata import AstroData

ad = AstroData('N20110313S0188.fits')
ad.write('newfile2.fits')
ad.close()
```

Since in Python and when working with AstroData objects, the memory can be shared between variables, it is sometimes necessary to create a "true" copy of an AstroData object to keep us from modifying the original.

By using `deepcopy` on an AstroData object the copy is a true copy, it has its own memory allocation. This allows one to modify the copy while leave the original AstroData intact. This feature is useful when an operation requires both the modified and the original AstroData object since by design a simple copy still point to the same location in memory.

```
from astrodata import AstroData
from copy import deepcopy

ad = AstroData('N20110313S0188.fits')
adcopy = deepcopy(ad)
```

In the example above, `adcopy` is now completely independent of `ad`. This also means that you have doubled the memory footprint.

### 3.3.2 Create New MEF Files from Scratch

Another use case is creating a new MEF files when none existed before. The pixel data needs to be created as a numpy ndarray. The header must be created as pyfits header.

```
from astrodata import AstroData
import pyfits as pf
import numpy as np

# Create an empty header.  AstroData will take care of adding the minimal
# set of header cards to make the file FITS compliant.
new_header = pf.Header()

# Create a pixel data array.  Fill it with whatever values you need.
# Here we just create a fill gradient.
new_data = numpy.linspace(0., 1000., 2048*1024).reshape(2048,1024)

# Create an AstroData object and give it a filename
new_ad = AstroData(data=new_data, header=new_header)
new_ad.filename = 'gradient.fits'

# Write the file to disk and close
new_ad.write()
new_ad.close()
```

CHAPTER

FOUR

# MEF STRUCTURE MAPPING

## 4.1 File Structure Definitions

## 4.2 Gemini Data Structure

(SCI, VAR, DQ, MDF)

## 4.3 Using Structures

## 4.4 Adding New Structure

CHAPTER

FIVE

# ASTRODATATYPES

## 5.1 What are AstroDataTypes

(explain what they are. data type & data processing status. classification based on headers only. explain how to install the Gemini types.)

## 5.2 Using AstroDataTypes

There are two ways to check the AstroDataTypes of a dataset:

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')

if ad.is_type('GMOS_IMAGING'):
    # do special steps for GMOS_IMAGING type data

if 'GMOS_IMAGING' in ad.types:
    # do special steps for GMOS_IMAGING type data
```

The attribute `ad.types` returns a list of all the AstroDataTypes associated with the dataset. It can be useful when interactively exploring the various types associated with a dataset, or when there's a need to write all the types to the screen or to a file, for logging purposes, for example. Use at your discretion based on your need.

"Data Types" are referred to as *Typology* in the AstroDataTypes code. "Data Processing Status" are referred to as *Status*. There are two additional attributes that might be useful if those two concepts need to be addressed separately: `ad.typesStatus` and `ad.typesTypology`. They are used exactly the same way as `ad.types`.

??? ad.refresh_types()

## 5.3 Creating New AstroDataTypes

(refer to programmer's manual, but give some idea of what needs to be done and the basic principles)

CHAPTER

SIX

# FITS HEADERS

## 6.1 AstroData Descriptors

AstroData Descriptors provide a "header keyword-to-concept" mapping that allows one to access header information in a consistent manner, regardless of which instrument the dataset is from. The mapping is coded in a configuration package that is provided by the observatory or the user.

For example, if one were interested to know the filter used for an observation, normally one would need to know which specific keyword or set of keywords to look at. Once the concept of "filter" is coded in a Descriptor, one now only needs to call the `filtername` Descriptor.

To get the list of descriptors available for an AstroData object:

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')
ad.all_descriptor_names()
```

Most Descriptor names are readily understood, but one can get a short description of what the Descriptor refers to by call the Python help function, for example:

```
help(ad.airmass)
```

Descriptors associated with standard FITS keywords are available from the `ADCONFIG_FITS` package distributed in `astrodata_FITS`. All the Descriptors associated with other concepts used by the Gemini software are found in the `ADCONFIG_Gemini` package, part of `astrodata_Gemini`.

As a user reducing Gemini data or coding for existing Gemini data, all you need to do is make sure that astrodata_FITS and astrodata_Gemini have been installed. If you are coding for a new Gemini instrument, or for another observatory, Descriptors and AstrodataTypes will need to be coded. That's a more advanced topic addressed elsewhere. (KL?? ref to last section of this page)

## 6.2 Accessing Headers

Whenever possible the Descriptors should be used to get information from the headers. This allows for maximum re-use of the code as it will work on any datasets with an AstroDataTypes. Here are a few examples using Descriptors:

```
from astrodata import AstroData
from copy import deepcopy

ad = AstroData('N20111124S0203.fits')
adcopy = deepcopy(ad)
```

```
print 'The airmass is : ',ad.airmass()

if ad.exposure_time() < 240.:
  print 'This is a short exposure'

# This call will multiply the pixel values in all three science extensions
# by their respective gain.  There's no need to loop through the science
# extension explicitly.
adcopy.mult(adcopy.gain())

fhwm_arcsec = 3.5 * ad.pixel_scale()
```

Of course not all the header content has been mapped with Descriptors. Here is how to get the value of a specific header keyword:

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')

# Get keyword value from the PHU
aofold_position = ad.phu_get_key_value('AOFOLD')

# Get keyword value from a specific extension
naxis2 = ad.ext_get_key_value(('SCI',1), 'NAXIS2')

# Get keyword value from an extension when there's only one extension
# This happens, for example, when looping through multiple extensions.
for extension in ad['SCI']:
   naxis2 = extension.get_key_value('NAXIS2')
   print naxis2
```

Multi-extension FITS files, MEF, have this concept of naming and versioning the extensions. The header keywords controlling name and version are `EXTNAME` and `EXTVER`. AstroData uses that concept extensively. See ??? for information on the typical structure of AstroData objects. The name and version of an extension is obtained this way:

```
name = ad[1].extname()
version = ad[1].extver()
print name, version
```

To get a whole header from an AstroData object, one would do:

```
# Get the header for the PHU as a pyfits Header object
phuhdr = ad.phu.header

# Get the header for extension SCI, 1 as a pyfits Header object
exthdr = ad['SCI',1].header

# print the header content in the interactive shell
# For a specific extension:
ad['SCI',2].header
# For all the extensions:  (PHU excluded)
ad.get_headers()

ad.close()
```

## 6.3 Updating and Adding Headers

Header cards can be updated or added to header. As for the access to the headers, the PHU have their own methods, different from the extension, but essentially doing the same thing. To write to a PHU use the `phu_set_key_value()` method. To write to the header of an extension, use the `ext_set_key_values()`. The difference is that one has to specify the extension ID in the latter case.

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')

# Add a header card to the PHU
#  The arguments are *keyword*, *value*, *comment*.  The comment is optional.
ad.phu_set_key_value('MYTEST', 99, 'Some meaningless keyword')

# Modify a header card in the second extension
#  The arguments are *extension*, *keyword*, *value*, *comment*.  The comment
#  is optional.  If a comment already exists, it will be left untouched.
ad.ext_set_key_value(1,'GAIN',5.)

# The extension can also be specified by name and version.
ad.ext_set_key_value(('SCI',2), 'GAIN', 10.)

# A utility method also exists for use in astrodata objects that contain
# only one extension.  This is particularly useful when looping through
# the extensions.  There's no need to specify the extension number since
# there's only one.  The arguments are *keyword*, *value*, *comment*, with
# comment being optional.
for extension in ad['SCI']:
    extension.set_key_value('TEST',9, 'This is a test.')
```

The name and version of an extension can be set or reset manually with the `rename_ext` method:

```
ad['SCI',1].rename_ext('VAR',4)
```

Be careful with this function. Having two extensions with the same name and version in an AstroData data object, or a MEF files for that matter, can lead to strange problems.

## 6.4 Adding Descriptors Definitions for New Instruments

(refer to Emma's document.)

CHAPTER

SEVEN

# PIXEL DATA

## 7.1 Operate on the Pixel Data

The pixel data is stored as a numpy ndarray. This means that anything that can be done with numpy on a ndarray can be done on the pixel data stored in the AstroData object. Examples include arithmetic, statistics, display, plotting, etc. Please refer to numpy documentation for details on what it offers. In this chapter, we will present some typical examples.

But first, here's how one accesses the data array stored in an AstroData object:

```
from astrodata import AstroData

ad = AstroData('N20110313S0188.fits')

# The PHU does not have any pixel data.  Only the extensions can have pixel data.
the_data = ad['SCI',2].data

# or to loop through the extensions.  Here we just print the sum of all the pixels
# for each extension.
for extension in ad['SCI']:
   the_data = extension.data
   print the_data.sum()

ad.close()
```

## 7.2 Arithmetic on AstroData Objects

AstroData supports basic arithmetic directly: addition, subtraction, multiplication, division. The big advantage of using the AstroData implementation of those operator is that if the AstroData object has variance and data quality planes, those will be calculated and propagated to the output appropriately.

```
from astrodata import AstroData

ad = AstroData('N20110313S0188.fits')

# addition
#   ad = ad + 5.
ad.add(5.)

# subtraction
#   ad = ad - 5.
```

```
ad.sub(5.)

# multiplication.  Using descriptor as operand.
#   ad = ad * gain
ad.mult(ad.gain())

# division. Using descriptor as operand.
#   ad = ad / gain
ad.div(ad.gain())
```

When using the AstroData arithmetic, all the science (EXTNAME='SCI') frames are operated on.

The AstroData arithmetic methods can be stringed together. Note that because the calculations are done "in-place", operator precedence cannot be respected.

```
ad.add(5).mult(10).sub(5)
# means:  ad = ((ad + 5) * 10) - 5
# not: ad = ad + (5 * 10) - 5

ad.close()
```

The AstroData data arithmetic method modify the data "in-place". This means that the data values are modified and the original values are no more. If you need to keep the original values unmodified, for example, you will need them later, use `deepcopy` to make a separate copy on which you can work.

```
from astrodata import AstroData
from copy import deepcopy

ad = AstroData('N20110313S0188.fits')

# To do:  x = x*10 + x
#  One must use deepcopy because after the mult(10), 'ad' has been modified
#  and it is that modified version that will be use in add(ad)

# Let's follow a pixel through the math

value_before = ad['SCI',1].data[50,50]
expected_value_after = value_before*10 + value_before

ad.mult(10).add(ad)
bad_value_after = ad['SCI',1].data[50,50]

print expected_value_after, bad_value_after

# The result of the arithmetic above is x = (x*10) + (x*10)

# To do the right thing, one can use ``deepcopy``
# First let's reload a fresh ad.
ad = AstroData('N20110313S0188.fits')
adcopy = deepcopy(ad)

ad.add(adcopy.mult(10))

good_value_after = ad['SCI',1].data[50,50]
print expected_mean_after, good_mean_after

ad.close()
adcopy.close()
```

As one can see, for complex equation, using the AstroData arithmetic method can get fairly confusing. Operator overload would solve this situation but it has not been implemented yet. Therefore, we recommend to use numpy for really complex equation since operator overload is implemented and the operator precedence is respected. The downside is that if you need the variance plane propagate correctly, you will have to do the math yourself.

```
from astrodata import AstroData

ad = AstroData('N20110313S0188.fits')

# Let's do 'x = x*10 + x' again but this time we operate directly on
# the numpy ndarray return by '.data'.  We will follow a pixel through
# the math like before.

value_before = ad['SCI',1].data[50,50]
expected_value_after = value_before*10 + value_before

for extension in ad['SCI']:
    data_array = extension.data
    data_array = data_array*10 + data_array
    extension.data = data_array

value_after = ad['SCI',1].data[50,50]
print expected_value_after, value_after

ad.close()
```

## 7.3 Variance

Here we demonstrate the variance propagation when using AstroData arithmetic methods. First let us create and append variance planes to our file. We will just add the poisson noise and ignore read noise for the purpose of this example.

```
from astrodata import AstroData
from copy import deepcopy

ad = AstroData('N20110313S0188.fits')
ad.info()

for extension in ad['SCI']:
    variance = extension.data / extension.gain().as_pytype()
    variance_header = extension.header
    variance_extension = AstroData(data=variance, header=variance_header)
    variance_extension.rename_ext('VAR')
    ad.append(variance_extension)

ad.info()

# Let's just save a copy of this ad for later use.
advar = deepcopy(ad)
```

Now let us follow a science pixel and a variance pixel through the AstroData arithmetic.

```
#     output = x * x
# var_output = var * x^2 + var * x^2

value_before = ad['SCI',1].data[50,50]
```

```
variance_before = ad['VAR',1].data[50,50]
expected_value_after = value_before + value_before
expected_variance_after = 2 * (variance_before * value_before * value_before)

ad.mult(ad)

value_after = ad['SCI',1].data[50,50]
variance_after = ad['VAR',1].data[50,50]
print expected_value_after, value_after
print expected_variance_after, variance_after

ad.close()
```

So all it took to multiply the science extensions by themselves and propagate the variance accordingly was `ad.mult(ad)`.

To do the same thing operating directly on the numpy array:

```
# Let's recall the ad with the variance planes we created earlier
ad = deepcopy(advar)

for i in range(1,ad.count_exts('SCI')+1):
    d = ad['SCI',i].data
    v = ad['VAR',i].data
    data = d*d
    variance = v * d*d + v * d*d
    ad['SCI',i].data = data
    ad['VAR',i].data = variance

print ad['VAR',1].data[50,50]
```

## 7.4 Display

Displaying `numpy` arrays from Python is straighforward with the `numdisplay` module. The module also has a function to read the position the cursor, which can be useful when developing an interactive task.

Start a display tool, like DS9 or ximtool. Then try the commands below.:

```
from astrodata import AstroData
from numdisplay import display
from numdisplay import readcursor

ad = AstroData('N20110313S0188.fits')

display(ad['SCI',1].data)

# To scale "a la IRAF"
display(ad['SCI',1].data, zscale=True)

# To set the minimum and maximum values to display
display(ad['SCI',1].data, z1=700, z2=10000)
```

If you need to retrieve cursor position inputs, the numdisplay.readcursor function can help. It does not respond to mouse clicks, but it does respond to keyboard entries.:

```
# Invoke readcursor() and put the cursor on top of the image.
# Type any key.
```

```
# cursor_coo will contain the x, y positions and in the last column the key that was typed.
cursor_coo = readcursor()
print cursor_coo

# If you just want to extract the x,y coordinates:
(xcoo, ycoo) = cursor_coo.split()[:2]
print xcoo, ycoo

# If you are also interested in the keystoke:
(xcoo, ycoo, junk, keystroke) = cursor_coo.split()
print 'You pressed this key: "%s"' % keystroke
```

## 7.5 Useful tools from the Numpy and SciPy Modules

The `numpy` and `scipy` modules offer a multitude of functions and tools. They both have their own documentation. Here we simply highlight a few functions that could be used for common things an astronomer might want to do. The idea is to get the reader started in her exploration of `numpy` and `scipy`.

```
from astrodata import AstroData
import numpy as np
import numpy.ma as ma
import scipy.ndimage.filters as filters
from numdisplay import display

ad = AstroData('N20110313S0188.fits')
data = ad['SCI',2].data

# The shape of the ndarray stored in data is given by .shape
# The first number is NAXIS2, the second number is NAXIS1.
data.shape

# Calculate the mean and median of the entire array.
# Note how the way mean and median are called differently.
data.mean()
np.median(data)

# If the desired operation is a clipped mean, ie. rejecting
# values before calculating the mean, the numpy.ma module
# can be used to mask the data.  Let's try a clipped mean
# at -3 and +3 times the standard deviation

# ma.masked_outside() with mask out anything outside +/- 3*stddev of the mean.
# mask_extreme contains the "mask" returned by masked_outside()
stddev = data.std()
mean = data.mean()
mask_extreme = ma.masked_outside(data, mean-3*stddev, mean+3*stddev).mask

# ma.array() applies the mask to data.
# The compressed() method converts the masked data into a ndarray on
# which we can run .mean().
clipped_mean = ma.array(data, mask=mask_extreme).compressed().mean()

# Another common image operation is the filtering of an image.
# To gaussian filter an image, use scipy.ndimage.filters.gaussian_filter.
# The filters module offers several other functions for image processing,
# see help(filters)
```

```
conv_data = np.zeros(data.size).reshape(data.shape)
sigma = 10.
filters.gaussian_filter(data, sigma, output=conv_data)
display(data, zscale=True)
display(conv_data, zscale=True)

# If you wanted to put this convoled data back in the AstroData
# object you would do:
ad['SCI',2].data = conv_data
```

## 7.6 Using the AstroData Data Quality Plane

TO BE WRITTEN -> transform DQ plane into a numpy mask and do statistics

## 7.7 Manipulate Data Sections

Sections of the data array can be accessed and processed. It is important to note here that when indexing a numpy array, the left most number refers to the highest dimension's axis. Also important, is to remember that the numpy arrays are 0-indexed, not 1-indexed like in Fortran and IRAF. For example, in a 2-D numpy array, the pixel position (x,y) = (50,75) would be accessed as data[74,49].

Here are some examples using data sections.:

```
from astrodata import AstroData
import numpy as np

ad = AstroData('N20110313S0188.fits')
data = ad['SCI',2].data

# Let's get statistics for a 25x25 pixel-wide box centered on pixel 50,75.
mean = data[62:87,37:62].mean()
median = np.median(data[62:87,37:62])
stddev = data[62:87,37:62].std()
minimum = data[62:87,37:62].min()
maximum = data[62:87,37:62].max()
print "Mean      Median Stddev       Min    Max\n", mean, median, stddev, minimum, maximum
```

Now let us apply our knownledge so far to do a quick overscan subtraction. In this example, we make use of Descriptors, astrodata arithmetic functions, data sections, numpy 0-based arrays, and numpy statistics function mean().:

```
# Get the (EXTNAME,EXTVER)-keyed dictionary for the overscan section and
# the data section.
oversec_descriptor = ad.overscan_section().as_dict()
datasec_descriptor = ad.data_section().as_dict()

# Loop through the extensions.
for ext in ad['SCI']:
    extnamever = (ext.extname(),ext.extver())
    (x1, x2, y1, y2) = oversec_descriptor[extnamever]
    (dx1, dx2, dy1, dy2) = datasec_descriptor[extnamever]

    # Measure and subtract the overscan level
    mean_overscan = ad[extnamever].data[y1:y2,x1:x2].mean()
    ad[extnamever].sub(mean_overscan)
```

```
    # Trim the data to remove the overscan section and keep only
    # the data section.
    ad[extnamever].data = ad[extnamever].data[dy1:dy2,dx1:dx2]
```

## 7.8 Work on Data Cubes

```
from astrodata import AstroData
from numdisplay import display
from pylab import *

adcube = AstroData('gmosifu_cube.fits')
adcube.info()

# The pixel data is a 3-dimensional numpy array with wavelength is axis 0, and
# x,y positions in axis 2 and 1, respectively.  (In the FITS file, wavelength
# is in axis 3, and x, y are in axis 1 and 2, respectively.)
adcube.data.shape

# To sum along the wavelength axis
sum_image = adcube.data.sum(axis=0)
display(sum_image, zscale=True)

# To plot a 1-D representation of the wavelength axis at pixel position (7,30)
plot(adcube.data[:,29,6])
show()

# To plot the same thing using the wavelength values for the x axis of the plot
# one needs to use the WCS to calculate the pixel to wavelength conversion.
crval3 = adcube.get_key_value('CRVAL3')
cdelt3 = adcube.get_key_value('CDELT3')
spec_length = adcube.data[:,29,6].size
wavelength = crval3 + arange(spec_length)*cdelt3
plot(wavelength, adcube.data[:,29,6])
show()
```

## 7.9 Plot Data

In Python, the main tool to create plots is `matplotlib`. We have used it in the previous section on data cubes. Here we do not aimed at covering all of `matplotlib`; the reader should refer to that package's documentation. Rather we will give a few examples that might be of use for quick inspection of the data.

```
from astrodata import AstroData
from pylab import *

adimg = AstroData('N20110313S0188.fits')
adspec = AstroData('estgsS20080220S0078.fits')

# Line plot from image.  Row #1044.
line_index = 1043
line = adimg['SCI',2].data[line_index, :]
plot(line)
show()
```

```
# Column plot from image, averaging across 11 pixels around column #327.
col_index = 326
width = 5
col_section = adimg['SCI',2].data[:,col_index-width:col_index+width+1]
column = col_section.mean(axis=1)
plot(column)
show()

# Contour plot for section
galaxy = adimg['SCI',2].data[1045:1085,695:735]
contour(galaxy)
axis('equal')
show()

# Spectrum in pixel
plot(adspec['SCI',1].data)
show()

# Spectrum in wavelength (CRPIX1 = 1)
crpix1 = adspec['SCI',1].get_key_value('CRPIX1')
crval1 = adspec['SCI',1].get_key_value('CRVAL1')
cdelt1 = adspec['SCI',1].get_key_value('CDELT1')
length = adspec['SCI',1].get_key_value('NAXIS1')
wavelengths = crval1 + (arange(length)-crpix1+1)*cdelt1
plot(wavelengths, adspec['SCI',1].data)
show()
```

CHAPTER

EIGHT

# TABLE DATA

Astrodata does not provide any special wrappers for FITS Table Data. But since astrodata is built on top of pyfits, the standard pyfits table functions can be used. The reader should refer to the pyfits documentation for complete details. Here we show a few useful examples of basic usage.

## 8.1 Read from a FITS Table

A FITS table is stored in a MEF file as a BinTableHDU. The table data is retrieved from the AstroData object with the same .data attribute as for pixel extension, but for FITS tables .data returns a FITS_rec, which is a pyfits class. Here is how to get information out of a FITS table.:

```
from astrodata import AstroData

adspec = AstroData('estgsS20080220S0078.fits')
adspec.info()
# The first extension in that file is a FITS table with ''EXTNAME'' MDF, and ''EXTVER'' 1.
# MDF stands for "Mask Definition File".  In Gemini data, those are used in the data reduction
# to identify, to first order, where spectra fall on the detector.

# Let's get the table data out of the AstroData object
table = adspec['MDF'].data

# Get the column names with 'names' or more details with 'columns'
table.names
table.columns

# Get all the data for a column
x_ccd_values = table.field('x_ccd')
third_col = table.field(2)

# Print the table content
print table

# Print the first 2 rows
print table[:2]

# Select rows based on some criterion
select_table = table[table.field('y_ccd') > 2000.]
print select_table
```

## 8.2 Create a FITS Table

Creating a FITS table is mostly a matter of creating the columns, name and data. The name is a string, the data is stored in a `numpy` array.:

```
from astrodata import AstroData
import pyfits as pf
import numpy as np

# Create the input data
snr_id = np.array(['S001','S002','S003'])
feii = np.array([780.,78.,179.])
pabeta = np.array([740.,307.,220.])
ratio = pabeta/feii

# Create the columns
col1 = pf.Column(name='SNR_ID', format='4A', array=snr_id)
col2 = pf.Column(name='ratio', format='E', array=ratio)
col3 = pf.Column(name='feii', format='E', array=feii)
col4 = pf.Column(name='pabeta', format='E', array=pabeta)

# Assemble the columns
cols = pf.ColDefs([col1,col2,col3,col4])

# Create the table HDU
tablehdu = pf.new_table(cols)

# Create an AstroData object to contain the table
# and write to disk.
new_ad = AstroData(tablehdu)
new_ad.rename_ext('MYTABLE',1)
new_ad.write('mytable.fits')
```

A new FITS table can also be appended to an already existing AstroData object with the `.append()` function.

## 8.3 Operate on a FITS Table

The `pyfits` manual is the recommended source for a more complete documentation on working on FITS table with Python. Here are a few examples of what one can modify a FITS table.:

```
from astrodata import AstroData
import pyfits as pf
import numpy as np

# Let us first create tables to play with
snr_id = np.array(['S001','S002','S003'])
feii = np.array([780.,78.,179.])
pabeta = np.array([740.,307.,220.])
ratio = pabeta/feii
col1 = pf.Column(name='SNR_ID', format='4A', array=snr_id)
col2 = pf.Column(name='ratio', format='E', array=ratio)
col3 = pf.Column(name='feii', format='E', array=feii)
col4 = pf.Column(name='pabeta', format='E', array=pabeta)
cols_t1 = pf.ColDefs([col1,col3])
cols_t2 = pf.ColDefs([col1,col4])
cols_t3 = pf.ColDefs([col2])
```

```
table1 = pf.new_table(cols_t1)
table2 = pf.new_table(cols_t2)
table3 = pf.new_table(cols_t3)

# Merge tables
#   WARNING: The input tables must NOT share any common field names.
#      For example, table1 and table2 cannot be merged this way since they share col1.
merged_cols = table1.columns + table3.columns
merged_table = pf.new_table(merged_cols)
merged_table.columns.names  # or merged_table.data.names
print merged_table.data

# Add/Delete column
#   To "add" the 'pabeta' column from table2 to table1
table1.columns.add_col(table2.columns[table2.columns.names.index('pabeta')])
table1 = pf.new_table(table1.columns)
table1.columns.names
print table1.data

#   To "delete" the 'pabeta' column from this new table1
table1.columns.del_col('pabeta')
table1 = pf.new_table(table1.columns)
table1.columns.names
print table1.data

# Insert column
#   To insert a column, one has to extract the columns
#   and reorganize them into a new table.
#   Insert the first, and only column, in table3, between the first and second
#   column in table1
t1_col1 = table1.columns[0]
t1_col2 = table1.columns[1]
t3_col1 = table3.columns[0]
table1 = pf.new_table([t1_col1,t3_col1,t1_col2])
table1.columns.names
print table1.data

# Change the name of a column
#   WARNING: There is method .change_name but it does not seem to be
#            working properly.
table1.columns[table1.columns.names.index('feii')].name='ironII'
table1 = pf.new_table(table1.columns)

# Add/Delete row
#  Adding and deleting rows requires the creation of a new table
#  of the correct, new size.
#
#  Add 2 new entries to table2.  Only 'SNR_ID' and 'pabeta' will be
#  added as those are the columns already present in table2.
nb_new_entries = 2
new_entries = {'SNR_ID': ['S004','S005'],
               'ratio' : [1.12, 0.72],
               'feii'  : [77., 87.],
               'pabeta': [69., 122.]
               }
nrowst2 = table2.data.shape[0]
large_table = pf.new_table(table2.columns, nrows=nrowst2+nb_new_entries)
for name in table2.columns.names:
```

```
    large_table.data.field(name)[nrowst2:]=new_entries[name]
table2 = large_table

# Delete the last 2 entries from table2
nb_bad_entries = 2
nrowst2 = table2.data.shape[0]
small_table = pf.new_table(table2.columns, nrows=nrowst2-nb_bad_entries)
for name in table2.columns.names:
    small_table.data.field(name)[:]=table2.data.field(name)[:-nb_bad_entries]
table2 = small_table

# Change the 'pabeta' value for source S002 in table2
rowindex = np.where(table2.data.field('SNR_ID') == 'S002')[0][0]
table2.data.field('pabeta')[rowindex] = 888.
```

CHAPTER

NINE

# LOG UTILITY

## 9.1 The Astrodata Log Utility

Astrodata uses a logging utility based on the Python logging facility. astrodata/adutils/logutils.py creates logfile, default name reduce.log

**config() method** mode: standard, stream, null, debug consolve_lvl: controls the console logging level file_name: logfile name (default=reduce.log) stomp: clobber

get_logger()

update_indent(): control indenting during recipe/primitive execution.

logger mode standard: default console-> stdinfo, default to file-> fullinfo

## 9.2 Writing to Log

Using the logging facility involves *getting* the logger, *configuring* the logger,

```
log.<loglevel>(<message_to_log>)
```

default name: reduce.log

```
from astrodata.adutils import logutils

log = logutils.get_logger(__name__)

??
log = self.log
self.log.stdinfo()
```

In primitives, call logger "once at the top"

## 9.3 Log Levels

Several log levels are supported, some are directly from the Python logging facility, others are defined in Astrodata. Here are definitions of the log levels and usage examples.

**critical** A serious error, indicating that the program itself may be unable to continue running. For example:

```
try:
    ...
except:
    # Log the message from the exception
    log.critical(repr(sys.exc_info()[1]))

# or simply
...
log.critical("Something really bad happened.  Exiting now.")
```

**error** Due to a serious problem, the software has not been able to perform some function. The error does not necessarily prevent the program from continuing.

```
log.error('An error occurred while trying to calculate the \
           nbiascontam, using default value = 4')
```

**warning** An indication that something unexpected happened, or indicative of some problem in the near future. The software is still working as expected, but might be using some default or recovery settings.

```
log.warning("A [DQ,%d] extension already exists in %s" %
            (extver, ad.filename))
```

**status** Start and end processing information, number of files, name of the input or output files. In other words, "What's happening? What's being processed?"

```
log.status("List for stack id=%s" % sid)
if len(stacklist) > 0:
    for f in stacklist:
        log.status("    %s" % os.path.basename(f))
else:
    log.status("No datasets in list")
```

**stdinfo** Scientific information like seeing measurements, statistics, etc. or what is scientifically being done to the data. This is information that an astronomer might want to see displayed on the screen.

```
log.stdinfo("Adding the read noise component of the variance")
log.stdinfo("RA: %.2f +- %.2f    Dec: %.2f +- %.2f   arcsec" %
             (ra_mean, ra_sigma, dec_mean, dec_sigma))
```

**info** Confirmation that things are working as expected. The information here is more programmatical than scientific.

**fullinfo** Detailed information on the processing, like input parameters, header changes. Useful information for a log file but not necessary for standard output (screen output).

```
log.fullinfo("Tiling extensions together to get statistics from CCD2")
log.fullinfo("Using data section [%i:%i,%i:%i] from CCD2 for statistics" %
              (xborder,sci_data.shape[1]-xborder,
               yborder,sci_data.shape[0]-yborder))
```

**debug** Very detailed engineering information for used in debugging. For example:

```
...
log.debug("SplotETI __init__")
...
log.debug("SplotETI.execute()")
...
log.debug("SplotETI.run()")
...
log.debug("SplotETI.recover()")
...
```

CHAPTER

TEN

# DATASET VALIDATION

## 10.1 File Structure Validation

CHAPTER

ELEVEN

# ADVANCED TOPICS

## 11.1 Creating a New Configuration Module

## 11.2 Adding Support for a New Instrument

CHAPTER

TWELVE

# OTHER TOPICS

CHAPTER

THIRTEEN

# LIST OF STANDARD DESCRIPTORS

(this should be an appendix)

List the descriptors that we have and give a definition.

CHAPTER

FOURTEEN

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*