Kapteyn Documentation

Release 3.0.0-beta

J.P. Terlouw, M.G.R. Vogelaar, M.A. Breddels

March 22, 2016

CHAPTER 1

Obtaining and using the package

1.1 Introduction

The Kapteyn Package is a collection of Python modules and applications developed by the computer group of the Kapteyn Astronomical Institute, University of Groningen, The Netherlands. The purpose of the package is to provide tools for the development of astronomical applications with Python.

The package is suitable for both inexperienced and experienced users and developers and documentation is provided for both groups. The documentation also provides in-depth chapters about celestial transformations and spectral translations.

Some of the package's features:

- The handling of spatial and spectral coordinates, WCS projections and transformations between different sky systems. Spectral translations (e.g., between frequencies and velocities) are supported and also mixed coordinates. (Modules wcs and celestial)
- Versatile tools for writing small and dedicated applications for the inspection of FITS headers, the extraction and display of (FITS) data, interactive inspection of this data (color editing) and for the creation of plots with world coordinate information. (Module maputils) As one example, a gallery of all-sky plots is provided.
- A class for the efficient reading, writing and manipulating simple table-like structures in text files. (Module tabarray)
- Utilities for use with matplotlib such as obtaining coordinate information from plots, interactively modifiable colormaps and timer events (module mplutil); tools for parsing and interpreting coordinate information entered by the user (module positions).

1.1.1 Overview

The following modules are included:

- `wcs`, a binary module which handles spatial and spectral coordinates and provides WCS projections and transformations between different sky systems. Spectral translations (e.g., between frequencies and velocities) are supported and also mixed coordinates.
- `celestial`, containing NumPy-based functions for creating matrices for transformation between different celestial systems. Also a number of other utility functions are included.
- `wcsgrat`, for calculating parameters for WCS graticules. It does not require a plot package.
- `maputils`. Provides methods for reading FITS files. It can extract 2-dim image data from data sets with three or more axes. A class is added which prepares FITS data to plot itself as an image with Matplotlib.

- `positions`, enabling a user/programmer to specify positions in either pixel- or world coordinates.
- `rulers`, defining a class for drawing rulers.
- `shapes`, defining a class for interactively drawing shapes that define an area in an image. For each area a number of properties of the data is calculated. This module can duplicate a shape in different images using transformations to world coordinates. This enables one for instance to compare flux in two images with different WCS systems.
- `mplutil`, utilities for use with matplotlib. Classes AxesCallback, providing a more powerful mechanism for handling events from LocationEvent and derived classes than matplotlib provides itself; TimeCallback for handling timer events and VariableColormap which implements a matplotlib Colormap subclass with special methods that allow the colormap to be modified.
- `kmpfit`, providing a class and a function for non-linear least-squares fitting, using the Levenberg-Marquardt technique. It is based on the implementation in C of Craig Markwardt's MPFIT.
- `tabarray`, providing a class for the efficient reading, writing and manipulating simple table-like structures in text files.

1.1.2 Prerequisites

To install the Kapteyn Package, at least Python [1] 2.4 and NumPy [2] (both with header files) are required. For using it, the availability of PyFITS [3] or Astropy [4] and matplotlib [5] is recommended. Windows users may also need to install Readline [6] or an equivalent package.

Mark Calabretta's WCSLIB [7] does not need to be installed separately anymore. Its code is now included in the Kapteyn Package under the GNU Lesser General Public License.

1.1.3 Download

The Kapteyn Package and the example scripts can be downloaded via links on the package's homepage: http://www.astro.rug.nl/software/kapteyn/

1.1.4 Installing

First unpack the downloaded .tar.gz or .zip file and go to the resulting directory. Then one of the following options can be chosen:

1. Install into your Python system (you usually need root permission for this):

```
python setup.py install
```

2. If you prefer not to modify your Python installation, you can create a directory under which to install the module e.g., *mydir*. Then install as follows:

```
python setup.py install --install-lib mydir
```

   To use the package you then need to include *mydir* in your PYTHONPATH.

[1] http://www.python.org/
[2] http://numpy.scipy.org/
[3] http://www.stsci.edu/resources/software_hardware/pyfits
[4] http://www.astropy.org/
[5] http://matplotlib.sourceforge.net/
[6] http://newcenturycomputers.net/projects/readline.html
[7] http://www.atnf.csiro.au/people/mcalabre/WCS/

3. If you want to use this package only for GIPSY, you can install it as follows:

```
python setup.py install --install-lib $gip_exe
```

   The GIPSY installation procedure normally does this automatically, so usually this will not be necessary.

Windows installer

An experimental installer for Microsoft Windows (together with other packages that the Kapteyn Package depends on) is also available. Currently only for Python 2.6 on 32-bit systems. http://www.astro.rug.nl/software/kapteyn_windows/

Scisoft problem

If you have Scisoft installed on your computer, it may interfere with the installation of the Kapteyn Package. To install it properly, disable the setup of Scisoft in your startup file (e.g. ~/.cshrc, .profile) by commenting it out.

Mac OS X Compiler problem

There is a known problem with Apple's llvm-gcc-4.2 compiler. This compiler is known to crash with an internal compiler error (Segmentation fault: 11) when WCSLIB routine wcserr.c is compiled. For this reason, setup.py tries to detect this compiler and use the clang compiler instead. If compilation still fails, one could try to prefix a shell variable definition to the install command like this:

```
export CC=CLANG; python setup.py install ...
```

1.1.5 Contact

The authors can be reached at:

Kapteyn Astronomical Institute
Postbus 800
NL-9700 AV Groningen
The Netherlands
Telephone: +31 50 363 4073
E-mail: gipsy@astro.rug.nl

1.2 How to start

1.2.1 Introduction

This chapter is intended to be a guide on how to use the modules from the Kapteyn Package for your own astronomical software. The Kapteyn Package provides building blocks for software that has a focus on the use of world coordinates and/or plotting image data.

To get an overview of what is possible, have a look at Tutorial maputils module which contains many examples of world coordinate annotations and plots of astronomical data. It can be a good starting point to use the source code in the example scripts to process your own data by making only small changes to the code.

If you are only interested in coordinate transformations, then the Tutorial wcs module is a good starting point.

1.2.2 Which module and documents to use?

You want:	You need:
For a set of world coordinates, I want to transform these to another projection system. I have a FITS header.	`wcs`, Tutorial wcs module
I want to transform world coordinates between sky- and reference systems	`wcs`, Tutorial wcs module
I want a parser to convert a string with position information to pixel- and/or world coordinates.	`positions`
I want to transform image data in a FITS file from one projection system to another	`maputils`, Tutorial maputils module
I want to build a utility that converts a header with a PC or CD matrix to a 'classic' header with CRPIX, CRVAL, CDELT and CROTA	`maputils`, Tutorial maputils module
I want to create a utility that can display a mosaic of image data	`maputils`, Tutorial maputils module
I want to plot an all sky map with graticules	`maputils`, Tutorial maputils module
I want to calculate flux in a set of images	`maputils`, `shapes`, Tutorial maputils module
I want to create a simple FITS file viewer with user interaction for the colors etc.	`maputils`, Tutorial maputils module
I want to read a large data file very fast	`tabarray`, Tutorial tabarray module
Given a year, month and day number, I want the corresponding Julian date	`celestial`, Tutorial wcs module
I want to know the obliquity of the ecliptic at a Julian date?	`celestial`, Tutorial wcs module, Background information module celestial
I want to convert my spectral axis from frequency to relativistic velocity	`wcs`, Tutorial maputils module, Background information spectral translations

1.2.3 Functionality of the modules in the Kapteyn Package

Wcs

- Given a FITS header or a Python dictionary with header information about a World Coordinate System (WCS), transform between pixel- and world coordinates.
- Different coordinate representations are possible (tuple of scalars, NumPy array etc.)
- Transformations between sky and reference systems.
- Epoch transformations
- Support for 'alternate' headers (a header can have more than one description of a WCS)
- Support for mixed coordinate transformations (i.e. pixel- and world coordinates at input are mixed).
- Spectral coordinate translations, e.g. convert a frequency axis to an optical velocity axis.

Celestial

- Coordinate transformations between sky and reference systems. Also available via module `wcs`

- Epoch transformations. Also available via module `wcs`
- Many utility functions e.g. to convert epochs, to parse strings that define sky- and reference systems, calculate Julian dates, precession angles etc.

Wcsgrat

- Most of the functionality in this module is provided via user friendly methods in module `maputils`.
- Calculate grid lines showing constant latitude as function of varying longitude or vice versa.
- Methods to set the properties of various plot elements like tick marks, tick labels and axis labels.
- Methods to calculate positions of labels inside a plot (e.g. for all sky plots).

Maputils

- Easy to combine with Matplotlib
- Convenience methods for methods of modules `wcs`, `celestial`, `wcsgrat`
- Overlays of different graticules (each representing a different sky system),
- Plots of data slices from a data set with more than two axes (e.g. a FITS file with channel maps from a radio interferometer observation)
- Plots with a spectral axis with a 'spectral translation' (e.g. Frequency to Radio velocity)
- Rulers with distances in world coordinates, corrected for projections.
- Plots for data that cover the entire sky (allsky plot)
- Mosaics of multiple images (e.g. HI channel maps)
- A simple movie loop program to view 'channel' maps.
- Interactive colormap selection and modification.

Positions

- Convert strings to positions in pixel- and world coordinates

Rulers

- Plot a straight line with markers at constant distance in world coordinates. Its functionality is available in module `maputils`

Shapes

- Advanced plotting with user interaction. A user defines a shape (polygon, ellipse, circle, rectangle, spline) in an image and the shape propagates (in world coordinates) to other images. A shape object keeps track of its area (in pixels) and the sum of the pixels within the shape. From these a flux can be calculated.

Tabarray

- Fast I/O for data in ASCII files on disk.

Mplutil

- Various advanced utilities for event handling in Matplotlib. Most of its functionality is used in module `maputils`.

1.3 License

1.3.1 Kapteyn Package

The Kapteyn Package is provided under the following license:

Please cite the Kapteyn Package

If you have used the Kapteyn Package in the preparation of a publication, please **cite**. We need these citations to justify time and resources spent on the software.

You may cite it as follows (BibTeX format):

1.3.2 SciPy modules

To the modules included from the SciPy package, the following license applies:

1.3.3 WCSLIB

WCSLIB, which is included in the Kapteyn Package's distribution, is provided under the following license:

1.3.4 MPFIT

MPFIT's implementation in C, of which a modified version is included, is provided under the following license:

1.4 Release notes

CHAPTER 2

Tutorials

2.1 Tutorial wcs module

2.1.1 Introduction

This tutorial aims at starters. Experienced users find relevant but compact documentation in the module documentation. In this tutorial we address different practical situations where we need to convert between pixel- and world coordinates. Many examples are working scripts, others are very useful to try in an interactive Python session.

`wcs` is the core of the Kapteyn Package. An important feature of that package is that it provides a world coordinate system which is easy to incorporate in your own (Python) environment and `wcs` provides the basic methods to do this. Together with module `celestial` it allows a user to transform between pixel coordinates and world coordinates for a set of supported projections and sky systems. Module `celestial` provides a rotation matrix for sky transformations and is more or less embedded in `wcs`, so (for standard work) there is no need to import it separately.

Module `wcs` module has a number of important features:

- Flexible I/O of coordinates
- Support for spatial and spectral data
- Support for 'mixed' coordinates
- Support for conversions between different celestial systems
- Objects have useful attributes
- Easy to combine with other software written in Python

2.1.2 Coordinate representations

One coordinate axis

For experiments and debug sessions, module `wcs` allows for very simple and flexible input and output of coordinates. This module interfaces with Mark Calabretta's WCSLIB and is, because of the flexible I/O, a valuable tool to test this well known library.

Main goal of module `wcs` is to enable transformations between pixel coordinates and world coordinates The pixel coordinates are defined by the FITS standard. The transformation is defined by meta data which are usually found in FITS headers. So it may be obvious that FITS files play an important role in the use of Module `wcs`.

However, FITS data processed by `wcs` can also be FITS keywords that are stored in a Python dictionary. This invites to experiment with WCSLIB even more because one can create a (minimal) FITS header from scratch. In an attempt to create the most simple use of `wcs` we started to write a minimal FITS header. It defines only one axis. The minimal requirement for FITS keywords are CTYPE, CRVAL, CRPIX and CDELT. A description of these keywords can be found in The FITS standard.

We entered an axis type in *CTYPE1* that WCSLIB does not recognize as a known type. With this trick we force the system to do a linear transformation. It shows that you have to be careful with values for CTYPE because you will not be warned when a CTYPE is not recognized.

For the conversions between pixel coordinates and world coordinates we defined methods in a class which we called the `wcs.Projection` class. An object of this class is created using the header of the FITS file for which we want WCS transformations. It accepts also a user defined Python dictionary with FITS keywords and values. We use this last option in this tutorial to be more flexible when we want to apply changes to the header.

The methods for single axes are called `wcs.Projection.toworld1d()` and `wcs.Projection.topixel1d()`. FITS defines CRVAL as the world coordinate that corresponds to the pixel value in CRPIX. Let's check this with the most basic example we could think of:

```
#!/usr/bin/env python
from kapteyn import wcs
header = { 'NAXIS'  : 1,
           'CTYPE1' : 'PARAM',
           'CRVAL1' : 5,
           'CRPIX1' : 10,
           'CDELT1' : 1
         }
proj = wcs.Projection(header)
print proj.toworld1d(10)

# Output:
# 5.0
```

Indeed, at pixel coordinate 10 (=CRPIX), the world coordinate is 5 (=CRVAL). If we want to know which pixel coordinate corresponds to world coordinate 5, then we use `proj.topixel1d(5)` to get the answer (which is the value of CRPIX: 10). Note that we forced the system to apply linear transformations only.

In many of the examples that we present in this tutorial we included a so called *closure* test. This is a test which uses the result of a transformation to test the inverse transformation which should result into the original value. Sometimes the result is not exactly what you expect because we work with a limited number precision. A simple closure test is:

```
proj = wcs.Projection(header)
w = proj.toworld1d(10)
p = proj.topixel1d(w)
print "CRPIX: ", p

# Output:
# CRPIX:  10.0
```

Coordinate transformations are often done in bulk, so of course the transformation methods accept more than one coordinate to convert. They can be represented as a Python list, a Python tuple or a NumPy array. The representation of the output is the same as that of the input coordinates. The output of the next statements therefore is not a surprise:

```
#!/usr/bin/env python
from kapteyn import wcs
import numpy

header = { 'NAXIS'  : 1,
           'CTYPE1' : 'PARAM',
           'CRVAL1' : 5,
           'CRPIX1' : 10,
           'CDELT1' : 1
```

```
          }

proj = wcs.Projection(header)

w1 = proj.toworld1d( range(9,12) )
w2 = proj.toworld1d( [9,10,11] )
w3 = proj.toworld1d( (9,10,11) )
w4 = proj.toworld1d( numpy.array([9,10,11]) )
print w1, type(w1)
print w2, type(w2)
print w3, type(w3)
print w4, type(w4)
closure = proj.topixel1d(w4)     # Closure test
print closure, type(closure)

# Output:
# [4.0, 5.0, 6.0] <type 'list'>
# [4.0, 5.0, 6.0] <type 'list'>
# (4.0, 5.0, 6.0) <type 'tuple'>
# [ 4.  5.  6.] <type 'numpy.ndarray'>
# [ 9.  10. 11.] <type 'numpy.ndarray'>
```

The first two sequences are lists. The third is a tuple and the last is a NumPy array. The pixel coordinates 9, 10 and 11 should give values in the neighbourhood of *CRVAL1* and the step size is 1 (*CDELT1=1*), in arbitrary units.

Note: An advantage of NumPy arrays is that you can use them in mathematical expressions to process the array content. For example: assume you have a sequence of velocities in a numpy array V but want to express the numbers in km/s, then change the content with expression: V /= 1000

For representation purposes we often want to print a pixel coordinate and the corresponding world coordinate on one line. Then we often use Pythons built-in function *zip* to combine two sequences to avoid a call to transformation methods in the print loop:

```
p = range(5,15)
w = proj.toworld1d(p)
for pix,wor in zip(p,w):
   print "%d: %f" % (pix,wor)

# Output:
# 9: 4.000000
# 10: 5.000000
# 11: 6.000000
```

Note: Class wcs has an attribute called **debug**. If you set its value to *True* then you get debug information from WCSLIB showing what has been correctly parsed from the given header data. Use it as follows:

```
wcs.debug = True
proj = wcs.Projection(header)
```

Next we apply the procedures described above to a real example where we created an artificial header with FITS data. The header describes a single axis of spectral type. Units are standard FITS units and are given in keyword *CUNIT1*. The example shows that we can access the keywords from the artificial header (or a real FITS header) directly and use their values for example to find the length of the axis in pixels, or to find the units of the world coordinates of that axis:

```
#!/usr/bin/env python
from kapteyn import wcs
header  = { 'NAXIS'  : 1,
            'NAXIS1' : 64,
            'CTYPE1' : 'FREQ',
            'CRVAL1' : 1.37835117405e9,
            'CRPIX1' : 32,
            'CUNIT1' : 'Hz',
            'CDELT1' : 9.765625e4
         }
proj = wcs.Projection(header)
n = header['NAXIS1']                 # Get the length of the spectral axis
p = range(1, n+1)                    # Set pixel range accordingly
w = proj.toworld1d(p)                # Do the transformation
print "Pixel  %s (%s)" % (header['CTYPE1'],header['CUNIT1'])
for pix,frq in zip(p,w):
   print "%5d: %f" % (pix,frq)

# Output:
# Pixel  FREQ (Hz)
#    1: 1375323830.300000
#    2: 1375421486.550000
#    3: 1375519142.800000
#    4: 1375616799.050000
#    5: 1375714455.300000
```

In the example we wanted to make a table with pixel coordinates and the corresponding world coordinates. According to the header there are 64 pixels (*NAXIS1*) along the axis so the first pixel coordinate is 1 and the last is 64. The axis represents frequencies. A start frequency is given by *CRVAL1* and a step size is given by *CDELT1*. Note that the coordinate transformation is linear.

Generic methods *toworld()* and *topixel()*

The methods `wcs.Projection.toworld1d()` and `wcs.Projection.topixel1d()` are special versions of the more general methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()`. These methods can be used to convert pixel data for more than one axis at the same time which is necessary for coupled axes, for example in spatial maps where longitude and latitude are not independent axes.

These general methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()` accept the same sequences as the '1d' versions. The reason that we introduced the '1d' versions is that for non-experienced Python programmers it usually is confusing that in the one dimensional case the general methods only accept tuples and not scalars and that a tuple with one element (for example 10) needs to be written as *(10,)*.

If you want to replace method toworld1d() by topixel1d() in the first example, then the relevant lines become:

```
>>> p = proj.toworld( (10,) )
>>> (5.0,)
```

for one scalar and for a list of values:

```
>>> p = proj.toworld( (range(9,12),) )
>>> ([4.0, 5.0, 6.0],)
```

If you want to extract the scalar or the list from the tuple, use element 0 of the tuple.

```
>>> p = proj.toworld( (range(9,12),) )
>>> print p[0]
>>> [4.0, 5.0, 6.0]
```

Two coordinate axes

As described in the previous section we use `wcs.Projection.toworld()` and `wcs.Projection.topixel()` if the number of axes in our data is more than 1. The input and output tuples for projection objects with two coordinate axes consist of two elements. The first element corresponds to the first axis in the projection object and the second element to the second axis. The following Python code constructs an artificial header which describes the world coordinate system of two spatial axes. Then we want to find the world coordinates of the reference pixels (*CRPIX1*, *CRPIX2*) and expect the reference values (*CRVAL1*, *CRVAL2*) as output tuple:

```
#!/usr/bin/env python
from kapteyn import wcs
header   = { 'NAXIS'  : 2,
             'NAXIS1' : 5,
             'CTYPE1' : 'RA---NCP',
             'CRVAL1' : 45,
             'CRPIX1' : 5,
             'CUNIT1' : 'deg',
             'CDELT1' : -0.01,
             'NAXIS2' : 10,
             'CTYPE2' : 'DEC--NCP',
             'CRVAL2' : 30,
             'CRPIX2' : 5,
             'CUNIT2' : 'deg',
             'CDELT2' : +0.01,
          }
proj = wcs.Projection(header)
pixel = (5,5)
world = proj.toworld(pixel)
print world

# Output:
# (45.0, 30.0)
```

Comments about the composed header: the header is composed from scratch. but it could very well have been copied from an existing FITS header. In either case you should verify items **CUNITn** and **CTYPEn** because they are are important. In section 2.1.1 of *[Ref1]* we read that in WCSLIB:

Note: *any CTYPEi not covered by convention and agreement shall be taken to be linear*.

The CTYPE consists of a coordinate type (max 4 characters) followed by '-' followed by a three character code that represents the algorithm to calculate the world coordinates ('ABCD-XYZ'). Shorter coordinate types are padded with the '-' character, shorter algorithm codes are padded on the right with blanks ('RA—NCP', 'RA—UV_ '). So if we were sloppy and wrote RA–NCP and DEC-NCP then WCSLIB assigns a linear conversion algorithm. It does not complain, but you get unexpected results. If your CTYPEs are correct but the units are not standard and are not recognized by WCSLIB, then you get an Python exception after you try to create the Projection object. For example, if you specified CUNIT1='Degree' then the error message displayed by the exception is: *"Invalid coordinate transformation parameters"*.

If you want to be sure that WCSLIB recognizes your coordinate type and unit, you can print the Projection attributes `wcs.Projection.types` and `wcs.Projection.units` as in the example below. Unrecognized types are returned as *None*.

```
>>> proj = wcs.Projection(header)
>>> print "WCS units: ",proj.units
    WCS units:  ('deg', 'deg')
```

```
>>> print "WCS type: ",proj.types
    WCS type:  ('longitude', 'latitude')
```

With the same variable *header* as in the previous script we demonstrate that each element in the coordinate tuple can be a list of scalars. Let's convert pixel positions (3,3), (4,4), ..., (7,7) etc. to their corresponding world coordinates:

```
proj = wcs.Projection(header)
x = range(3,8)
y = range(3,8)
pixel = (x,y)
world = proj.toworld(pixel)
print world

# Output:
# ([45.023089356221305, 45.011545841750113, 45.0, 44.988451831142257, 44.97690133535837],
#  [29.979985885372404, 29.989996472289789, 30.0, 30.009996474046854, 30.019985899953429])
```

The output is a tuple with *two* elements. Each element is a list. The first list contains the longitude coordinates for input pixel coordinates (3,3), (4,4) etc. The second list contains the latitude coordinates for the input pixel coordinates (3,3), (4,4) etc.

Note: Note that longitude and latitude are not independent. You need always two pixel coordinates (x,y) to get a world coordinate pair (RA,DEC).

Here input and output coordinates for the methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()` are tuples. The dimension of the tuple corresponds to the number of axes in the Projection object, and each element in the tuple can be a list of scalars. In some situations it is more intuitive to start with a list of 2 dimensional positions. The `wcs` module allows for this type of input. You can get the same coordinate output as the previous script if you replace the body by:

```
proj = wcs.Projection(header)
pixels = [(3,3), (4,4), (5,5), (6,6), (7,7)]
world = proj.toworld(pixels)
print world

# Output:
# [(45.023089356221305, 29.979985885372404), (45.011545841750113, 29.989996472289789), (45.0, 30.0),
# (44.988451831142257, 30.009996474046854), (44.97690133535837, 30.019985899953429)]
```

Note that the representation of the output differs from the previous script because the representation of the input differs, i.e.: a list with tuples. The dimension of the tuples being the number of axes in your projection object.

Note: The coordinate representation in methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()` of the output is the same as that of the input.

Mixed transformations (pixel- and world coordinates) using method `wcs.Projection.mixed()`

We describe the mixed() method in some detail in the section about data sets with three or more axes. Here we show how to use the method in a simple case. Suppose you want to mark data in a plot at constant declination in pixels (i.e. parallel to the x-axis of the plot) but with equal steps in Right Ascension, then you need method `wcs.Projection.mixed()`:

```python
#!/usr/bin/env python
from kapteyn import wcs
import numpy
header  = { 'NAXIS'  : 2,
            'NAXIS1' : 5,
            'CTYPE1' : 'RA---TAN',
            'CRVAL1' : 45,
            'CRPIX1' : 5,
            'CUNIT1' : 'deg',
            'CDELT1' : -0.01,
            'NAXIS2' : 10,
            'CTYPE2' : 'DEC--TAN',
            'CRVAL2' : 30,
            'CRPIX2' : 10,
            'CUNIT2' : 'deg',
            'CDELT2' : +0.01,
         }
proj = wcs.Projection(header)
# 1 pixel and 1 world coordinate pair
pixel_in = (numpy.nan, 10)
world_in = (45.0, numpy.nan)
world_out, pixel_out = proj.mixed(world_in, pixel_in)
print world_out
print pixel_out

# Output:
# (45.0, 30.0)
# (5.0, 10.0)

# A loop over a number of Right Ascensions at constant Declination
for ra in range(44, 47):
   world_in = (ra,numpy.nan)
   world_out, pixel_out = proj.mixed(world_in, pixel_in)
   print "World: ", world_out, "Pixel: ", pixel_out

# Output:
# World:  (44.0, 29.99622120337045) Pixel:  (91.61133499750801, 10.000000000096229)
# World:  (45.0, 30.0) Pixel:  (5.0, 10.0)
# World:  (46.0, 29.99622120337045) Pixel:  (-81.61133499750801, 10.000000000096248)
```

First we have a pixel position of which the x coordinate is set to *unknown*. We use a special value for this: *numpy.nan* which is the representation of NumPy's Not A Number. The y coordinate is set to 10. For the `wcs.Projection.mixed()`, we need to specify the *unknown* values in the pixel position with a world coordinate. In the example we entered 45.0 (deg). The mixed() method returns two tuples. One for the pixel position and one for the position in world coordinates. The *unknown* values are calculated in an iterative process. The second part of the example is a loop over a number of world coordinates in Right Ascension, and a constant pixel coordinate in the y-direction (i.e. 10). The output (as listed as comment in the code) shows two things that need to be addressed. First we notice that the output pixel is not exactly 10. This is related to finite precision of numbers when a solution is calculated in an iterative way. The second observation is more important: the Declination varies while the y coordinate in pixels is constant. But this is exactly what we expect for spatial data when a projection is involved.

A note about efficiency:

Note: The transformation routines accept sequences of coordinates. Calculations with sequences are more efficient than repetitive calls in a loop.

So in our example it is more efficient to avoid the loop over the right ascensions. This can be done by creating an input tuple with two lists. The output is the same as in the example above, but the representation is different. As we stated earlier, the representation of the output is the same as the representation of the input (a tuple with two lists):

```
# As example above but without a loop
ra = range(44, 47)
dec = [numpy.nan]*len(ra)  # NumPy trick to repeat elements in a list.
world_in = (ra, dec)
x = [numpy.nan]*len(ra)
y = [10]*len(ra)
pixel_in = (x, y)
world_out, pixel_out = proj.mixed(world_in, pixel_in)
print world_out
print pixel_out

# Output:
# ([44.0, 45.0, 46.0], [29.99622120337045, 30.0, 29.99622120337045])
# ([91.61133499750801, 5.0, -81.61133499750801], [10.000000000096229, 10.0, 10.000000000096248])
```

Three or more coordinate axes

In this section we discuss method `wcs.Projection.sub()` which allows us to define coordinate transformations for positions with less dimensions than the dimension of the data structure. In practice we encounter many astronomical measurements based on three or more independent axes. Well known examples are of course the data sets from radio interferometers. Usually these are spatial maps observed at different frequencies and sometimes as function of Stokes parameters (polarization). If we are only interested in spatial maps and don't bother about the other axes, we can create a Projection object with only the relevant axes. This is done with the `wcs.Projection.sub()` method from the Projection class.

map = proj.sub(axes=None, nsub=None)

The method has two parameters. You can specify parameter *nsub* which sets the first *nsub* axes from the original Projection object to the actual axes. Or you can use the other parameter axes which is a tuple or a list with axis numbers. Axis numbers in WCSLIB follow the FITS standard so they start with 1. The order in the sequence is important. The axis description sequence in a FITS file is not bound to rules and luckily WCSLIB accepts permuted axis number sequences. This can be illustrated with the next example. First we show the code and then explain the output:

```
#!/usr/bin/env python
from kapteyn import wcs
import numpy
header  = { 'NAXIS'  : 3,
            # First spatial axis
            'NAXIS1' : 5,
            'CTYPE1' : 'RA---TAN',
            'CRVAL1' : 45,
            'CRPIX1' : 5,
            'CUNIT1' : 'deg',
            'CDELT1' : -0.01,
            # A dummy axis
            'NAXIS2' : 5,
            'CTYPE2' : 'PARAM',
            'CRVAL2' : 444,
            'CRPIX2' : 99,
            'CDELT2' : 1.0,
            'CUNIT2' : 'wprf',
            # Second spatial axis
            'NAXIS3' : 0,
            'CTYPE3' : 'DEC--TAN',
            'CRVAL3' : 30,
            'CRPIX3' : 10,
            'CUNIT3' : 'deg',
            'CDELT3' : +0.01
         }
proj = wcs.Projection(header)
map = proj.sub( [1,3] )
pixel = (header['CRPIX1'], header['CRPIX3'])
world = map.toworld(pixel)
print world

# Output:
# (45.0, 30.0)

map = proj.sub( [3,1] )
pixel = (header['CRPIX3'], header['CRPIX1'])
world = map.toworld(pixel)
print world

# Output:
# (30.0, 45.0)

line  = proj.sub( 2 )
crpix = header['CRPIX2']
pixels = range(crpix-5,crpix+6)
world = line.toworld1d(pixels)
print world

# Output:
# [439.0, 440.0, 441.0, 442.0, 443.0, 444.0, 445.0, 446.0, 447.0, 448.0, 449.0]
```

We created a header representing a spatial map as function of some parameter along the CTYPE2='PARAM' axis. This axis is not recognized by WCSLIB and a linear transformation is applied. Also special is that the spatial axes do not have conventional numbers. First we want to set up a transformation of pixel (x,y) to (R.A., Dec) for the pixel values in (CRPIX1, CRPIX3) -which should transform to (CRVAL1, CRVAL3)-. Then we reverse the spatial axis sequence to set up a transformation from (y,x) to (Dec, R.A.). Finally we want a transformation only for the PARAM axis. Its axis number is 2. With the output we show that for this axis indeed the transformation between pixels and world coordinates is a linear. transformation.

The axis sequence in the `wcs.Projection.sub()` method sets the axis order with parameter *axes*. It sets in fact the order of the coordinates in the transformation methods `wcs.Projection.toworld()`, `wcs.Projection.topixel()` and `wcs.Projection.mixed()`. Parameter *axes* is either a single integer or a list/tuple of integers e.g. sub(2) vs. sub([3,1]).

2.1.3 NumPy arrays and matrices

NumPy matrices

In many Python applications programmers use NumPy arrays and matrices because it is easy to manipulate them. First let's explore what can be done with a NumPy matrix as coordinate representation. A NumPy matrix is a rank 2 array with special properties. The first list in the numpy.matrix() constructor in the next example is the first row in the matrix and the second list is the second row. The first row contains the x coordinate of the pixels and the second row contains the y coordinates. In the next script we want to convert pixel positions (4,5), (5,5) and (6,5) to world coordinates. So the first list in the matrix constructor are the x coordinates [4,5,6] and the second are the y coordinates [5,5,5]. We convert these with:

```
proj = wcs.Projection(header)
pixel = numpy.matrix( [[4,5,6],[5,5,5]] )
world = proj.toworld(pixel)
print world
# Output:
# [[ 45.01154701  45.          44.98845299]
# [ 29.99999798  30.          29.99999798]]

pixel = proj.topixel(world)
print pixel

# Output:
# [[ 4.00000001  5.          5.99999999]
# [ 5.          5.          5.        ]]
```

The output is what we expected. It is a NumPy matrix with two rows. The first row contains the longitudes and the second the latitudes. The numbers seem ok (three RA's at almost constant declination). We added a closure test by using the output world coordinates as input for the `wcs.Projection.topixel()` method. As you can see, the closure test returns the original input.

There is also a matrix representation that is equivalent to the list of coordinate tuples in the previous section. We want an input matrix to contain the coordinates: *[[4,5],[5,5],[6,5]]*. For this representation you have to set an attribute of the projection object. The name of the attribute is `wcs.Projection.rowvec`. Its default value is *False*. When you set it to *True* then each row in the matrix represents a position in x and y. Here is an example:

```
proj = wcs.Projection(header)
proj.rowvec = True
pixel = numpy.matrix( [[4,5],[5,5],[6,5]] )
world = proj.toworld(pixel)
print world

# Output:
# [[ 45.01154701  29.99999798]
# [ 45.          30.        ]
# [ 44.98845299  29.99999798]]

pixel = proj.topixel(world)
print pixel

# Output:
# [[ 4.00000001  5.        ]
# [ 5.          5.        ]
# [ 5.99999999  5.        ]]
```

Note: The rowvec attribute can also be set in the constructor of the projection object as follows: *proj = wcs.Projection(header, rowvec=True)*

NumPy arrays

It is possible to build a NumPy array with x coordinates and another for the y coordinates. You can use these arrays in a tuple. Then the elements in the tuple are not lists, as in the previous section, but NumPy arrays. With the same example in mind as the one with the NumPy matrix we demonstrate this option in the following script:

```
proj = wcs.Projection(header)
x = numpy.array( [4,5,6] )
y = numpy.array( [5,5,5] )
pixel = (x, y)
world = proj.toworld(pixel)
print world

# Output:
# (array([ 45.01154701,  45. ,  44.98845299]), array([ 29.99999798,  30. ,  29.99999798]))

pixel = proj.topixel(world)
print pixel

# Output:
# (array([ 4.00000001,  5.         ,  5.99999999]), array([ 5.,  5.,  5.]))
```

As you can see, the representation of the output is the same as that of the input. The result is a tuple and the elements of the tuple are 1 dimensional (rank 1, shape N) NumPy arrays. The first array contains the RA's and the second the Dec's. The closure test also gives the expected result.

Using NumPy arrays to convert an entire map

For applications that transform all the positions in a data set (or in a subset of the data) in one run (e.g. for re-projections of images), it is possible to store all the positions in a NumPy array with shape (NAXIS2, NAXIS1, 2) (note the order). The array can be handled by the `wcs.Projection.toworld()` and `wcs.Projection.topixel()` in one step. You could say that we have a two-dimensional array of which the elements are coordinate pairs. The example code below could be part of the body of a real application that re-projects an image:

```
from kapteyn import wcs
import numpy

header = {  'NAXIS'  : 2,
            'NAXIS1' : 5,
            'CTYPE1' : 'RA---TAN',
            'CRVAL1' : 45,
            'CRPIX1' : 5,
            'CUNIT1' : 'deg',
            'CDELT1' : -0.01,
            'NAXIS2' : 10,
            'CTYPE2' : 'DEC--TAN',
            'CRVAL2' : 30,
            'CRPIX2' : 10,
            'CUNIT2' : 'deg',
            'CDELT2' : +0.01,
         }

proj = wcs.Projection(header)
n1 = 10
n2 = 8
pixel = numpy.zeros(shape=(n2,n1,2))
for y in xrange(n2):
   for x in xrange(n1):
      pixel[y, x] = (x+1, y+1)

world = proj.toworld(pixel)
print world

```

```
# Output:
# [[[ 45.04614616  29.90999204]
#    [ 45.03460962  29.90999556]
#    [ 45.02307308  29.90999807]
#    [ 45.01153654  29.90999957]
# etc.

pixel = proj.topixel(world)
print pixel

# Output:
# [[[  1.    1.]
#    [  2.    1.]
#    [  3.    1.]
#    [  4.    1.]
# etc.
```

In this example we have NAXIS2=10 y values and NAXIS1=5 x values. The indices start at 0, but the FITS pixel indices start at 1. That's why the coordinate tuple reads as (x+1, y+1).

Note: In this module the values in the NumPy arrays and matrices are of type 'f8' (64 bit).

2.1.4 Attributes

Attributes lonaxnum, lataxnum and specaxnum

In the previous examples we had foreknowledge of the axis numbers that represented a spatial axis or a spectral axis. If you read a header from a FITS file then it is not always obvious what the axes represent and in which order they are stored in the FITS header. In those circumstances the projection attributes `wcs.Projection.lonaxnum`, `wcs.Projection.lataxnum` and `wcs.Projection.specaxnum` are very useful. These attributes are axis numbers, i.e. they start with 1 and the highest number is equal to header item 'NAXIS'. In the source below we provide a header which shows an unexpected axis order representing a number of spatial maps as function of frequency. For demonstration purposes we create two separate Projection objects. The first, called *line*, represents the spectral axis. This is a sub projection of the parent projection object and the axis number is that of the spectral axis. We add a spectral translation to get velocities in the output.

The second, called *map*, is the spatial map with axis longitude first and latitude second. We try to create these objects in a try/except clause. For any header, this results in the requested sub projections for a spatial map and spectral axis or an error message and an exception. The construction with the attributes and the try/except clause saves us tedious work because without, we need to find and inspect the axis numbers ourselves.

Note: If WCSLIB cannot find a value of one of the requested attributes, its value is set to *None*

```
#!/usr/bin/env python
from kapteyn import wcs
header = { 'NAXIS'  : 3,
           'NAXIS3' : 5,
           'CTYPE3' : 'RA---NCP',
           'CRVAL3' : 45,
           'CRPIX3' : 5,
           'CUNIT3' : 'deg',
           'CDELT3' : -0.01,
           'CTYPE2' : 'FREQ',
            'CRVAL2' : 1378471216.4292786,
            'CRPIX2' : 32,
            'CUNIT2' : 'Hz',
            'CDELT2' : 97647.745732,
            'RESTFRQ': 1.420405752e+9,
            'NAXIS1' : 10,
            'CTYPE1' : 'DEC--NCP',
            'CRVAL1' : 30,
            'CRPIX1' : 15,
            'CUNIT1' : 'deg',
            'CDELT1' : +0.01
          }
try:
   proj = wcs.Projection(header)
   line = proj.sub(proj.specaxnum).spectra('VRAD')
   map  = proj.sub( (proj.lonaxnum, proj.lataxnum) )
except:
   print "Could not find a spatial map AND a spectral line!"
   raise

print proj.lonaxnum, proj.lataxnum, proj.specaxnum

# Output:
# 3 1 2

# A transformation along the spectral axis:
pixels = range(30, 35)
Vwcs = line.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000

# Output:
# 30 8891.97019336
# 31 8871.36054878
# 32 8850.75090419
# 33 8830.14125961
# 34 8809.53161503

# A transformation of a coordinate in a spatial map:
ra  = header['CRVAL'+str(proj.lonaxnum)]
dec = header['CRVAL'+str(proj.lataxnum)]
print map.topixel( (ra,dec) )

# Output:
# (5.0, 15.0)

# Are these indeed the CRPIXn?
ax1 = "CRPIX"+str(proj.lonaxnum)
ax2 = "CRPIX"+str(proj.lataxnum)
print map.topixel( (ra,dec) ) == (header[ax1], header[ax2])

# Output:
# True
```

Note the check at the end of the code. It should return *True* (i.e. within some limited precision). We started with world coordinates equal to the values of CRVALn from the header and we assert that these correspond to pixel values equal to the corresponding CRPIXn.

Two dimensional data slices with only one spatial axis

Suppose we have a 3D data set with CTYPE's: (RA—NCP, DEC–NCP, VOPT-F2W) and we want to write coordinate labels in a plot that represents the data as function of one spatial axis and the spectral axis (usually called a position-velocity plot or XV map)? It is obvious that we need extra information about the spatial axis that is left out. Usually this is a pixel position that corresponds to the position on the missing axis along which a data slice is taken. These data slices are fixed on pixel coordinates and not on world coordinates.

Assume the XV data we want to plot has axis types DEC–NCP and VOPT-F2W, then we need to specify at which pixel coordinate in Right Ascension the data is extracted.

What we need is a sub-projection (i.e. a Projection object which is modified by method *sub()*) which represents the WCSLIB types: ('latitude', 'spectral', 'longitude'). Given the CTYPE's from the header, the axis permutation sequence that is needed for the sub projection is (2,3,1). Now we require a method that for instance calculates for a given world coordinate in Declination (e.g. 60.1538880206 deg) and a velocity (e.g. -243000.0 m/s) and a fixed pixel for R.A. (e.g. 51) the corresponding pixel coordinates.

The required method is called `wcs.Projection.mixed()`. In a previous section we discussed its use. Method *mixed()* has for a Projection object *p* the following syntax and parameters.

world, pixel = p.mixed(world, pixel, span=None, step=0.0, iter=7)

It is a hybrid transformation suited for celestial coordinates. It uses an iterative method to find an unknown pixel- or world coordinate. The iteration is controlled by parameters span, step and iter. They have reasonable defaults which usually give good results. The method needs knowledge about elements that need to be solved. Unknown values that need to be solved are initially set to NaN (i.e. numpy.nan).

With the numbers we listed, the input world coordinate tuple will be *world_in = (60.1538880206, -243000.0, numpy.nan)*. The input pixel tuple will be: *pixel_in = (numpy.nan, numpy.nan, 51)* then we find the missing coordinates after applying the lines:

```python
subproj = proj.sub([2,3,1])
world_in = (60.1538880206, -243000.0, numpy.nan)
pixel_in = (numpy.nan, numpy.nan, 51)
world_out, pixel_out = subproj.mixed(world_in, pixel_in)
print "world_out = ", world_out
# world_out = (60.1538880206, -243000.0, -51.282084795900005)
print "pixel_out = ", pixel_out
# pixel_out = (51.0, -20.0, 51.0)
```

The *mixed()* method in `wcs` is more powerful than its equivalent in the C-version of WCSLIB. It accepts the same coordinate representations as for *topixel()* and *toworld()* whereas the library version accepts only one coordinate pair per call.

2.1.5 Invalid coordinates

Suppress exceptions for invalid coordinates

We introduced matrices and arrays as coordinate representations to facilitate the input and output of many coordinates in one call. This is in many practical situations the most efficient way to process those coordinates. However if there is a pixel coordinate in a sequence that could not be converted to a world coordinate then an exception will be raised and your script will stop. One can suppress the exception and flag the unknown coordinate. You need to set the `wcs.Projection.allow_invalid` attribute of the projection object. Invalid coordinates then are flagged in the output with a NaN (i.e. numpy.nan). On the other hand, if the input contains a NaN, the corresponding converted coordinate will also be a NaN. You can test whether a value is a NaN with function *numpy.isnan()*. NaN's cannot be compared so a simple test as in:

```
>>> x = numpy.nan
>>> if x == numpy.nan:          # ... fails
```

will fail because the result is always *False* The test x != x will give True if x is NaN.

In practice it will be difficult to get into problems if you convert from world coordinates to pixel coordinates, but when you start with pixel coordinates then it is possible that a corresponding world coordinate is not available. For a projection like Aitoff's projection it is obvious that the rectangle in which an all sky map in this the projection is enclosed, contains such pixels.

Here is an example how one can deal with invalid transformations:

```
#!/usr/bin/env python
from kapteyn import wcs
import numpy
header = { 'NAXIS'  : 2,
           'NAXIS1' : 5,
           'CTYPE1' : 'RA---AIT',
           'CRVAL1' : 45,
           'CRPIX1' : 5,
           'CUNIT1' : 'deg',
           'CDELT1' : -0.01,
           'NAXIS2' : 10,
           'CTYPE2' : 'DEC--AIT',
           'CRVAL2' : 30,
           'CRPIX2' : 5,
           'CUNIT2' : 'deg',
           'CDELT2' : +0.01,
         }
proj = wcs.Projection(header)
proj.allow_invalid = True
pixel_in = numpy.matrix( [[4000,5000,6000],[5000,5000,7580]] )
world = proj.toworld(pixel_in)
print "World coordinates:\n",world
pixel_out = proj.topixel(world)
print "Back to pixels:\n", pixel_out

if numpy.isnan(pixel_out).any():
   print "Some pixels could not be converted"

indices = numpy.where(numpy.isnan(pixel_out))
print "Index of NaNs: ", indices
print pixel_in[indices]
```

2.1.6 Reading data from a FITS file

Reading a FITS header

Until now, we created our own header as a Python dictionary. But usually our starting point is a FITS file. A FITS file can contain more than one header. Header data is read from a FITS file with methods from module `pyfits`. Select the unit you want and store it in a variable (like *header*) so that it can be parsed by wcs. Below we demonstrate how to read the first header from a FITS file.

A flag is set to enter WCSLIB's debug mode:

```
#!/usr/bin/env python
from kapteyn import wcs
```

```python
import pyfits

wcs.debug = True
f = raw_input('Enter name of FITS file: ')
hdulist = pyfits.open(f)
header = hdulist[0].header
proj = wcs.Projection(header)

# Part of the output of arbitrary FITS file:
# Output:
#      flag: 137
#      naxis: 3
#      crpix: 0x99b53d8
#               51            51           -20
#         pc: 0x99adf10
#    pc[0][]:    1             0             0
#    pc[1][]:    0             1             0
#    pc[2][]:    0             0             1
#      cdelt: 0x99b71c8
#           -0.007166     0.007166     4200
#      crval: 0x992bd30
#           -51.282       60.154      -2.43e+05
#      cunit: 0x99ad768
#             "deg"
#             "deg"
#             "m/s"
#      ctype: 0x999a7f8
#             "RA---SIN"
#             "DEC--SIN"
#             "VELO"
```

For testing and debugging one often wants to inspect the items in a FITS header. PyFITS has a nice method to make a list with all the FITS cards. In the next example we added a little filter, using list comprehension, to filter all items that start with 'HISTORY'. Also we added output for the two projection attributes `wcs.Projection.types` and `wcs.Projection.units`. The script is a useful tool to inspect the FITS file and to check its parsing by module `wcs`:

```python
#!/usr/bin/env python
from kapteyn import wcs
import pyfits

f = raw_input('Enter name of FITS file: ')
hdulist = pyfits.open(f)
header = hdulist[0].header
clist = header.ascardlist()
c2 = [str(k) for k in header.ascardlist() if not str(k).startswith('HISTORY')]
for i in c2:
   print i

proj = wcs.Projection(header)
print "WCS found types: ", proj.types
print "WCS found units: ", proj.units
```

Reading WCS data for a spatial map

For some world coordinate related applications we want to force the input to represent a spatial map. A spatial map has axes of type longitude and latitude. For example if you need to re-project a map from one projection system to another, then you need a matching axis pair, representing a spatial system. If you don't know beforehand what the numbers are of the axes in your FITS file that represent these types, you need a way of checking this. There are some rules. First, we must be able to create a Projection object according to the WCSLIB rules (i.e. the axes must have a valid name and extension). For spatial axes, WCSLIB also requires a matching axis pair. So if you have a FITS file with a R.A. axis and not a Dec axis then module `wcs` will generate an exception with the message *Inconsistent or unrecognized coordinate axis types*.

Finally, if you have a valid header and made a Projection object, then you still have to find the axis numbers that represent a 'longitude' axis and a 'latitude' axis (remember: the number of axes in your data could be more than 2) and the latitude axis could be defined earlier than the longitude axis so the order is also important.

In a previous section we discussed the attributes `wcs.Projection.lonaxnum` and `wcs.Projection.lataxnum`. They can be used to find the requested spatial axis numbers (remember their value is *None* if the requested axis is not available). In the following script we try to create the Projection and sub Projection objects with Python's try/except mechanism.

If we have a valid projection and the right axes, then we check the axes types (and order) with attribute *wcs.Projection.types*:

```
#!/usr/bin/env python
from kapteyn import wcs
import pyfits

f = raw_input('Enter name of FITS file: ')
hdulist = pyfits.open(f)
header = hdulist[0].header
try:
   proj = wcs.Projection(header)
   map = proj.sub((proj.lonaxnum, proj.lataxnum))
except:
   print "Aborting program. Could not find (valid) spatial map."
   raise

# Just a check:
print map.types
```

2.1.7 Celestial transformations with wcs

Celestial systems

Methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()` convert between pixel coordinates and world coordinates. If these world coordinates are spatial, they are calculated for the sky- and reference system as defined in the header (FITS header, GIPSY header, header dictionary). To compare positions one must therefore ensure that these positions are all defined in the same sky- and reference system. If such a position is given in another system (e.g. galactic instead of equatorial), then you have to transform the position to the other sky- and/or reference system. Sometimes you might find a so called *alternate* header in the header information of a FITS file. In an alternate header the WCS related keywords end on a letter A-Z (e.g. CRVAL1A).

Usually these alternate headers describe a world coordinate system for another sky system. But because there could also be different epochs involved, it is worthwhile to have a system that can transform world coordinates between sky- and reference systems and that can do epoch transformations as well.

For the Kapteyn Package we wrote module `celestial`. This module can be used as stand alone module if one is interested in celestial transformations of world coordinates only. But the module is well integrated in module `wcs` so one can use it in the context of `wcs`, with the class `wcs.Transformation`. for conversions of world coordinates between sky-/reference systems and also, if pixel coordinates are involved, methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()` can interpret an alternative sky-

/reference system as the system for which a coordinate has to be calculated. The alternative sky-/reference system is stored in attribute `wcs.projection.skyout`.

Note: If you need transformations of world coordinates between any of the supported input sky-/reference system then you should use objects and methods from class `wcs.Transformation`.

If you need to convert pixel coordinates in a system defined by (FITS) header information, then set the **skyout** attribute of a Projection object and use methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()`

The celestial definitions are described in detail in the background information of module `celestial`. We list the most important features of a celestial definition:

Supported Sky systems (detailed information in *Sky systems*):

1. Equatorial: Equatorial coordinates (α, δ), see next list with reference systems
2. Ecliptic: Ecliptic coordinates (λ, β) referred to the ecliptic and mean equinox
3. Galactic: Galactic coordinates (lII, bII)
4. Supergalactic: De Vaucouleurs Supergalactic coordinates (sgl, sgb)

Supported Reference systems (detailed information in *Reference systems*):

1. FK4: Mean place pre-IAU 1976 system.
2. FK4_NO_E: The old FK4 (barycentric) equatorial system but without the "E-terms of aberration"
3. FK5: Mean place post IAU 1976 system.
4. ICRS: The International Celestial Reference System.
5. J2000: This is an equatorial coordinate system based on the mean dynamical equator and equinox at epoch J2000.

Epochs (detailed information in *Epochs for the equinox and epoch of observation*):

The equinox and epoch of observations are instants of time and are of type string. These strings are parsed by a function of module `celestial` called `celestial.epochs()`. The parser rules are described in the documentation for that function. Each string starts with a prefix. Supported prefixes are:

1. B: Besselian epoch
2. J: Julian epoch
3. JD: Julian date
4. MJD: Modified Julian Day
5. RJD: Reduced Julian Day
6. F: Old and new FITS format (old: *DD/MM/YY* new: *YYYY-MM-DD* or *YYYY-MM-DDTHH:MM:SS*)

Example: Next example is a simple test program for epoch specifications. The function `celestial.epochs()` returns a tuple with three elements:

- the Besselian epoch
- the Julian epoch
- the Julian date.

```python
#!/usr/bin/env python
from kapteyn import wcs
```

```
ep = ['J2000', 'j2000', 'j 2000.5', 'B 2000', 'JD2450123.7',
      'mJD 24034', 'MJD50123.2', 'rJD50123.2', 'Rjd 23433',
      'F29/11/57', 'F2000-01-01', 'F2002-04-04T09:42:42.1']

for epoch in ep:
   B, J, JD = wcs.epochs(epoch)
   print "%24s = B%f, J%f, JD %f" % (epoch, B, J, JD)
```

The output is:

```
#                    J2000 = B2000.001278, J2000.000000, JD 2451545.000000
#                    j2000 = B2000.001278, J2000.000000, JD 2451545.000000
#                 j 2000.5 = B2000.501288, J2000.500000, JD 2451727.625000
#                   B 2000 = B2000.000000, J1999.998723, JD 2451544.533398
#              JD2450123.7 = B1996.109887, J1996.108693, JD 2450123.700000
#                mJD 24034 = B1924.680025, J1924.680356, JD 2424034.500000
#               MJD50123.2 = B1996.109887, J1996.108693, JD 2450123.700000
#               rJD50123.2 = B1996.108518, J1996.107324, JD 2450123.200000
#                Rjd 23433 = B1923.033172, J1923.033539, JD 2423433.000000
#                F29/11/57 = B1957.910029, J1957.909651, JD 2436171.500000
#              F2000-01-01 = B1999.999909, J1999.998631, JD 2451544.500000
# F2002-04-04T09:42:42.1 = B2002.257054, J2002.255728, JD 2452368.904654
```

The strings that start with prefix 'F' are strings read from FITS keywords that represent the date of observation.

The sky definition

Given an arbitrary celestial position and a sky system specification you can transform to any of the other sky system specifications. Module wcs recognizes the following built-in sky specifications:

```
wcs.equatorial - wcs.ecliptic - wcs.galactic - wcs.supergalactic
```

Reference systems are:

```
wcs.fk4 - wcs.fk4_no_e - wcs.fk5 - wcs.icrs - wcs.j2000
```

The syntax for an equatorial sky specification is either a tuple (order of the elements is arbitrary):

```
(sky system, equinox, reference system, epoch of observation)
e.g.: obj.skyout = (wcs.equatorial, "J1983.5", wcs.fk4, "B1960_OBS")
```

or a string with minimal match:

```
(equatorial, equinox, referencesystem, epoch of observation"
e.g.: obj.skyout = "equa J1983.5 FK4 B1960_OBS"
```

Celestial transformations

In this section we check some basic celestial coordinate transformations. Background information can be found in *[Ref2]* or in the background information for module celestial.

Two parameters instantiate an object from class Transformation. The first is a definition of the input celestial system and the second is a definition for the celestial output system. Method `wcs.Transformation.transform()` transforms coordinates associated with the celestial input system to coordinates connected to the celestial output system.

The galactic pole has FK4 coordinates (192.25,27.4) in degrees. If we want to verify this, we need to convert this FK4 coordinate to the corresponding galactic coordinate, which should be (0,90) within the limits of precision of the used numbers. The following script shows that this could be true:

```
from kapteyn import wcs

world_eq = (192.25, 27.4)    # FK4 coordinates of galactic pole
tran = wcs.Transformation("EQ,fk4,B1950.0", "GAL")
world_gal = tran.transform(world_eq)
print world_gal

# Output:
# (120.8656324107187, 89.999949251695512)

# Closure test:
world_eq = tran.transform(world_gal, reverse=True)
print world_eq

# Output:
# (192.25, 27.400000000000002)
```

We added a closure test (parameter *reverse=True*) to give you some feeling about the accuracy. Closure tests usually show errors < 1e-12. We expected the pole at 90 deg., but the difference is about 5e-05 deg. That is too much so there must be another reason for the difference. The reason is described in the background information of module `celestial`. The galactic pole is not a star and the so called elliptic terms of aberration (only for FK4) are not apply to its position. So in fact the pole is given in FK4-NO-E coordinates. If we repeat the exercise with the appropriate input celestial definition, we get:

```
from kapteyn import wcs

world_eq = (192.25, 27.4)    # FK4 coordinates of galactic pole
tran = wcs.Transformation("EQUATORIAL, fk4_no_e, B1950.0", "galactic")
world_gal = tran.transform(world_eq)
print world_gal

# Output:
# (0.0, 90.0)

world_eq = tran.transform(world_gal, reverse=True)
print world_eq

# Output:
# (192.25, 27.400000000000002)
```

which gives the result as expected. Note that we used attribute *reverse* of the Transformation class. The two previous examples show that the transformation class is very useful to check basic celestial transformations.

As another test of a standard celestial transformation, let's check the transformation between galactic and supergalactic coordinates. The supergalactic pole (0,90) deg. has galactic(II) world coordinates (47.37,6.32) deg. The conversion program becomes then:

```
from kapteyn import wcs

world_gal = (47.37, 6.32)    # Galactic l,b (II) of supergalactic pole
tran = wcs.Transformation(wcs.galactic, wcs.supergalactic)
world_sgal = tran.transform(world_gal)
print world_sgal

# Output:
```

```
# (0.0, 90.0)

world_eq = trans.transform(world_sgal, reverse=True)
print world_gal

# Output:
# (47.369999999999997, 6.3200000000000003)
```

which agrees with the theory.

The sky system specifications allow for defaults. So if one wants coordinates in the equatorial system with reference system FK5 and equinox J2000 then the specification *wcs.fk5* will suffice. Below we demonstrate how to transform a coordinate from the FK4 system to FK5. In fact we want to demonstrate that FK4 is slowly rotating with respect to the inertial FK5 system. We do that by varying the assumed time of observation and convert the position (R.A.,Dec) = (0,0). This behaviour is explained in the background documentation of module `celestial`:

```
#!/usr/bin/env python
from kapteyn import wcs

world_eq1 = (0,0)
s_out = wcs.fk5
epochs = range(1950,2010,10)
for ep in epochs:
   s_in = "EQUATORIAL B1950 fk4 " + 'B'+str(ep)
   tran = wcs.Transformation(s_in, s_out)
   world_eq2 = tran.transform(world_eq1)
   print 'B'+str(ep), world_eq2

# Output:
# B1950 (0.64069100057541584, 0.27840943507737015)
# B1960 (0.64069761256120361, 0.27839733383470032)
# B1970 (0.64070422454697784, 0.27838524161663253)
# B1980 (0.64071083653273853, 0.27837314488625808)
# B1990 (0.64071744851848544, 0.27836104815588009)
# B2000 (0.64072406050421915, 0.27834895142549831)
```

Usually FK4 catalog values are in equinox and epoch B1950.0, so this program shows an exceptional case.

Note: We are not restricted to the transformation of one coordinate. The input of positions follow the rules of coordinate representations as described for methods `wcs.Projection.toworld()` and `wcs.Projection.topixel()`.

Combining projections and celestial transformations

In previous sections we showed examples how to use methods of an object of class Projection to convert between pixel coordinates and world coordinates. We added the option to change the celestial definition. If your data is a spatial map and its sky system is FK5, then we can convert pixel positions to world coordinates in for example galactic coordinates by specifying a value for attribute `wcs.Projection.skyout`. In our case this would be for a projection object called *proj*:

```
>>> proj.skyout = wcs.galactic
```

In the next example we test (like in one of the previous examples) a conversion between an equatorial system and the galactic system. The FK4-NO-E coordinates of the galactic pole are the values (*CRVAL1*, *CRVAL2*) from the header.

First we calculate a couple of world coordinates in the native celestial definition. Then we verify that that native system is indeed FK4-NO-E and the equinox is B1950. That can be verified with:

```
>>> proj.skyout = (wcs.equatorial, wcs.fk4_no_e, 'B1950')
```

Finally we test the conversion to galactic coordinates with:

```
>>> proj.skyout = wcs.galactic
```

With the output sky set to galactic, we find the galactic pole in galactic coordinates i.e. (90,0) deg. Finally we want to know what the values of the input pixel coordinates are if the output sky system is supergalactic. The galactic pole is (90, 6.32) deg. in supergalactic coordinates. Within the limits of the precision of the used numbers we find the expected output with this script:

```
from kapteyn import wcs
header = { 'NAXIS'  : 2,
           'NAXIS1' : 5,
           'CTYPE1' : 'RA---TAN',
           'CRVAL1' : 192.25,
           'CRPIX1' : 5,
           'CUNIT1' : 'degree',
           'CDELT1' : -0.01,
           'NAXIS2' : 10,
           'CTYPE2' : 'DEC--TAN',
           'CRVAL2' : 27.4,
           'CRPIX2' : 5,
           'CUNIT2' : 'degree',
           'CDELT2' : +0.01,
           'RADESYS': 'FK4-NO-E',
           'EQUINOX': 1950.0
         }

proj = wcs.Projection(header)

pixel = [(4,5),(5,5),(6,5)]   # List with coordinate tuples
world = proj.toworld(pixel)
print world
# [(192.26126360281495, 27.399999547653639), (192.25, 27.399999999999999), ...

proj.skyout = "Equatorial FK4-NO-E B1950"
world = proj.toworld(pixel)
print world
# [(192.26126360281495, 27.399999547653639), (192.24999999999997, 27.400000000000002),...

proj.skyout = "galactic"
world = proj.toworld(pixel)
print world
# [(33.00000000001878,  89.990000000101531), (0.0, 90.0), ...

proj.skyout = wcs.supergalactic
world = proj.toworld(pixel)
print world
# [(90.002497049104363, 6.3296871263660073), (90.000000000000014, 6.3199999999999995), ...
```

Note that the second tuple on each line of the output represents the world coordinates at CRPIX. Also important is the observation that the longitude for galactic coordinates shows erratic behaviour. The reason is that close to a pole, the longitudes are less well defined (and undefined on the pole) and the errors in longitudes become important because we are calculating with numbers with a limited precision.

Attributes of a Projection object related to celestial systems

There are a number of attributes of an object of class `wcs.Projection`, related to celestial systems, that can be used to inspect the parsed FITS header. The native system in the previous example could be derived from attribute `wcs.Projection.skysys`:

```
from kapteyn import wcs
header = { 'NAXIS'  : 2,
           'NAXIS1' : 5,
           'CTYPE1' : 'RA---TAN',
           'CRVAL1' : 192.25,
           'CRPIX1' : 5,
           'CUNIT1' : 'degree',
           'CDELT1' : -0.01,
           'NAXIS2' : 10,
           'CTYPE2' : 'DEC--TAN',
           'CRVAL2' : 27.4,
           'CRPIX2' : 5,
           'CUNIT2' : 'degree',
           'CDELT2' : +0.01,
           'RADESYS': 'FK4-NO-E',
           'EQUINOX': 1950.0,
           'MJD-OBS': 36010.2
         }

proj = wcs.Projection(header)
print "Attributes of 'proj':"
print "skysys:    ", proj.skysys
print "equinox:   ", proj.equinox
print "epoch:     ", proj.epoch
print "dateobs:   ", proj.dateobs
print "mjdobs:    ", proj.mjdobs
print "epobs:     ", proj.epobs

# Attributes of 'proj':
# skysys:      (0, 5, 'B1950.0')
# equinox:     1950.0
# epoch:       B1950.0
# dateobs:     None
# mjdobs:      36010.2
# epobs:       MJD36010.2
```

Below a table with a short explanation of the attributes. More information about epochs and equinoxes can be found in the documentation of `celestial`.

Attribute	Explanation
skysys	A single value or tuple which defines the native system. Tuples can contain the sky system, the reference system, the equinox and the date of observation.
equinox	equinox is a floating point number. It is read from the FITS header (keyword EQUINOX). The equinox is a moment in time used for the definition of an equatorial system.
epoch	This attribute is the epoch of the equinox. That is the value of the equinox with prefix ‘J’ or ‘B’. The context (a.o. keyword RADESYS) sets the prefix.
dateobs	Date of observation. Floating point number given by FITS keyword DATE-OBS
mjdobs	Date of observation. Floating point number given by FITS keyword MJD-OBS
epobs	Date of observation as an epoch, i.e. copied from mjdobs or dateobs and prefixed by ‘F’ or ‘MJD’

Available functions from `celestial`

Some of the functions defined in the module `celestial` are also available in the namespace of `wcs`. One of these is `celestial.epochs()` for which we wrote an example in the previous section. Others are `celestial.lon2hms()`, `celestial.lon2dms()` and `celestial.lat2hms()` to format degrees into hours, minutes, seconds or degrees, minutes and seconds. Finally the function `celestial.skymatrix()` is also available to `wcs`; it calculates the rotation matrix to convert a coordinate from one sky system to another and it calculates the E-terms (see background documentation for celestial) if appropriate. Usually you will only use this function to compare rotation matrices with matrices from the literature or to do some debugging. Some examples on the Python command line:

Formatting spatial coordinates:

```
>>> wcs.lon2hms(45.0)
'03h 00m  0.0s'
>>> wcs.lon2hms(23.453839, 4)
'01h 33m 48.9214s'
>>> wcs.lon2dms(245.0, 4)
Out[10]: ' 245d  0m  0.0000s'
>>> wcs.lat2dms(45.0)
'+45d 00m  0.0s'
>>> help(wcs.lon2hms)
```

Calculate a rotation matrix:

```
>>> wcs.skymatrix(wcs.galactic, wcs.supergalactic)
(matrix([[ -7.35742575e-01,   6.77261296e-01,  -6.08581960e-17],
        [ -7.45537784e-02,  -8.09914713e-02,   9.93922590e-01],
        [  6.73145302e-01,   7.31271166e-01,   1.10081262e-01]]), None, None)
```

2.1.8 Spectral transformations

Introduction

The most important documentation about conversions of spectral coordinates in WCSLIB is found paper "Representations of spectral coordinates in FITS" (paper III, *[Ref3]*) In the next sections we show how `wcs`/WCSLIB can deal with spectral conversions with the focus on conversions between frequencies and velocities. We discuss conversion examples shown in the paper in detail and try to illustrate how `wcs` deals with FITS data from (legacy) AIPS and GIPSY sources. In many of those files the reference frequencies and reference velocities are not given in the same reference system (e.g. topocentric vs. barycentric). It is estimated that there are many of these FITS files and that their headers generate wrong results when they are used to create an object the constructor of `wcs.Projection` class unmodified. For FITS files generated with legacy software some extra interpretation of the FITS header is applied. This procedure is described in more detail in the background information related to spectral coordinates.

Transformations between frequencies and velocities

We built applications that use WCSLIB to convert grid positions, in an image or a spectrum, to world coordinates. For spectral axes with frequency as the primary type (e.g. in the FITS header we read CTYPE3='FREQ'), it is possible to convert between pixel coordinates and frequencies, but also, if the header provides the correct information, between pixel coordinates and velocities. WCSLIB expects that in a FITS header the given frequencies are bound to the same standard of rest (i.e. reference system) as the given reference velocity. In practice however there are many FITS files that list the frequencies in the topocentric system and a reference velocity in an inertial system (barycentric, lsrk). In those FITS files the inertial systems are usually abbreviated with 'HEL' or 'LSR' (Heliocentric, Local Standard of Rest) and the velocities are usually not the true velocities but are either the so called *radio* or *optical* velocities (of which we give the definitions in the background information about spectral coordinates).

Basic spectral line header example

In "Representations of spectral coordinates in FITS" (*[Ref3]*) section 10.1 deals with an example of a VLA spectral line cube which is regularly sampled in frequency (CTYPE3='FREQ'). The section describes how one can define alternative FITS headers to deal with different velocity definitions. We want to examine this exercise in more detail than provided in the article to illustrate how a FITS header can be modified. In the background information you find a more elaborate discussion. Here we summarize some results.

The topocentric spectral properties in the FITS header from the paper are:

```
CTYPE3= 'FREQ'
CRVAL3=  1.37835117405e9
CDELT3=  9.765625e4
CRPIX3=  32
CUNIT3= 'Hz'
RESTFRQ= 1.420405752e+9
SPECSYS='TOPOCENT'
```

Usually such descriptions are part of a header that describes a three dimensional data structure where the first two axes represent a spatial map as function of the third axis which is a spectral axis. This example tells us that the spatial data corresponding with channel 32 was observed at a topocentric frequency (SPECSYS='TOPOCENT') of 1.37835117405 GHz. The step size in frequency is 97.65625 kHz. A rest frequency (1.420405752e+9 Hz) is needed to convert frequencies to velocities. The description of standard FITS keywords can be found in *[FITS]*

The topocentric frequency (for the receiver) was derived from a barycentric optical velocity of 9120 km/s that was requested by an observer.

We prepared a minimal header to simulate this FITS header and calculate world coordinates for the spectral axis. The numbers are frequencies. The units are *Hz* and the central frequency is *CRVAL3*. The step in frequency is *CDELT3*. Our minimal header (here presented as a Python dictionary) shows only one axis so our header items got axis number 1 (e.g. *CRVAL1*, *CDELT1*, etc.):

```
from kapteyn import wcs
header = { 'NAXIS'  :  1,
           'CTYPE1' : 'FREQ',
           'CRVAL1' :  1.37835117405e9,
           'CRPIX1' :  32,
           'CUNIT1' : 'Hz',
           'CDELT1' :  9.765625e4
         }
proj = wcs.Projection(header)
pixels = range(30,35)
Fwcs = proj.toworld1d(pixels)
for p,f in zip(pixels, Fwcs):
   print p, f

# Output:
30 1378155861.55
31 1378253517.8
32 1378351174.05
33 1378448830.3
34 1378546486.55
```

The output shows frequency as function of pixel coordinate. Pixel coordinate 32 (=*CRPIX1*) shows the value of *CRVAL1*. Now we have a method to find at which frequency a spatial map in the data cube was observed.

WCSLIB velocities from frequency data

Usually similar FITS headers provide information about a velocity. Velocities is what we need for the analysis of the kinematics and dynamics of the observed objects. But there are several definitions for velocities (*radio*, *optical*, *apparent radial*).

For the radio interferometer, like the WSRT, an observer requesting for an observation, needs to specify:

- A rest frequency
- A velocity or Doppler shift
- A frame definition (bary or lsrk)
- A conversion type (z, radio, optical)
- A time of observation. This time is needed (together with the location of the observatory) to calculate the topocentric frequencies needed for the receivers

The observer requests that an observation must correspond to a velocity or Doppler shift (see list below) and a reference system. Only then topocentric frequencies for the receivers can be calculated.

To convert to another spectral type the constructor from class `wcs.Projection` needs to know which spectral type we want to convert to. The translation is set then with `wcs.Projection.spectra()`. which stands for *spectral translation*.

The parameter that we need to set the translation is *ctype*. Its syntax follows the FITS convention, see note below.

Note: The first four characters of a spectral CTYPE specify the new coordinate type, the fifth character is '-' and the next three characters specify a predefined algorithm for computing the world coordinates from intermediate physical coordinates (*[Ref3]*).

The following spectral types are supported (from *[Ref3]*):

Type	Name	Symbol	Units	Associated with
FREQ	Frequency	ν	Hz	ν
ENER	Energy	E	J	ν
WAVN	Wavenumber	κ	1/m	ν
VRAD	Radio velocity	V	m/s	ν
WAVE	Vacuum wavelength	λ	m	λ
VOPT	Optical velocity	Z	m/s	λ
ZOPT	Redshift	z	-	λ
AWAV	Air wavelength	λa	m	λa
VELO	Apparent radial velocity	v	m/s	v
BETA	Beta factor (v/c)	β	-	v

The non-linear algorithm codes are (from *[Ref3]*):

Code	sampled in	Expressed as
F2W	Frequency	Wavelength
F2V	Frequency	Apparent radial velocity
F2A	Frequency	Air wavelength
W2F	Wavelength	Frequency
W2V	Wavelength	Apparent radial velocity
W2A	Wavelength	Air wavelength
V2F	Apparent radial velocity	Frequency
V2W	Apparent radial velocity	Wavelength
V2A	Apparent radial velocity	Air wavelength
A2F	Air wavelength	Frequency
A2W	Air wavelength	Wavelength
A2V	Air wavelength	Apparent radial velocity

If we want to convert pixel coordinates to optical velocities for our example header, then module `wcs` needs to create a new projection object with *ctype* = VOPT-F2W because VOPT represents an optical velocity and F2W sets the non linear algorithm which converts from the domain where the step size is constant (frequency) to a velocity associated with wavelength (see table above). The following script shows how to use the method `wcs.Projection.spectra()` to create this new object and how to convert the pixel coordinates:

```python
#!/usr/bin/env python
from kapteyn import wcs
header = { 'NAXIS'  : 1,
           'CTYPE1' : 'FREQ',
           'CRVAL1' : 1.37835117405e9,
           'CRPIX1' : 32,
           'CUNIT1' : 'Hz',
           'CDELT1' : 9.765625e4,
           'RESTFRQ': 1.420405752e+9
         }
proj = wcs.Projection(header)
spec = proj.spectra('VOPT-F2W')
pixels = range(30,35)
Vwcs = spec.toworld1d(pixels)
print "Pixel, velocity (%s)" % spec.units
for p,v in zip(pixels, Vwcs):
   print p, v/1000.0

# Output:
# Pixel, velocity (m/s)
# 30 9190.68652655
# 31 9168.7935041
# 32 9146.90358389
# 33 9125.01676527
# 34 9103.13304757
```

Some comments about this example:

- It shows how to add the spectral translation to the projection object. For a conversion from frequency to optical velocity one can derive a new object with *spec = proj.spectra('VOPT-F2W')* or *proj = wcs.Projection(header).spectra('VOPT-F2W')*.
- The output is a list with pixel coordinates and *topocentric* velocities. This explains why we don't see the requested velocity (9120 km/s) at CRPIX because that velocity was barycentric.
- When we enter an invalid algorithm code for the velocity, the script will raise an exception.

Why do we need a rest frequency?

To get a velocity, the rest frequency needs to be added (RESTFRQ=) to our minimal header. What you get then is a list of velocities according to:

$$Z = c(\frac{\lambda - \lambda_0}{\lambda_0}) = c\,(\frac{\nu_0 - \nu}{\nu}) \tag{2.1}$$

We adopted variable Z for velocities following the optical definition. The frequency as (linear) function of pixel coordinate is:

$$\nu = \nu_{ref} + (N - N_{\nu_{ref}})\delta\nu \tag{2.2}$$

where:

- ν_{ref} is the *reference frequency* (CRVAL1)
- N is the pixel coordinate (FITS definition) we are interested in,
- $N_{\nu_{ref}}$ is the frequency reference pixel (CRPIX1)
- $\delta\nu$ is the frequency increment (CDELT1)

Let's check this with a small script:

```
from kapteyn import wcs

header = { 'NAXIS'  : 1,
           'CTYPE1' : 'FREQ',
           'CRVAL1' : 1.37835117405e9,
           'CRPIX1' : 32,
           'CUNIT1' : 'Hz',
           'CDELT1' : 9.765625e4,
           'RESTFRQ': 1.420405752e+9
         }
proj = wcs.Projection(header)
spec = proj.spectra(ctype='VOPT-F2W')
pixels = range(30,35)
Vopt = spec.toworld1d(pixels)

print "Pixel coordinate and velocity (%s) with wcs module:" % spec.units
for p,Z in zip(pixels, Vopt):
   print p, Z/1000.0

print "\nPixel coordinate and velocity (%s) with documented formulas:" % spec.units
for p in pixels:
   nu = header['CRVAL1'] + (p-header['CRPIX1'])*header['CDELT1']
   Z = wcs.c*(header['RESTFRQ']-nu)/nu     # wcs.c is speed of light in m/s
   print p, Z/1000.0

# Pixel coordinate and velocity (m/s) with wcs module:
# 30 9190.68652655
# 31 9168.7935041
# 32 9146.90358389
# 33 9125.01676527
# 34 9103.13304757

# Pixel coordinate and velocity (m/s) with documented formulas:
# 30 9190.68652655
# 31 9168.7935041
# 32 9146.90358389
# 33 9125.01676527
# 34 9103.13304757
```

More checks are documented in the background information for spectral coordinates. This one should give you some idea how WCSLIB transforms spectral coordinates. But we still didn't address the question about the reference systems. In our code example, this velocity *Z* is topocentric (defined in the reference system of the observatory) and is not suitable for comparisons because the Earth is moving around its axis and around the Sun. Other reference systems are the barycenter of the Solar system and the Local Standard of Rest. During observations one knows the location of the source, the time of observation and the location of the observatory on Earth. Software then can calculate the (true) velocity of the Earth with respect to a selected inertial reference system and we can transform from topocentric velocities to velocities in another system. Usually these correction velocities (called *topocentric correction*) are not recorded in the FITS file of the data set. The keyword to look for is VELOSYS=

In the background information about spectral coordinates we give a recipe how one can change the value of the reference frequency in CRVAL1 to a barycentric value. The result is CRVAL1=1.37847121643e+9 If you substitute this value for CRVAL1 in the previous script, the output is:

```
Pixel coordinate and velocity (m/s) with wcs module:
30 9163.77531673
31 9141.88610757
32 9119.99999984
33 9098.11699288
34 9076.23708605
```

At pixel coordinate 32 (CRPIX1) the velocity is 9120 km/s as we required. So `wcs` always returns velocities in the same system as the system of reference frequency.

Warning: Reference frequencies given in FITS keyword CRVALn refer to a reference system. This system should be given with FITS keyword SPECSYS (e.g. SPECSYS='TOPOCENT'). Module `wcs` converts between frequencies and velocities in the *same* reference system. You should inspect your FITS header to find what this system is.

Warning: Legacy FITS headers often define frequencies in a Topocentric system. Also a reference velocity is given in another reference system. WCSLIB needs instructions how to convert between these systems. If legacy headers are recognized, module `wcs` tries to convert the frequency system to the reference system of the reference velocity. See also the next section and the background documentation about spectral coordinates

Spectral CTYPE's with special extensions

There are many (old) FITS headers which describe a system where the reference frequency is topocentric and the required reference velocity is given for another reference system. These velocities are given with keywords like VELR or DRVALn and the reference system for the velocities is given as an extension in CTYPEn (e.g.: CTYPE3='FREQ-OHEL'). Image processing systems like AIPS and GIPSY have their own tools to deal with this. If `wcs` recognizes a legacy header, it tries to convert the reference frequency to the system of the required velocity:

```python
from kapteyn import wcs

header = { 'NAXIS'   : 1,
           'CTYPE1' : 'FREQ-OHEL',
           'CRVAL1' : 1.415418199417E+09,
           'CRPIX1' : 32,
           'CUNIT1' : 'HZ',
           'CDELT1' : -7.812500000000E+04,
           'VELR'   : 1.050000000000E+06,
           'RESTFRQ': 0.14204057520000E+10
         }

proj = wcs.Projection(header)
```

```
ctype = 'FREQ-???'
if ctype != None:
   spec = proj.spectra(ctype)
   print "\nSelected spectral translation with algorithm code:", spec.ctype[0]
else:
   spec = proj

crpix = header['CRPIX1']
print "CRVAL from header=%f, CRVAL modified=%f" % (header['CRVAL1'], spec.crval[0])
print "CDELT from header=%f, CDELT modified=%f" % (header['CDELT1'], spec.cdelt[0])
for i in range(-2,+3):
   px = crpix + i
   world = spec.toworld1d(px)
   print "%d %f" % (px, world)

# Output:
# Selected spectral translation with algorithm code: FREQ
# CRVAL from header=1415418199.417000, CRVAL modified=1415448253.482287
# CDELT from header=-78125.000000, CDELT modified=-78123.341180
# 30 1415604500.164647
# 31 1415526376.823467
# 32 1415448253.482287
# 33 1415370130.141107
# 34 1415292006.799927
```

As spectral translation we selected 'FREQ'. If you inspect the output list with frequencies then you will see that the list doesn't show the topocentric frequencies (with CRVAL1 at CRPIX1) but frequencies in the reference system of the given (helocentric) velocity. The attributes *spec.crval[0]* and *spec.cdelt[0]* show new values unequal to the header values.

If you want a list with topocentric frequencies then just omit to apply the `wcs.Projection.spectra()` method (i.e. use *ctype* = *None* in example). The output is what we expect:

```
# Output:
# CRVAL from header=1415418199.417000, CRVAL modified=1415418199.417000
# CDELT from header=-78125.000000, CDELT modified=-78125.000000
# 30 1415574449.417000
# 31 1415496324.417000
# 32 1415418199.417000
# 33 1415340074.417000
# 34 1415261949.417000
```

A note about algorithm codes

It is not always easy to figure out what the algorithm code should be if you want to convert to another spectral type. Therefore WCSLIB allows wildcard characters for the last or the last three characters in CTYPEn. In our example valid entries are:

- *spec = proj.spectra(ctype='VOPT-F2W')*
- *spec = proj.spectra(ctype='VOPT-F2?')*
- *spec = proj.spectra(ctype='VOPT-???')*

The missing algorithm code is returned in `wcs.Projection.ctype` as in:

```
>>> spec = proj.spectra(ctype='VOPT-???')
>>> print "Spectral translation with algorithm code:", spec.ctype[0]
    Spectral translation with algorithm code: VOPT-F2W
```

Module `wcs` uses this feature to build a list with all spectral translations that are allowed for a given Projection object. For each type in the table with spectral types, the wildcards are used to find the algorithm code (assuming that for the given Projection objects and the spectral type only one algorithm is possible). A tuple is created with the allowed spectral translation as first element and its associated unit as second element) and the tuple is added to the list `wcs.Projection.altspec`.

Note: For a given header the attribute `wcs.Projection.altspec` stores all possible spectral translations.

The attribute is useful if you want to write code that prompts a user to enter a spectral translation from a list of allowed translations. It can be used as follows:

```
from kapteyn import wcs

header = { 'NAXIS'   : 1,
           'CTYPE1' : 'VOPT',
           'CRVAL1' : 9120,
           'CRPIX1' : 32,
           'CUNIT1' : 'km/s',
           'CDELT1' : -21.882651442,
           'RESTFRQ': 1.420405752e+9
         }

proj = wcs.Projection(header)
print "Allowed spectral translations:"
for as in proj.altspec:
   print as
spec = proj.spectra(ctype='FREQ-???')
print "\nSelected spectral translation with algorithm code:", spec.ctype[0]

# Output:
# Allowed spectral translations:
# ('FREQ-W2F', 'Hz')
# ('ENER-W2F', 'J')
# ('VOPT', 'm/s')
# ('VRAD-W2F', 'm/s')
# ('VELO-W2V', 'm/s')
# ('WAVE', 'm')
# ('ZOPT', '')
# ('BETA-W2V', '')

# Selected spectral translation with algorithm code: FREQ-W2F
```

From velocities to frequencies

In the background information about spectral coordinates we calculated that for a barycentric system the step size in barycentric velocity is -21.882651442 km/s. Then we are able to setup a header with velocities and use a spectral translation that converts to frequencies, as in the next example:

```
from kapteyn import wcs

header = { 'NAXIS'   : 1,
           'CTYPE1' : 'VOPT-F2W',
           'CRVAL1' : 9120,
           'CRPIX1' : 32,
           'CUNIT1' : 'km/s',
           'CDELT1' : -21.882651442,
            'RESTFRQ': 1.420405752e+9
          }

proj = wcs.Projection(header)
spec = proj.spectra(ctype='FREQ-???')
print "Spectral translation with algorithm code:", spec.ctype[0]
pixels = range(30,35)
Freq = spec.toworld1d(pixels)

print "Pixel coordinate and frequency (%s)" % spec.units
for p,f in zip(pixels, Freq):
   print p, f

# Output:
# Pixel coordinate and frequency (Hz):
# 30 1378275920.94
# 31 1378373568.68
# 32 1378471216.43
# 33 1378568864.18
# 34 1378666511.92
```

The reference frequency is at pixel coordinate 32 and its value (1378471216.43) is exactly the barycentric reference frequency that we used before. What happens if we left out the algorithm code in the header? The output differs (except for the reference frequency at pixel 32). That is because it is assumed that the increments in wavelength are constant and not those in frequency. This is confirmed by the returned algorithm code which is *FREQ-W2F* if CTYPE1='VOPT'

Processing real FITS data

With the knowledge we have at this moment, it is easy to make a small utility that looks for a spectral axis in a FITS file and if it can find one, it converts 5 pixel coordinates in the neighbourhood of CRPIX to world coordinates for all allowed spectral translations:

```
from kapteyn import wcs
import pyfits

f = raw_input("Enter name of FITS file: ")
hdulist = pyfits.open(f)
header = hdulist[0].header
proj = wcs.Projection(header)
ax = proj.specaxnum
if ax == None:
   print "No spectral axis available"
else:
   print "Spectral type from header:", proj.ctype[ax-1]
   crpix = header['CRPIX'+str(ax)]
   for alt in proj.altspec:
      line = proj.sub((ax,)).spectra(alt[0])
      print "Pixel, world for translation %s" % alt[0]
      for i in range(-2,+3):
         px = crpix + i
         world = line.toworld1d(px)    #  to world coordinates
         print "%d %.10g (%s)" % (px, world, alt[1])
```

The projection object reads its header data from the first hdu of the FITS file (*hdulist[0].hdr*) and is set to only convert the spectral axis of the data set: *proj.sub((ax,))*. Remember that the argument is a Python tuple but we have only one axis so the tuple has an extra comma. Header items can be read from the header directly (e.g. *header['CRPIX3']*).

That's how we find the value of CRPIX for the spectral axis. The allowed spectral translations are read from attribute `wcs.Projection.altspec`.

We ran the example for a fits file called *mclean.fits* which is a HI data cube and the third axis is the spectral axis:

```
Enter name of FITS file: mclean.fits
Spectral type from header: FREQ
Pixel, world for translation FREQ
28 1415604500 (Hz)
29 1415526377 (Hz)
30 1415448253 (Hz)
31 1415370130 (Hz)
32 1415292007 (Hz)
Pixel, world for translation ENER
28 9.379902296e-25 (J)
29 9.379384645e-25 (J)
30 9.378866994e-25 (J)
31 9.378349343e-25 (J)
32 9.377831692e-25 (J)
Pixel, world for translation VOPT-F2W
28 1016794.655 (m/s)
29 1033396.411 (m/s)
30 1050000 (m/s)
31 1066605.422 (m/s)
32 1083212.677 (m/s)
etc. etc.
```

2.1.9 References

2.2 Tutorial maputils module

2.2.1 Introduction

Module `maputils` is your toolkit for writing small and dedicated applications for the inspection and of FITS headers, the extraction, manipulation display and re-projection of (FITS) data, interactive inspection of this data (color editing) and for the creation of plots with world coordinate information. Many of the examples in this tutorial are small applications which can be used with your own data with only small modifications (like file names etc.).

Module `maputils` provides methods to draw a *graticule* in a plot showing the world coordinates in the given projection and sky system. One can plot spatial rulers which show offsets of constant distance whatever the projection of the map is. We will also demonstrate how to create a so called *all-sky plot*

The module combines the functionality in modules `wcs` and `celestial` from the Kapteyn package, together with Matplotlib, into a powerful module for the extraction and display of FITS image data or external data described by a FITS header or a Python dictionary with FITS keywords (so in principle you are not bound to FITS files).

We show examples of:

- overlays of different graticules (each representing a different sky system),
- plots of data slices from a data set with more than two axes (e.g. a FITS file with channel maps from a radio interferometer observation)
- plots with a spectral axis with a 'spectral translation' (e.g. Frequency to Radio velocity)
- rulers showing offsets in spatial distance
- overlay a second image on a base image

- plot that covers the entire sky (allsky plot)
- mosaics of multiple images (e.g. HI channel maps)
- a simple movie loop program to view 'channel' maps.

We describe simple methods to add **interaction** to the Matplotlib canvas e.g. for changing color maps or color ranges.

In this tutorial we assume a basic knowledge of FITS files. Also a basic knowledge of Matplotlib is handy but not necessary to be able to modify the examples in this tutorial to suit your own needs. For useful references see information below.

See also:

FITS standard A pdf document that describes the current FITS standard.

Matplotlib Starting point for documentation about plotting with Matplotlib.

PyFITS Package for reading and writing FITS files.

Astropy Newer package for astronomy, also containing the PyFITS functionality.

Module `celestial` Documentation of sky- and reference systems. Useful if you need to define a celestial system.

Module `wcsgrat` Documentation about graticules. Useful if you want to fine tune the wcs coordinate grid.

2.2.2 Maputils basics

Building small display- and analysis utilities with `maputils` is easy. The complexity is usually in finding the right parameters in the right methods or functions to achieve special effects. The structure of a script to create a plot using `maputils` can be summarized with:

- Import maputils module
- Get data from a FITS file or another source
- Create a plot window and tell it where to plot your data
- From the object that contains your data, derive new objects
- With the methods of these new objects, plot an image, contours, graticule etc.
- Do the actual plotting and in a second step fine tune plot properties of various objects
- Inspect, print or save your plot or save your new data to file on disk.

In the example below it is easy to identify these steps:

Example: mu_basic1.py - Show image and allow for color interaction

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

f = maputils.FITSimage("m101.fits")
fig = plt.figure()
frame = fig.add_subplot(1,1,1)
annim = f.Annotatedimage(frame)
annim.Image()
annim.Graticule()
annim.plot()
annim.interact_imagecolors()
plt.show()
```

2.2.3 FITS files

A simple utility to analyze a FITS file

With module maputils one can extract image data from a FITS file, modify it and write it to another FITS file on disk. The methods we use for these purposes are based on package PyFITS (or the equivalent in Astropy, astropy.io.fits) but are adapted to function in the environment of the Kapteyn Package. Note that PyFITS is not part of the Kapteyn Package.

With maputils one can also extract data slices from data described by **more** than **two** axes (e.g. images as function of velocity or polarization). For data with only two axes, it can swap those axes (e.g. to swap R.A. and Declination). Also the limits of the data can be set to extract part of 2-dimensional data. To be able to create plots of unfamiliar data without any user interaction, you need to know some of the characteristics of this data before you can extract the right slice. Module maputils provides routines that can display this relevant information.

First you need to create an object from class maputils.FITSimage. Some information is extracted from the FITS header directly. Other information is extracted from attributes of a wcs.Projection object defined in module wcs. Method maputils.FITSimage.str_axisinfo() gets its information from a header and its associated Projection object. It provides information about things like the sky system and the spectral system, as strings, so the method is suitable to get verbose information for display on terminals and in gui's.

Next we show a simple script which prints meta information of the FITS file *ngc6946.fits*:

Example: mu_fitsutils.py - Print meta data from FITS header or Python dictionary

```
from kapteyn import maputils
from matplotlib import pylab as plt

fitsobject = maputils.FITSimage('ngc6946.fits')

print("HEADER:\n")
print fitsobject.str_header()

print("\nAXES INFO:\n")
print fitsobject.str_axisinfo()

print("\nEXTENDED AXES INFO:\n")
print fitsobject.str_axisinfo(long=True)

print("\nAXES INFO for image axes only:\n")
print fitsobject.str_axisinfo(axnum=fitsobject.axperm)

print("\nAXES INFO for non existing axis:\n")
print fitsobject.str_axisinfo(axnum=4)

print("SPECTRAL INFO:\n")
fitsobject.set_imageaxes(axnr1=1, axnr2=3)
print fitsobject.str_spectrans()
```

This code generates the following output:

```
print """
HEADER:

SIMPLE  =                    T / SIMPLE FITS FORMAT
BITPIX  =                  -32 / NUMBER OF BITS PER PIXEL
NAXIS   =                    3 / NUMBER OF AXES
NAXIS1  =                  100 / LENGTH OF AXIS
etc.
```

```

AXES INFO:

Axis 1: RA---NCP  from pixel 1 to   100
{crpix=51 crval=-51.2821 cdelt=-0.007166 (DEGREE)}
{wcs type=longitude, wcs unit=deg}
etc.

EXTENDED AXES INFO:

axisnr     - Axis number:  1
axlen      - Length of axis in pixels (NAXIS):  100
ctype      - Type of axis (CTYPE):  RA---NCP
axnamelong - Long axis name:  RA---NCP
axname     - Short axis name:  RA
etc.

WCS INFO:

Current sky system:                  Equatorial
reference system:                    ICRS
Output sky system:                   Equatorial
Output reference system:             ICRS
etc.

SPECTRAL INFO:

0   FREQ-V2F (Hz)
1   ENER-V2F (J)
2   WAVN-V2F (1/m)
3   VOPT-V2W (m/s)
etc.
"""
```

The example extracts data from a FITS file on disk as given in the example code. To make the script a real utility one should allow the user to enter a file name. This can be done with Python's *raw-input* function but to make it robust one should check the existence of a file, and if a FITS file has more than one header, one should prompt the user to specify the header. We also have to deal with alternate headers for world coordinate systems (WCS) etc. etc.

Note: To facilitate parameter settings we implemented so called *prompt function*. These are external functions which read context in a terminal and then set reasonable defaults for the required parameters in a prompt. These functions can serve as templates for more advanced functions which are used in gui environments.

The projection class from `wcs` interprests and stores the header information. It serves as the interface between *maputils* and the library *WCSLIB*.

Example: mu_projection.py - Get data from attributes of the projection system

```python
from kapteyn import maputils

print "Projection object from FITSimage:"
fitsobj = maputils.FITSimage("mclean.fits")
print "crvals:", fitsobj.convproj.crval
fitsobj.set_imageaxes(1,3)
print "crvals after axes specification:", fitsobj.convproj.crval
fitsobj.set_spectrans("VOPT-???")
print "crvals after setting spectral translation:", fitsobj.convproj.crval
print "Projection object from Annotatedimage:"
annim = fitsobj.Annotatedimage()
print "crvals:", annim.projection.crval
```

The output:

```
Projection object from FITSimage:
crvals: (178.7792, 53.655000000000001)
crvals after axes specification: (178.7792, 1415418199.4170001, 53.655000000000001)
crvals after setting spectral translation: (178.7792, 1050000.0000000042, 53.655000000000001)
Projection object from Annotatedimage:
crvals: (178.7792, 1050000.0000000042, 53.655000000000001)
```

Explanation:

As soon as the FITSimage object is created, it is assumed that the first two axes are the axes of the data you want to display. After setting alternative axes, a slice is taken and the projection system is changed. Now it shows attributes in the order of the slice and we see a third value in the tuple with term:*CRVAL*'s. That's because the last value represents the necessary matching spatial axis which is needed to do conversions between pixels and world coordinates.

As a next step we set a spectral translation for the second axis which is a frequency axis. Again the projection system changes. We did set the translation to an optical velocity and the printed `CRVAL` is indeed the reference optical velocity from the header of this FITS file (1050 km/s).

When an object from class `maputils.Annotatedimage` is created, the projection data is copied from the `maputils.FITSimage` object. It can be accessed through the `maputils.Annotatedimage.projection` attribute (last line of the above example).

Specification of a map

Class `maputils.FITSimage` extracts data from a FITS file so that a map can be plotted with its associated world coordinate system. So we have to specify a number of parameters to get the required image data. This is done with the following methods:

- **Header** - The constructor `maputils.FITSimage` needs name and path of the FITS file. It can be a file on disk or an URL. The file can be zipped. A FITS file can contain more than one header data unit. If this is an issue you need to enter the number of the unit that you want to use. A case insensitive name of the hdu is also allowed. A FITS header can also contain one or more *alternate* headers. Usually these describe another sky or spectral system. We list three examples. The first is a complete description of the FITS header. The second get its parameters from an external 'prompt' function (see next section) and the third uses a prompt function with a pre specfication of parameter *alter* which sets the alternate header.

```
>>> fitsobject = maputils.FITSimage('alt2.fits', hdunr=0, alter='A', memmap=1)
>>> fitsobject = maputils.FITSimage(promptfie=maputils.prompt_fitsfile)
>>> fitsobject = maputils.FITSimage(promptfie=maputils.prompt_fitsfile, alter='A')
>>> fitsobject = maputils.FITSimage('NICMOSn4hk12010_mos.fits', hdunr='err')
```

  Later we will also discuss examples where we use external headers (i.e. not from a FITS file, but as a Python dictionary) and external data (i.e. from another source or from processed data). The user/programmer is responsible for the right shape of the data. Here is a small example of processed data:

```
>>> f = maputils.FITSimage("m101.fits", externaldata=log(abs(fftA)+1.0))
```

- **Axis numbers** - Method `maputils.FITSimage.set_imageaxes()` sets the axis numbers. These numbers follow the FITS standard, i.e. they start with 1. If the data has only two axes then it is possible to swap the axes. This method can be used in combination with an external prompt function. If the data has more than two axes, then the default value for the axis numbers *axnr1=1* and *axnr2=2*. One can also enter the names of the axes. The input is case insensitive and a minimal match is applied to the axis names found in the CTYPE keywords in the header. Examples are:

```
>>> fitsobject.set_imageaxes(promptfie=maputils.prompt_imageaxes)
>>> fitsobject.set_imageaxes(axnr1=2, axnr2=1)
>>> fitsobject.set_imageaxes(2,1)          # swap
>>> fitsobject.set_imageaxes(1,3)          # XV map
>>> fitsobject.set_imageaxes('R','D')      # RA-DEC map
>>> fitsobject.set_imageaxes('ra','freq') # RA-FREQ map
```

- **Data slices from data sets with more than two axes** - For an artificial set called 'manyaxes.fits', we want to extract one spatial map. The axes order is [frequency, declination, right ascension, stokes]. We extract a data slice at FREQ=2 and STOKES=1. This spatial map is obtained with the following lines:

```
>>> fitsobject = maputils.FITSimage('manyaxes.fits') # FREQ-DEC-RA-STOKES
>>> fitsobject.set_imageaxes(axnr1=3, axnr2=2, slicepos=(2,1))
```

- **Coordinate limits** - If you want to extract only a part of the image then you need to set limits for the pixel coordinates. This is set with `maputils.FITSimage.set_limits()`. The limits can be set manually or with a prompt function. Here are examples of both:

```
>>> fitsobject.set_limits(pxlim=(20,40), pylim=(22,38))
>>> fitsobject.set_limits((20,40), (22,38))
>>> fitsobject.set_limits(promptfie=maputils.prompt_box)
```

- **Output sky definition** - For conversions between pixel- and world coordinates one can define to which output sky definition the world coordinates are related. The sky parameters are set with `maputils.FITSimage.set_skyout()`. The syntax for a sky definition (sky system, reference system, equinox, epoch of observation) is documented in `celestial.skymatrix()`.

```
>>> fitsobject = maputils.FITSimage('m101.fits')
>>> fitsobject.set_skyout("Equatorial, J1952, FK4_no_e, J1980")
    or:
>>> fitsobject.set_skyout(promptfie=maputils.prompt_skyout)
```

Writing data to a FITS file or to append data to a FITS file is also possible. The method written for these purposes is called `writetofits()`. It has parameters to scale data and it is possible to skip writing history and comment keywords. Have a look at the examples:

```
>>> # Write data with scaling
>>> fitsobject.writetofits(history=True, comment=True,
                           bitpix=fitsobject.bitpix,
                           bzero=fitsobject.bzero,
                           bscale=fitsobject.bscale,
                           blank=fitsobject.blank)
```

```
>>> # Append data to existing FITS file
>>> fitsobject.writetofits("standard.fits", append=True, history=False, comment=False)
```

2.2.4 Prompt functions

Usually one doesn't know exactly what's in the header of a FITS file or one has limited knowledge about the input parameters in `maputils.FITSimage.set_imageaxes()` Then a helper function to get the right input is available. It is called `maputils.prompt_imageaxes()` which works only in a terminal.

But a complete description of the world coordinate system implies also that it should possible to set limits for the pixel coordinates (e.g. to extract the most interesting part of the entire image) and specify the sky system in which we present a spatial map or the spectral translation (e.g. from frequency to velocity) for an image with a spectral axis. It is easy to turn our basic script into an interactive application that sets all necessary parameters to extract the required image data from a FITS file. The next script is an example how we use prompt functions to ask a user to enter relevant information. These prompt functions are external functions. They are aware of the data context and set reasonable defaults for the required parameters.

Example: mu_getfitsimage.py - Use prompt functions to set attributes of the FITSimage object and print information about the world coordinate system

```
from kapteyn import maputils

fitsobject = maputils.FITSimage(promptfie=maputils.prompt_fitsfile)
print fitsobject.str_axisinfo()
fitsobject.set_imageaxes(promptfie=maputils.prompt_imageaxes)
fitsobject.set_limits(promptfie=maputils.prompt_box)
print fitsobject.str_spectrans()
fitsobject.set_spectrans(promptfie=maputils.prompt_spectrans)
fitsobject.set_skyout(promptfie=maputils.prompt_skyout)

print("\nWCS INFO:")
print fitsobject.str_wcsinfo()
```

Example: fitsview - Use prompt functions to create a script that displays a FITS image

As a summary we present a small but handy utility to display a FITS image using prompt functions. The name of the FITS file can be a command line argument e.g.: `./fitsimage m101.fits` For this you need to download the code and make the script executable (e.g. chmod u+x) and run it from the command line like:

```
>>> ./fitsview m101.fits
```

```
#!/usr/bin/env python
from kapteyn import wcsgrat, maputils
from matplotlib import pylab as plt
import sys

# Create a maputils FITS object from a FITS file on disk
if len(sys.argv) > 1:
   filename = sys.argv[1]
   fitsobject = maputils.FITSimage(filespec=filename,
                promptfie=maputils.prompt_fitsfile, prompt=False)
else:
   fitsobject = maputils.FITSimage(promptfie=maputils.prompt_fitsfile)

fitsobject.set_imageaxes(promptfie=maputils.prompt_imageaxes)
fitsobject.set_limits(promptfie=maputils.prompt_box)
fitsobject.set_skyout(promptfie=maputils.prompt_skyout)
fitsobject.set_spectrans(promptfie=maputils.prompt_spectrans)
clipmin, clipmax = maputils.prompt_dataminmax(fitsobject)

# Get connected to Matplotlib
fig = plt.figure()
frame = fig.add_subplot(1,1,1)

# Create an image to be used in Matplotlib
annim = fitsobject.Annotatedimage(frame, clipmin=clipmin, clipmax=clipmax)
annim.Image()
annim.Graticule()
#annim.Contours()
frame.set_title(fitsobject.filename, y=1.03)
annim.plot()
annim.interact_toolbarinfo()
annim.interact_imagecolors()
annim.interact_writepos()

plt.show()
```

2.2.5 Image objects

Basic image

If one is interested in displaying image data only (i.e. without any wcs information) then we need very few lines of code as we show in the next example.

Example: mu_simple.py - Minimal script to display image data

```
from kapteyn import maputils
from matplotlib import pyplot as plt

f = maputils.FITSimage("m101.fits")
mplim = f.Annotatedimage()
im = mplim.Image()
mplim.plot()

plt.show()
```

Explanation:

This is a simple script that displays an image using the defaults for the axes, the limits, the color map and many other properties. From an object from class `maputils.FITSimage` an object from class `maputils.FITSimage.Annotatedimage` is derived.

This object has methods to create other objects (image, contours, graticule, ruler etc.) that can be plotted with Matplotlib. To plot these objects we need to call method `maputils.Annotatedimage.plot()`

If you are only interested in displaying the image and don't want any white space around the plot then you need to specify a Matplotlib frame as parameter for `maputils.Annotatedimage()`. This frame is created with Matplotlib's *add_subplot()* or *add_axes()*. The latter has options to specify origin and size of the frame in so called normalized coordinates [0,1]. Note that images are displayed while preserving the pixel aspect ratio. Therefore images are scaled to fit either the given width or the given height.

In the next example we reduce whitespace and use keyword parameters *cmap*, *clipmin* and *clipmax* to set a color map and the clip levels between which the color mapping is applied.

Example: mu_withimage.py - Display image data using keyword parameters

```
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj = maputils.FITSimage("m101.fits")
fitsobj.set_limits((180,344), (180,344))

fig = plt.figure(figsize=(4,4))
frame = fig.add_axes([0,0,1,1])

annim = fitsobj.Annotatedimage(frame, cmap="spectral", clipmin=10000, clipmax=15500)
annim.Image(interpolation='spline36')
print "clip min, max:", annim.clipmin, annim.clipmax
annim.plot()
```

```

plt.show()
```

Fig.: mu_withimage.py - Image with non default plot frame and parameters for color map and clip levels.

RGB image

It is possible to compose an image from three separate images which represent a red, green and blue component. In this case you need to create an `maputils.Annotatedimage` object first. The data associated with this image can be used to draw e.g. contours, while the parameters of the method `maputils.Annotatedimage.RGBimage()` compose the RGB image. The `maputils.Annotatedimage.RGBimage()` method creates a new data array and inserts your R, G & B images in the right place. Then it scales the composed data to a range between 0 and 1. For an RGB image we don't apply interactive color bar editing, but allow for the use of a function or lambda expression to rescale the data to enhance features. In the example we used keyword parameter *fun* to square the data to enhance the image a bit.

Example: mu_rgbdemo.py - display RGB image

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

# In the comments we show how to set a smaller box t display
f_red = maputils.FITSimage('m101_red.fits')
#f_red.set_limits((200,300),(200,300))
f_green = maputils.FITSimage('m101_green.fits')
#f_green.set_limits((200,300),(200,300))
f_blue = maputils.FITSimage('m101_blue.fits')
#f_blue.set_limits((200,300),(200,300))

# Show the three components R,G & B separately
# Show Z values when moving the mouse
fig = plt.figure()
frame = fig.add_subplot(2,2,1); frame.set_title("Red with noise")
a = f_red.Annotatedimage(frame); a.Image()
a.interact_toolbarinfo(wcsfmt=None, zfmt="%g")
frame = fig.add_subplot(2,2,2); frame.set_title("Greens are 1")
a = f_green.Annotatedimage(frame); a.Image()
a.interact_toolbarinfo(wcsfmt=None, zfmt="%g")
frame = fig.add_subplot(2,2,3); frame.set_title("Blues are 1")
a = f_blue.Annotatedimage(frame); a.Image()
a.interact_toolbarinfo(wcsfmt=None, zfmt="%g")

# Plot the composed RGB image
frame = fig.add_subplot(2,2,4); frame.set_title("RGB composed of previous")
annim = f_red.Annotatedimage(frame)
annim.RGBimage(f_red, f_green, f_blue, fun=lambda x:x*x, alpha=1)


# Note: color interaction not possible (RGB is fixed)
annim.interact_toolbarinfo(wcsfmt=None, zfmt="%g")
# Write RGB values to terminal after clicking left mouse button
annim.interact_writepos(pixfmt=None, wcsfmt="%.12f", zfmt="%.3e", hmsdms=False)
maputils.showall()
```

Explanation:

Three FITS files contain data in rectangular shapes in different positions. If you want to use only a part of the images you need to set limits (with method `set_limits()`) for each image separately. The shapes in these example data files have values 1 (or near 1) and do overlap to illustrate the composition of a new color. The region where three shapes overlap is white in the composed output image. For each RGB component a `maputils.FITSimage` is created. One of these is used to make a `maputils.FITSimage.Annotatedimage` object. The three *FITSimage* objects are used as parameters for method `maputils.Annotatedimage.RGBimage()` to set the individual components of a RGB image. The red component is a bit special because we added some Gaussian noise to it. A second parameter used in the method is *alpha*. This is an alpha factor which applies to all the pixels in the composed map.

This script also displays a message in the message tool bar with information about mouse positions and the corresponding image value. For an RGB image, all three image values (z values) are displayed. The format of the message is changed with parameters in `maputils.Annotatedimage.interact_toolbarinfo()` as in:

```
>>> annim.interact_toolbarinfo(wcsfmt=None, zfmt="%g")
```

Figure size

In the previous example we specified a size in inches for the figure. We provide a method `maputils.FITSimage.get_figsize()` to set the figure size in cm. Enter only the direction for which you want to set the size. The other size is calculated using the pixel aspect ratio, but then it is not garanteed that all labels will fit. The method is used as follows:

```
>>> fig = plt.figure(figsize=f.get_figsize(xsize=15, cm=True))
```

2.2.6 Graticules

Introduction

Module `maputils` can create graticule objects with method `maputils.Annotatedimage.Graticule()`. But in fact the method that does all the work is defined in module `wcsgrat` So in the next sections we often refer to module `wcsgrat`.

Module `wcsgrat` creates a *graticule* for a given header with WCS information. That implies that it finds positions on a curve in 2 dimensions in image data for which one of the world coordinates is a constant value. These positions are stored in a graticule object. The positions at which these lines cross one of the sides of the rectangle (made up by the limits in pixels in both x- and y-direction), are stored in a list together with a text label showing the world coordinate of the crossing.

Simple example

Example: mu_axnumdemosimple.py - Simple plot using defaults

```
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj = maputils.FITSimage("m101.fits")
mplim = fitsobj.Annotatedimage()
graticule = mplim.Graticule()
mplim.plot()

plt.show()
```

Explanation:

The script opens an existing FITS file. Its header is parsed by methods in module `wcs` and methods from classes in module `wcsgrat` calculate the graticule data. A plot is made with Matplotlib. Note the small rotation of the graticules.

The recipe:

- Given a FITS file on disk (*m101.fits*) we want to plot a graticule for the spatial axes in the FITS file.
- The necessary information is retrieved from the FITS header with PyFITS through class `maputils.FITSimage`.
- To plot something we need to tell method `maputils.FITSimage.Annotatedimage()` in which frame it must plot. Therefore we need a Matplotlib figure instance and a Matplotlib Axes instance (which we call a frame in the context of *maputils*).
- A graticule representation is calculated by `maputils.Annotatedimage.Graticule()` and stored in object *grat*. The maximum number of defaults are used.
- Finally we tell the Annotated image object *mplim* to plot itself and display the result with Matplotlib's function *show()*. This last step can be compressed to one statement: `maputils.showall()` which plots all the annotated images in your script at once and then call Matplotlib's function *show()*.

The `wcsgrat` module estimates the ranges in world coordinates in the coordinate system defined in your FITS file. The method is just brute force. This is the only way to get good estimates for large maps, rotated maps maps with weird projections (e.g. Bonne) etc. It is also possible to enter your own world coordinates to set limits. Methods in this module calculate 'nice' numbers to annotate the plot axes and to set default plot attributes.

Hint: Matplotlib versions older than 0.98 use module *pylab* instead of *pyplot*. You need to change the import statement to: *from matplotlib import pylab as plt*

Probably you already have many questions about what `wcsgrat` can do more:

- Is it possible to draw labels only and no graticule lines?
- Can I change the starting point and step size for the coordinate labels?
- Is it possible to change the default tick label format?
- Can I change the default titles along the axes?
- Is it possible to highlight (e.g. by changing color) just one graticule line?
- Can I plot graticules in maps with one spatial- and one spectral coordinate?
- Can I control the aspect ratio of the plot?
- Is it possible to set limits on pixel coordinates?

In the following sections we will give a number of examples to answer most of these questions.

Selecting axes for image or graticule

For data sets with **more** than **2** axes or data sets with swapped axes (e.g. Declination as first axis and Right Ascension as second), we need to make a choice of the axes and axes order. To demonstrate this we created a FITS file with four axes. The order of the axes is uncommon and should only demonstrate the flexibility of the `maputils` module. We list the data for these axes in this 'artificial' FITS file:

```
Filename: manyaxes.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU      44  (10, 50, 50, 4)  int32
Axis  1 is FREQ   runs from pixel 1 to    10  (crpix=5 crval,cdelt=1.37835, 9.76563e-05 GHZ)
Axis  2 is DEC    runs from pixel 1 to    50  (crpix=30 crval,cdelt=45, -0.01 DEGREE)
Axis  3 is RA     runs from pixel 1 to    50  (crpix=25 crval,cdelt=30, -0.01 DEGREE)
Axis  4 is POL    runs from pixel 1 to     4  (crpix=1 crval,cdelt=1000, 10 STOKES)
```

You can download the file manyaxes.fits for testing. The world coordinate system is arbitrary.

Example: mu_manyaxes.py - Selecting WCS axes from a FITS file with NAXIS > 2

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

# 1. Read the header
fitsobj = maputils.FITSimage("manyaxes.fits")

# 2. Create a Matplotlib Figure and Axes instance
figsize=fitsobj.get_figsize(ysize=12, xsize=11, cm=True)
fig = plt.figure(figsize=figsize)
frame = fig.add_subplot(1,1,1)

# 3. Create a graticule
fitsobj.set_imageaxes('freq','pol')
mplim = fitsobj.Annotatedimage(frame)
grat = mplim.Graticule(starty=1000, deltay=10)

# 4. Show the calculated world coordinates along y-axis
print "The world coordinates along the y-axis:", grat.ystarts

# 5. Show header information in attributes of the Projection object
#    The projection object of a graticule is attribute 'gmap'
print "CRVAL, CDELT from header:", grat.gmap.crval, grat.gmap.cdelt

# 6. Set a number of properties of the graticules and plot axes
grat.setp_tick(plotaxis="bottom",
               fun=lambda x: x/1.0e9, fmt="%.4f",
               rotation=-30 )

grat.setp_axislabel("bottom", label="Frequency (GHz)")
grat.setp_gratline(wcsaxis=0, position=grat.gmap.crval[0],
                   tol=0.5*grat.gmap.cdelt[0], color='r')
grat.setp_ticklabel(plotaxis="left", position=1000, color='m', fmt="I")
grat.setp_ticklabel(plotaxis="left", position=1010, color='b', fmt="Q")
grat.setp_ticklabel(plotaxis="left", position=1020, color='r', fmt="U")
grat.setp_ticklabel(plotaxis="left", position=1030, color='g', fmt="V")
grat.setp_axislabel("left", label="Stokes parameters")


# 7. Set a title for this frame
title = r"""Polarization as function of frequency at:
        $(\alpha_0,\delta_0) = (121^o,53^o)$"""
t = frame.set_title(title, color='#006400', y=1.01, linespacing=1.4)

# 8. Add labels inside plot
inlabs = grat.Insidelabels(wcsaxis=0, constval=1015,
                           deltapx=-0.15, rotation=90,
                           fontsize=10, color='r',
                           fun=lambda x: x*1e-9, fmt="%.4f.10^9")

w = grat.gmap.crval[0] + 0.2*grat.gmap.cdelt[0]
cv = grat.gmap.crval[1]
# Print without any formatting
inlab2 = grat.Insidelabels(wcsaxis=0, world=w, constval=cv,
                           deltapy=0.1, rotation=20,
                           fontsize=10, color='c')
```

```

pixel = grat.gmap.topixel((w,grat.gmap.crval[1]))
frame.plot( (pixel[0],), (pixel[1],), 'o', color='red' )

# 9. Plot the objects
maputils.showall()
```

The plot shows a system of grid lines that correspond to non spatial axes and it will be no surprise that the graticule is a rectangular system. The example follows the same recipe as the previous ones and it shows how one selects the required plot axes in a FITS file. The axes are set with `maputils.FITSimage.set_imageaxes()` with two numbers. The first axis of a set is axis 1, the second 2, etc. (i.e. FITS standard). The default is (1,2) i.e. the first two axes in a FITS header.

For a R.A.-Dec. graticule one should enter for this FITS file:

```
>>> f.set_imageaxes(3,2)
```

Note: If a FITS file has data which has more than two dimensions or it has two dimensions but you want to swap the x- and y axis then you need to specify the relevant FITS axes with `maputils.FITSimage.set_imageaxes()`. The (FITS) axes numbers correspond to the number n in the FITS keyword CTYPEn (e.g. CTYPE3='FREQ' then the frequency axis corresponds to number 3).

Let's study the plot in more detail:

- The header shows a Stokes axes with an uncommon value for `CRVAL` and `CDELT`. We want to label four graticule lines with the familiar Stokes parameters. With the knowledge we have about this `CRVAL` and `CDELT` we tell the Graticule constructor to create 4 graticule lines (`starty=1000, deltay=10`).
- The four positions are stored in attribute *ystarts* as in `grat.ystarts`. We use these numbers to change the coordinate labels into Stokes parameters with method `wcsgrat.Graticule.setp_ticklabel()`

```
    >>> grat.setp_ticklabel(plotaxis="left", position=1000, color='m', fmt="I")
```

- We use `wcsgrat.Insidelabels()` to add coordinate labels inside the plot. We mark a position near `CRVAL` and plot a label and with the same method we added a single label at that position.

This example shows an important feature of the underlying module `wcsgrat` and that is its possibility to change properties of graticules, ticks and labels. We summarize:

- *Graticule line* properties are set with `wcsgrat.Graticule.setp_gratline()` or the equivalent `wcsgrat.Graticule.setp_lineswcs1()` or `wcsgrat.Graticule.setp_lineswcs1()`. The properties are all Matplotlib properties given as keyword arguments. One can apply these to all graticule lines, to one of the wcs types or to one graticule line (identified by its position in world coordinates).
- *Graticule ticks* (the intersections with the borders) are modified by method `wcsgrat.Graticule.setp_tick()`. Ticks are identified by either the wcs axis (e.g. longitude or latitude) or by one of the four rectangular plot axes or by a position in world coordinates. Combinations of these are also possible. Plot properties are given as Matplotlib keyword arguments. The labels can be scaled and formatted with parameters *fun* and *fmt*. Usually one uses method `wcsgrat.Graticule.setp_ticklabel()` to change tick labels and `wcsgrat.Graticule.setp_tickmark()` to change the tick markers.
- Ticks can be native to a plot axis (e.g. an pixel X axis which corresponds to a R.A. axis in world coordinates. But sometimes you can have ticks from two world coordinate axes along the same pixel axis (e.g. for a rotated plot). Then it is possible to control which ticks are plotted and which not. A tick mode for one or more plot/pixel axes is set with `wcsgrat.Graticule.set_tickmode()`.
- *The titles along one of the rectangular plot axes* can be modified with `wcsgrat.Graticule.setp_axislabel()` which is a specialization of method

wcsgrat.Graticule.setp_plotaxis(). A label text is set with parameter *label* and the plot properties are given as Matplotlib keyword arguments.

- Properties of *labels inside a plot* are set in the constructor wcsgrat.Insidelabels.setp_label().
- Properties of *labels along a ruler* are set with method rulers.Ruler.setp_label(). Properties of the ruler line can be changed with rulers.Ruler.setp_line()
- For labels along the plotaxes which correspond to pixel positions one can change the properties of the labels with maputils.Pixellabels.setp_label() while the properties of the markers can be changed with: maputils.Pixellabels.setp_marker()

Let's summarize these methods in a table:

Object	Properties method
Graticule line piece	wcsgrat.Graticule.setp_gratline()
Graticule tick marker	wcsgrat.Graticule.setp_tickmark()
Graticule tick_label	wcsgrat.Graticule.setp_ticklabel()
Axis label	wcsgrat.Graticule.setp_axislabel()
Inside label	wcsgrat.Insidelabels.setp_label()
Ruler labels	rulers.Ruler.setp_label()
Ruler line	rulers.Ruler.setp_line()
Pixel labels	maputils.Pixellabels.setp_label()
Pixel markers	maputils.Pixellabels.setp_marker()
Free graticule line	wcsgrat.Graticule.setp_linespecial()

-*Table*- Objects related to graticules and their methods to set properties.

In the following sections we show some examples for changing the graticule properties. Note that for some methods we can identify objects either with the graticule line type (i.e. 0 or 1), the number or name of the plot axis ([0..4] or one of 'left', 'bottom', 'right', 'top' (or a minimal match of these strings). Some objects (e.g. tick labels) can also be identified by a position in world coordinates. Often also a combination of these identifiers can be used.

Graticule axis labels

Example: mu_labeldemo.py - Properties of axis labels

```python
from kapteyn import maputils
from matplotlib import pylab as plt

header = {
'NAXIS':                    2 ,
'NAXIS1':                 100 ,
'NAXIS2':                 100 ,
'CDELT1':  -7.165998823000E-03 ,
'CRPIX1':   5.100000000000E+01 ,
'CRVAL1':  -5.128208479590E+01 ,
'CTYPE1': 'RA---NCP          ' ,
'CUNIT1': 'DEGREE            ' ,
'CDELT2':   7.165998823000E-03 ,
'CRPIX2':   5.100000000000E+01 ,
'CRVAL2':   6.015388802060E+01 ,
'CTYPE2': 'DEC--NCP          ' ,
'CUNIT2': 'DEGREE            ' ,
}

fig = plt.figure(figsize=(6,5.2))
frame = fig.add_axes([0.15,0.15,0.8,0.8])
f = maputils.FITSimage(externalheader=header)
```

```
annim = f.Annotatedimage(frame)
grat = annim.Graticule()
grat.setp_axislabel(fontstyle='italic')        # Apply to all
grat.setp_axislabel("top", visible=True, xpos=0.0, ypos=1.0, rotation=180)
grat.setp_axislabel("left",
                    backgroundcolor='y',
                    color='b',
                    style='oblique',
                    weight='bold',
                    ypos=0.3)
grat.setp_axislabel("bottom",                  # Label in LaTeX
                    label=r"$\mathrm{Right\ Ascension\ (2000)}$",
                    fontsize=14)
annim.plot()
plt.show()
```

Graticule lines

Example: mu_gratlinedemo.py - Properties of graticule lines

```
from kapteyn import maputils
from matplotlib import pylab as plt

header = {'NAXIS': 2 ,'NAXIS1':100 , 'NAXIS2': 100 ,
'CDELT1':  -7.165998823000E-03, 'CRPIX1': 5.100000000000E+01 ,
'CRVAL1':  -5.128208479590E+01, 'CTYPE1': 'RA---NCP', 'CUNIT1': 'DEGREE ',
'CDELT2':   7.165998823000E-03 , 'CRPIX2': 5.100000000000E+01,
'CRVAL2': 6.015388802060E+01 , 'CTYPE2': 'DEC--NCP ', 'CUNIT2': 'DEGREE'
}

fig = plt.figure(figsize=(6,5.2))
frame = fig.add_subplot(1,1,1)
f = maputils.FITSimage(externalheader=header)
annim = f.Annotatedimage(frame)
grat = annim.Graticule()
grat.setp_gratline(lw=2)
grat.setp_gratline(wcsaxis=0, color='r')
grat.setp_gratline(wcsaxis=1, color='g')
grat.setp_gratline(wcsaxis=1, position=60.25, linestyle=':')
grat.setp_gratline(wcsaxis=0, position="20d34m0s", linestyle=':')
# If invisible, use: grat.setp_gratline(visible=False)

annim.plot()
plt.show()
```

Note: If you don't want to plot graticule lines, then use method `wcsgrat.setp_gratline()` with attribute *visible* set to *False*.

Graticule tick labels

Example: mu_ticklabeldemo.py - Properties of graticule tick labels

```
from kapteyn import maputils
from matplotlib import pylab as plt
```

```python

header = {'NAXIS': 2 ,'NAXIS1':100 , 'NAXIS2': 100 ,
'CDELT1': -7.165998823000E-03, 'CRPIX1': 5.100000000000E+01 ,
'CRVAL1': -5.128208479590E+01, 'CTYPE1': 'RA---NCP', 'CUNIT1': 'DEGREE ',
'CDELT2':  7.165998823000E-03, 'CRPIX2': 5.100000000000E+01,
'CRVAL2':  6.015388802060E+01, 'CTYPE2': 'DEC--NCP ', 'CUNIT2': 'DEGREE'
}

fig = plt.figure()
frame = fig.add_axes([0.20,0.15,0.75,0.8])
f = maputils.FITSimage(externalheader=header)
annim = f.Annotatedimage(frame)
grat = annim.Graticule()
grat2 = annim.Graticule(skyout='Galactic')
grat.setp_ticklabel(plotaxis="bottom", position="20h34m", fmt="%g",
                    color='r', rotation=30)
grat.setp_ticklabel(plotaxis='left', color='b', rotation=20,
                    fontsize=14, fontweight='bold', style='italic')
grat.setp_ticklabel(plotaxis='left', color='m', position="60d0m0s",
                    fmt="DMS", tex=False)
grat.setp_axislabel(plotaxis='left', xpos=-0.25, ypos=0.5)
# Rotation is inherited from previous setting
grat2.setp_gratline(color='g')
grat2.setp_ticklabel(visible=False)
grat2.setp_axislabel(visible=False)

annim.plot()
plt.show()
```

Graticule tick markers

Example: mu_tickmarkerdemo.py - Properties of graticule tick markers

```python
from kapteyn import maputils
from matplotlib import pylab as plt


header = {'NAXIS': 2 ,'NAXIS1':100 , 'NAXIS2': 100 ,
'CDELT1': -7.165998823000E-03, 'CRPIX1': 5.100000000000E+01 ,
'CRVAL1': -5.128208479590E+01, 'CTYPE1': 'RA---NCP', 'CUNIT1': 'DEGREE ',
'CDELT2': 7.165998823000E-03 , 'CRPIX2': 5.100000000000E+01,
'CRVAL2': 6.015388802060E+01 , 'CTYPE2': 'DEC--NCP ', 'CUNIT2': 'DEGREE'
}


fig = plt.figure()
#frame = fig.add_axes([0.15,0.15,0.8,0.8])
frame = fig.add_subplot(1,1,1)
f = maputils.FITSimage(externalheader=header)
annim = f.Annotatedimage(frame)
grat = annim.Graticule()
grat.setp_gratline(visible=False)
grat.setp_ticklabel(plotaxis="bottom", position="20h34m", fmt="%6f")
grat.setp_tickmark(plotaxis="bottom", position="20h34m",
                   color='b', markeredgewidth=4, markersize=20)
fig.text(0.5, 0.5, "Empty", fontstyle='italic', fontsize=18, ha='center',
         color='r')
```

```
annim.plot()
plt.show()
```

Graticule tick mode

Example: mu_tickmodedemo.py - Graticule's tick mode

```
from kapteyn import maputils
from matplotlib import pylab as plt

header = {'NAXIS': 2 ,'NAXIS1':100 , 'NAXIS2': 100 ,
'CDELT1': -7.165998823000E-03, 'CRPIX1': 5.100000000000E+01 ,
'CRVAL1': -5.128208479590E+01, 'CTYPE1': 'RA---NCP', 'CUNIT1': 'DEGREE ',
'CDELT2': 7.165998823000E-03 , 'CRPIX2': 5.100000000000E+01,
'CRVAL2': 6.015388802060E+01 , 'CTYPE2': 'DEC--NCP ', 'CUNIT2': 'DEGREE',
'CROTA2': 80
}

fig = plt.figure(figsize=(7,7))
fig.suptitle("Messy plot. Rotation is 80 deg.", fontsize=14, color='r')
fig.subplots_adjust(left=0.18, bottom=0.10, right=0.90,
                    top=0.90, wspace=0.95, hspace=0.20)
frame = fig.add_subplot(2,2,1)
f = maputils.FITSimage(externalheader=header)
annim = f.Annotatedimage(frame)
xpos = -0.42
ypos = 1.2
grat = annim.Graticule()
grat.setp_axislabel(plotaxis=0, xpos=xpos)
frame.set_title("Default", y=ypos)

frame2 = fig.add_subplot(2,2,2)
annim2 = f.Annotatedimage(frame2)
grat2 = annim2.Graticule()
grat2.setp_axislabel(plotaxis=0, xpos=xpos)
grat2.set_tickmode(mode="sw")
frame2.set_title("Switched ticks", y=ypos)

frame3 = fig.add_subplot(2,2,3)
annim3 = f.Annotatedimage(frame3)
grat3 = annim3.Graticule()
grat3.setp_axislabel(plotaxis=0, xpos=xpos)
grat3.set_tickmode(mode="na")
frame3.set_title("Only native ticks", y=ypos)

frame4 = fig.add_subplot(2,2,4)
annim4 = f.Annotatedimage(frame4)
grat4 = annim4.Graticule()
grat4.setp_axislabel(plotaxis=0, xpos=xpos)
grat4.set_tickmode(plotaxis=['bottom','left'], mode="Switch")
grat4.setp_ticklabel(plotaxis=['top','right'], visible=False)
frame4.set_title("Switched and cleaned", y=ypos)

maputils.showall()
```

Graticule 'inside' labels

Example: mu_insidelabeldemo.py - Graticule 'inside' labels

```python
from kapteyn import maputils
from matplotlib import pylab as plt

header = {'NAXIS': 2 ,'NAXIS1':100 , 'NAXIS2': 100 ,
'CDELT1': -7.165998823000E-03, 'CRPIX1': 5.100000000000E+01 ,
'CRVAL1': -5.128208479590E+01, 'CTYPE1': 'RA---NCP', 'CUNIT1': 'DEGREE ',
'CDELT2': 7.165998823000E-03 , 'CRPIX2': 5.100000000000E+01,
'CRVAL2': 6.015388802060E+01 , 'CTYPE2': 'DEC--NCP ', 'CUNIT2': 'DEGREE'
}

fig = plt.figure()
frame = fig.add_axes([0.15,0.15,0.8,0.8])
f = maputils.FITSimage(externalheader=header)
annim = f.Annotatedimage(frame)
grat = annim.Graticule()
grat2 = annim.Graticule(skyout='Galactic')
grat2.setp_gratline(color='g')
grat2.setp_ticklabel(visible=False)
grat2.setp_axislabel(visible=False)
inswcs0 = grat2.Insidelabels(wcsaxis=0, deltapx=5, deltapy=5)
inswcs1 = grat2.Insidelabels(wcsaxis=1, constval='95d45m')
inswcs0.setp_label(color='r')
inswcs0.setp_label(position="96d0m", color='b', tex=False, fontstyle='italic')
inswcs1.setp_label(position="12d0m", fontsize=14, color='m')
annim.plot()
annim.interact_toolbarinfo()
plt.show()
```

Graticule offset axes

Example: mu_offsetaxes.py - Graticule offset labeling

```python
from kapteyn import maputils
from matplotlib import pylab as plt


def plotgrat(n, ax1, ax2, offsetx=None, offsety=None, unitsx=None):
   f.set_imageaxes(ax1,ax2)
   frame = fig.add_subplot(4,2,n)
   annim = f.Annotatedimage(frame)
   grat = annim.Graticule(offsetx=offsetx, offsety=offsety, unitsx=unitsx)
   grat.setp_axislabel((0,1,2), fontsize=10)
   grat.setp_ticklabel(fontsize=7)

   xmax = annim.pxlim[1]+0.5; ymax = annim.pylim[1]+0.5
   ruler = annim.Ruler(x1=xmax, y1=0.5, x2=xmax, y2=ymax,
                       lambda0=0.5, step=10.0,
                       units='arcmin',
                       fliplabelside=True)

   ruler.setp_line(lw=2, color='r')
   ruler.setp_label(clip_on=True, color='r', fontsize=9)

   ruler2 = annim.Ruler(x1=0.5, y1=0.5, x2=xmax, y2=ymax,
                        lambda0=0.5, step=10.0/60.0,
                        fun=lambda x: x*60.0, fmt="%4.0f^\prime",
                        mscale=6, fliplabelside=True)
   ruler2.setp_line(lw=2, color='b')
   ruler2.setp_label(color='b', fontsize=9)
   grat.setp_axislabel("right", label="Offset (Arcmin.)",
                       visible=True, backgroundcolor='y')


# Main ...
fig = plt.figure(figsize=(7,8))
fig.subplots_adjust(left=0.17, bottom=0.10, right=0.92,
                    top=0.93, wspace=0.24, hspace=0.34)


header = { 'NAXIS':3,'NAXIS1':100, 'NAXIS2':100 , 'NAXIS3':101 ,
#'CDELT1':  -7.165998823000E-03,
'CDELT1': -11.165998823000E-03, 'CRPIX1': 5.100000000000E+01 ,
'CRVAL1':  -5.128208479590E+01, 'CTYPE1': 'RA---NCP' , 'CUNIT1': 'DEGREE',
'CDELT2':   7.165998823000E-03, 'CRPIX2': 5.100000000000E+01,
'CRVAL2':   6.015388802060E+01, 'CTYPE2': 'DEC--NCP', 'CUNIT2': 'DEGREE',
'CDELT3':   4.199999809000E+00, 'CRPIX3': -2.000000000000E+01,
'CRVAL3':  -2.430000000000E+02, 'CTYPE3': 'VELO-HEL', 'CUNIT3': 'km/s',
'EPOCH ':   2.000000000000E+03,
'FREQ0 ':   1.420405758370E+09
}

f = maputils.FITSimage(externalheader=header)
plotgrat(1,3,1)
plotgrat(2,1,3)
plotgrat(3,3,2)
plotgrat(4,2,3, unitsx="arcsec")
plotgrat(5,1,2, offsetx=True)
plotgrat(6,2,1)
plotgrat(7,3,1, offsetx=True, unitsx='km/s')
plotgrat(8,1,3, offsety=True)

maputils.showall()
```

Graticule minor tick marks

Example: mu_minorticks.py - Graticule with minor tick marks

```
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj = maputils.FITSimage("m101.fits")
fig = plt.figure()
fig.subplots_adjust(left=0.18, bottom=0.10, right=0.90,
                    top=0.90, wspace=0.95, hspace=0.20)
for i in range(4):
   f = fig.add_subplot(2,2,i+1)
   mplim = fitsobj.Annotatedimage(f)
   if i == 0:
      majorgrat = mplim.Graticule()
      majorgrat.setp_gratline(visible=False)
   elif i == 1:
      majorgrat = mplim.Graticule(offsetx=True, unitsx='ARCMIN')
      majorgrat.setp_gratline(visible=False)
   elif i == 2:
      majorgrat = mplim.Graticule(skyout='galactic', unitsx='ARCMIN')
      majorgrat.setp_gratline(color='b')
   else:
      majorgrat = mplim.Graticule(skyout='galactic',
                           offsetx=True, unitsx='ARCMIN')
      majorgrat.setp_gratline(color='b')

   majorgrat.setp_tickmark(markersize=10)
   majorgrat.setp_ticklabel(fontsize=6)
   majorgrat.setp_plotaxis(plotaxis=[0,1], fontsize=10)
   minorgrat = mplim.Minortickmarks(majorgrat, 3, 5,
             color="#aa44dd", markersize=3, markeredgewidth=2)

maputils.showall()
plt.show()
```

Explanation

Minor tick marks are created in the same way as major tick marks. They are created as a by-product of the instantiation of an object from class `wcsgrat.Graticule`. The method `maputils.Annotatedimage.Minortickmarks()` copies some properties of the major ticks graticule and then creates a new graticule object. The example shows 4 plots representing the same image in the sky.

1. The default plot with minor tick marks
2. Minor tick marks can also be applied on offset axes
3. We selected another sky system for our graticule. The tick marks are now applied to the galactic coordinate system.
4. This is the tricky plot. First of all we observe that the center of the offset axis is not in the middle of the bottom plot axis. This is because the Galactic sky system is plotted upon an equatorial system and therefore it is (at least for this part of the sky) rotated which causes the limits in world coordinates to be stretched. The start of the offset axis is calculated for the common limits in world coordinates and not just those along a plot axis. Secondly, one should observe that the graticule lines for the longitude follow the tick mark positions on the offset axis. And third, the offset seems to have different sign when compared to the Equatorial system. The reason for this is that we took a part of the sky where the Galactic system's longitude runs in an opposite direction.

Graticule label positions

There are many options to control the labeling along each axis in a plot. There are options in the contructor of a Graticule object to give a start value (*startx*, *starty*, in grids or world coordinates) for the first label along an axis. This label is the label that is written with all relevant information. Other labels are derived from this one. If a step size is given (*deltax* or *deltay*) then this will be used as step between labels. A sequence of start values are used to plot labels at their corresponding positions (a value of the step size is overruled). Values for the label positions in *startx*, *starty* given as a string follow the rules described in `positions`. Step sizes, given as a string, can be appended by a unit.

Example: mu_labelsspatial.py - Tricks to improve labeling of spatial axes

```
from kapteyn import maputils
from matplotlib import pylab as plt

header = { 'NAXIS'  : 3,
           'BUNIT'  : 'w.u.',
           'CDELT1' : -1.200000000000E-03,
           'CDELT2' : 1.497160000000E-03, 'CDELT3' : 97647.745732,
           'CRPIX1' : 5, 'CRPIX2' : 6, 'CRPIX3' : 32,
           'CRVAL1' : 1.787792000000E+02, 'CRVAL2' : 5.365500000000E+01,
           'CRVAL3' : 1378471216.4292786,
           'CTYPE1' : 'RA---NCP', 'CTYPE2' : 'DEC--NCP', 'CTYPE3' : 'FREQ-OHEL',
           'CUNIT1' : 'DEGREE', 'CUNIT2' : 'DEGREE', 'CUNIT3' : 'HZ',
           'DRVAL3' : 1.050000000000E+03,
           'DUNIT3' : 'KM/S',
           'FREQ0'  : 1.420405752e+9,
           'INSTRUME' : 'WSRT',
           'NAXIS1' : 100, 'NAXIS2' : 100, 'NAXIS3' : 64
}


fig = plt.figure(figsize=(7,7))
fig.suptitle("Axis labels for spatial maps", fontsize=14, color='r')
fig.subplots_adjust(left=0.18, bottom=0.10, right=0.90,
                    top=0.90, wspace=0.95, hspace=0.20)
frame = fig.add_subplot(2,2,1)
f = maputils.FITSimage(externalheader=header)
f.set_imageaxes(1,2)
annim = f.Annotatedimage(frame)
# Default labeling
grat = annim.Graticule()

frame = fig.add_subplot(2,2,2)
annim = f.Annotatedimage(frame)
# Plot labels with start position and increment
grat = annim.Graticule(startx='11h55m', deltax="15 hmssec", deltay="3 arcmin")

frame = fig.add_subplot(2,2,3)
annim = f.Annotatedimage(frame)
# Plot labels in string only
grat = annim.Graticule(startx='11h55m 11h54m30s')
grat.setp_tick(plotaxis="bottom", texsexa=False)

frame = fig.add_subplot(2,2,4)
annim = f.Annotatedimage(frame)
grat = annim.Graticule(startx="178.75 deg", deltax="6 arcmin", unitsx="degree")
grat.setp_ticklabel(plotaxis="left", fmt="s")

maputils.showall()
```

Explanation

1. Default
2. Plot labels with start position and increment. Note the use of a special unit 'hmssec'. `grat = annim.Graticule(startx='11h55m', deltax="15 hmssec", deltay="3 arcmin")`
3. The LaTeX labeling is jumpy. Try it without superscripts (`texsexa=False`):

```
>>> grat = annim.Graticule(startx='11h55m 11h54m30s')
>>> grat.setp_tick(plotaxis="bottom", texsexa=False)
```

4. Force the y axis NOT to plot seconds. Force the x axis to plot in degrees.

```
>>> grat = annim.Graticule(startx="178.75 deg", deltax="6 arcmin", unitsx="degree")
>>> grat.setp_ticklabel(plotaxis="left", fmt="s")
```

More information about plotting in sexagesimal format is found in `wcsgrat.makelabel()`

Example: mu_labelsspectral.py - Tricks to improve labeling of spectral and offset axes

```python
from kapteyn import maputils
from matplotlib import pylab as plt

header = { 'NAXIS'  : 3,
           'BUNIT'  : 'w.u.',
           'CDELT1' : -1.200000000000E-03,
           'CDELT2' : 1.497160000000E-03, 'CDELT3' : 97647.745732,
           'CRPIX1' : 5, 'CRPIX2' : 6, 'CRPIX3' : 32,
           'CRVAL1' : 1.787792000000E+02, 'CRVAL2' : 5.365500000000E+01,
           'CRVAL3' : 1378471216.4292786,
           'CTYPE1' : 'RA---NCP', 'CTYPE2' : 'DEC--NCP', 'CTYPE3' : 'FREQ-OHEL',
           'CUNIT1' : 'DEGREE', 'CUNIT2' : 'DEGREE', 'CUNIT3' : 'HZ',
           'DRVAL3' : 1.050000000000E+03,
           'DUNIT3' : 'KM/S',
           'FREQ0'  : 1.420405752e+9,
           'INSTRUME' : 'WSRT',
           'NAXIS1' : 100, 'NAXIS2' : 100, 'NAXIS3' : 64
}


fig = plt.figure(figsize=(7,10))
fig.suptitle("Axis label tricks (spectral+spatial offset)", fontsize=14, color='r')
fig.subplots_adjust(left=0.18, bottom=0.10, right=0.90,
                    top=0.90, wspace=0.95, hspace=0.20)
frame = fig.add_subplot(3,2,1)
f = maputils.FITSimage(externalheader=header)
f.set_imageaxes(3,2, slicepos=1)
annim = f.Annotatedimage(frame)
# Default labeling
grat = annim.Graticule()
grat.setp_tick(plotaxis="bottom", rotation=90)

frame = fig.add_subplot(3,2,2)
annim = f.Annotatedimage(frame)
# Spectral axis with start and increment
grat = annim.Graticule(startx="1.378 Ghz", deltax="2 Mhz", starty="53d42m")
grat.setp_tick(plotaxis="bottom", fontsize=7, fmt='%.3f%+9e')

frame = fig.add_subplot(3,2,3)
annim = f.Annotatedimage(frame)
# Spectral axis with start and increment
grat = annim.Graticule(spectrans="WAVE", startx="21.74 cm",
                       deltax="0.04 cm", starty="0.5")

frame = fig.add_subplot(3,2,4)
annim = f.Annotatedimage(frame)
# Spectral axis with start and increment
grat = annim.Graticule(spectrans="VOPT", startx="9120 km/s",
                       deltax="400 km/s", unitsy="arcsec")
grat.setp_tick(plotaxis="bottom", fontsize=7)

frame = fig.add_subplot(3,2,5)
annim = f.Annotatedimage(frame)
# Spectral axis with start and increment and unit
grat = annim.Graticule(spectrans="VOPT", startx="9000 km/s",
                         deltax="400 km/s", unitsx="km/s")

frame = fig.add_subplot(3,2,6)
annim = f.Annotatedimage(frame)
# Spectral axis with start and increment and formatter function
grat = annim.Graticule(spectrans="VOPT", startx="9000 km/s", deltax="400 km/s")
grat.setp_tick(plotaxis="bottom", fmt='%g', fun=lambda x:x/1000.0)
grat.setp_axislabel(plotaxis="bottom", label="Optical velocity (Km/s)")

maputils.showall()
```

Explanation

1. The default labeling of the frequency axis is too crowded. We apply the trick to rotate the axis labels `grat.setp_tick(plotaxis="bottom", rotation=90)`
2. Here we selected a start value and a step size for the label positions: `grat = annim.Graticule(startx="1.378 Ghz", deltax="2 Mhz", starty="53d42m")`. We use a special format syntax (*%+9e*) to tell the plot routines to format the numbers with an exponential: `grat.setp_tick(plotaxis="bottom", fontsize=7, fmt='%.3f%+9e')`
3. The spectral axis can be translated into a wave length axis using parameter *spectrans*. The units change from Hz to m. In the Graticule contructor we use strings for the start value and step size. Then we can use compatible units enter the values. Note that the y axis in our spectral plots is by default an offset axis. For spatial offset axes the reference (value 0) is at the middle of an axis. One can enter a value in grids or world coordinate (enter it as a string) to change this reference point: `grat = annim.Graticule(spectrans="WAVE", startx="21.74 cm", deltax="0.04 cm", starty="0.5")`
4. In this plot we use another spectral translation (optical velocity) with a start value and a step size. We changed the units of the offset axis to seconds of arc. `grat = annim.Graticule(spectrans="VOPT", startx="9120 km/s", deltax="400 km/s", unitsy="arcsec")`
5. Again with a spectral translation. But the units along the x axis are Km/s. Note that the default units (si units) are used for label positions if there are no units entered. This implies that the value in *unitsx* does not change the units for *startx*. It is always save to enter explicit units for *startx*: `grat = annim.Graticule(spectrans="VOPT", startx="9000 km/s", deltax="400 km/s", unitsx="km/s")`
6. You can get even more control if you enter a function or a lambda expression for parameter *fun*. You have to change the default axis title with method `wcsgrat.Graticule.setp_axislabel()`.

```
>>> grat.setp_tick(plotaxis="bottom", fmt='%g', fun=lambda x:x/1000.0)
>>> grat.setp_axislabel(plotaxis="bottom", label="Optical velocity (Km/s)")
```

More ‘axnum’ variations – Position Velocity diagrams

For the next example we used a FITS file with the following header information:

```
Axis 1: RA---NCP  from pixel 1 to   100  {crpix=51 crval=-51.2821 cdelt=-0.007166 (DEGREE)}
Axis 2: DEC--NCP  from pixel 1 to   100  {crpix=51 crval=60.1539 cdelt=0.007166 (DEGREE)}
Axis 3: VELO-HEL  from pixel 1 to   101  {crpix=-20 crval=-243 cdelt=4.2 (km/s)}
```

Example: mu_axnumdemo.py - Show different axes combinations for the same FITS file

```
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj= maputils.FITSimage('ngc6946.fits')
```

```
newaspect = 1/5.0          # Needed for XV maps

fig = plt.figure(figsize=(20/2.54, 25/2.54))
fig.subplots_adjust(left=0.18)
labelx = -0.10             # Fix the  position in x for labels along y

# fig 1. Spatial map, default axes are 1 & 2
frame1 = fig.add_subplot(4,1,1)
mplim1 = fitsobj.Annotatedimage(frame1)
mplim1.Image()
graticule1 = mplim1.Graticule(deltax=15*2/60.0)

# fig 2. Velocity - Dec
frame2 = fig.add_subplot(4,1,2)
fitsobj.set_imageaxes('vel', 'dec')
mplim2 = fitsobj.Annotatedimage(frame2)
mplim2.Image()
graticule2 = mplim2.Graticule()
graticule2.setp_axislabel(plotaxis='left', xpos=labelx)

# fig 3. Velocity - Dec (Version without offsets)
frame3 = fig.add_subplot(4,1,3)
mplim3 = fitsobj.Annotatedimage(frame3)
mplim3.Image()
graticule3 = mplim3.Graticule(offsety=False)
graticule3.setp_axislabel(plotaxis='left', xpos=labelx)
graticule3.setp_ticklabel(plotaxis="left", fmt='DMs')

# fig 4. Velocity - R.A.
frame4 = fig.add_subplot(4,1,4)
fitsobj.set_imageaxes('vel','ra')
mplim4 = fitsobj.Annotatedimage(frame4)
mplim4.Image()
graticule4 = mplim4.Graticule(offsety=False)
graticule4.setp_axislabel(plotaxis='left', xpos=labelx)
graticule4.setp_ticklabel(plotaxis="left", fmt='HMs')
graticule4.Insidelabels(wcsaxis=0, constval='20h34m',
                        rotation=90, fontsize=10,
                        color='r', ha='right')
graticule4.Insidelabels(wcsaxis=1, fontsize=10, fmt="%.2f", color='y')
mplim4.Minortickmarks(graticule4)

#Apply new aspect ratio for the XV maps
mplim2.set_aspectratio(newaspect)
mplim3.set_aspectratio(newaspect)
mplim4.set_aspectratio(newaspect)

maputils.showall()
```

We used Matplotlib's *add_subplot()* method to create 4 plots in one figure with minimum effort. The top panel shows a plot with the default axis numbers which are 1 and 2. This corresponds to the axis types RA and DEC and therefore the map is a spatial map. The next panel has axis numbers 3 and 2 representing a *position-velocity* or *XV map* with DEC as the spatial axis X. The default annotation is offset in spatial distances. The next panel is a copy but we changed the annotation from the default (i.e. offsets) to position labels. This could make sense if the map is unrotated. The bottom panel has RA as the spatial axis X. World coordinate labels are added inside the plot with a special method: `wcsgrat.Insidelabels()`. These labels are not formatted to hour/min/sec or deg/min/sec for spatial axes.

The two calls to this method need some extra explanation:

```
graticule4.Insidelabels(wcsaxis=0, constval='20h34m', rotation=90, fontsize=10,
                        color='r', ha='right')
graticule4.Insidelabels(wcsaxis=1, fontsize=10, fmt="%.2f", color='b')
```

The first statement sets labels that correspond to positions in world coordinates inside a plot. It copies the positions of the velocities, set by the initialization of the graticule object. It plots those labels at a Right Ascension equal to 20h35m which is equal to -51 (=309) degrees. It rotates these labels by an angle of 90 degrees and sets the size, color and alignment of the font. The second statement does something similar for the Right Ascension labels, but it adds also a format for the numbers.

Note also the line:

```
>>> graticule4 = mplim4.Graticule(offsety=False)
>>> graticule4.setp_ticklabel(plotaxis="left", fmt='HMs')
```

By default the module would plot labels which are offsets because we have only one spatial axis. We overruled this behaviour with keyword parameter *offsety=False*. Then we get world coordinates which by default are formatted in hour/minutes/seconds. If we want these labels to be plotted in another format, lets say decimal degrees, then one needs parameter *fun* to define some transformation and with *fmt* we set the format for that output, e.g. as in:

```
>>> graticule4.setp_tick(plotaxis="left", fun=lambda x: x+360, fmt="$%.1f^\circ$")
```

Finally note that the alignment of the titles along the left axis (which is a Matplotlib method) works in the frame of the graticule. It is important to realize that a *maputils* plot usually is a stack of matplotlib Axes objects (frames). The graticule object sets these axis labels and therefore we must align them in that frame (which is an attribute of the graticule object) as in:

```
>>> graticule3.setp_axislabel(plotaxis='left', xpos=labelx)
```

For information about the Matplotlib specific attributes you should read the documentation at the appropriate class descriptions (http://matplotlib.sourceforge.net).

Changing the default aspect ratio

For images and graticules representing spatial data it is important that the aspect ratio (CDELTy/CDELTx) remains constant if you resize the plot. A graticule object initializes itself with an aspect ratio based on the pixel sizes found in (or derived from) the header. It also calculates an appropriate figure size and size for the actual plot window in normalized device coordinates (i.e. in interval [0,1]). You can use these values in a script to set the relevant values for Matplotlib as we show in the next example.

Example: mu_figuredemo.py - Plot figure in non default aspect ratio

```
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj = maputils.FITSimage('example1test.fits')

fig = plt.figure(figsize=(5.5,5))
frame = fig.add_axes([0.1, 0.1, 0.8, 0.8])
annim = fitsobj.Annotatedimage(frame)
annim.set_aspectratio(1.2)
grat = annim.Graticule()

maputils.showall()
```

Note: For astronomical data we want equal steps in spatial distance in any direction correspond to equal steps in figure size. If one changes the size of the figure interactively, the aspect ratio should not change. To enforce this, the constructor of an object of class `maputils.Annotatedimage` modifies the input frame so that the aspect ratio is the aspect ratio of the pixels. This aspect ratio is preserved when the size of a window is changed. One can overrule this default by manually setting an aspect ratio with method `maputils.Annotatedimage.set_aspectratio()` as in:

```
>>> frame = fig.add_subplot(k,1,1)
>>> mplim = f.Annotatedimage(frame)
>>> mplim.set_aspectratio(0.02)
```

Combinations of graticules

The number of graticule objects is not restricted to one. One can easily add a second graticule for a different sky system. The next example shows a combination of two graticules for two different sky systems. It demonstrates also the use of attributes to change plot properties.

Example: mu_skyout.py - Combine two graticules in one frame

```
from kapteyn.wcs import galactic, equatorial, fk4_no_e, fk5
from kapteyn import maputils
from matplotlib import pylab as plt

# Open FITS file and get header
f = maputils.FITSimage('example1test.fits')

fig = plt.figure(figsize=(6,6))
frame = fig.add_subplot(1,1,1)
annim = f.Annotatedimage(frame)

# Initialize a graticule for this header and set some attributes
grat = annim.Graticule()
grat.setp_gratline(wcsaxis=[0,1],color='g') # Set graticule lines to green
grat.setp_ticklabel(plotaxis=("left","bottom"), color='b',
                    fontsize=14, fmt="Hms")
grat.setp_tickmark(plotaxis=("left","bottom"), markersize=-10)

# Select another sky system for an overlay
skyout = galactic        # Also try: skyout = (equatorial, fk5, 'J3000')
grat2 = annim.Graticule(skyout=skyout, boxsamples=20000)
grat2.setp_axislabel(plotaxis=("top", "right"), label="Galactic l,b",
                     visible=True)
grat2.set_tickmode(plotaxis=("top", "right"), mode="ALL")

grat2.setp_axislabel(plotaxis="top", color='r')
grat2.setp_axislabel(plotaxis=("left", "bottom"),  visible=False)
grat2.setp_ticklabel(plotaxis=("left", "bottom"),  visible=False)
grat2.setp_ticklabel(plotaxis=("top", "right"), fmt='Dms')
grat2.setp_gratline(color='r')

# Print coordinate labels inside the plot boundaries
grat2.Insidelabels(wcsaxis=0, color='m', constval=85, fmt='Dms')
grat2.Insidelabels(wcsaxis=1, fmt='Dms')
annim.Pixellabels(plotaxis=("right","top"), gridlines=True,
                  color='c', markersize=-3, fontsize=7)

annim.plot()
plt.show()
```

Explanation:

This plot shows graticules for equatorial coordinates and galactic coordinates in the same figure. The center of the image is the position of the galactic pole. That is why the graticule for the galactic system shows circles. The galactic graticule is also labeled inside the plot using method `wcsgrat.Insidelabels()` (Note that this is a method derived from class `wcsgrat.Graticule` and that it is not a method of class `maputils.Annotatedimage`). To get an impression of arbitrary positions expressed in pixels coordinates, we added pixel coordinate labels for the top and right axes with method `maputils.Annotatedimage.Pixellabels()`.

Plot properties:

- Use attribute *boxsamples* to get a better estimation of the ranges in galactic coordinates. The default sampling does not sample enough in the neighbourhood of the galactic pole causing a gap in the plot.
- Use method `wcsgrat.Graticule.setp_gratline()` to change the color of the longitudes and latitudes for the equatorial system.
- Method `wcsgrat.Graticule.setp_tickmark()` sets for both plot axis (0 == x axis, 1 = y axis) the tick length with *markersize*. The value is negative to force a tick that points outwards. Also the color and the font size of the tick labels is set. Note that these are Matplotlib keyword arguments.
- With `wcsgrat.Graticule.setp_axislabel()` we allow galactic coordinate labels and ticks to be plotted along the top and right plot axis. By default, the labels along these axes are set to be invisible, so we need to make them visible with keyword argument *visible=True*. Also a title is set for these axes.

Note: There is a difference between plot axes and wcs axes. The first always represent a rectangular system with pixels while the system of the graticule lines (wcs axes) usually is curved (sometimes they are even circular. Therefore many plot properties are either associated with one a plot axis and/or a world coordinate axes.

Spectral translations

To demonstrate what is possible with spectral coordinates and module `wcsgrat` we use real interferometer data from a set called *mclean.fits*. A summary of what can be found in its header:

```
Axis  1: RA---NCP  from pixel 1 to   512  {crpix=257 crval=178.779 cdelt=-0.0012 (DEGREE)}
Axis  2: DEC--NCP  from pixel 1 to   512  {crpix=257 crval=53.655 cdelt=0.00149716 (DEGREE)}
Axis  3: FREQ-OHEL from pixel 1 to    61  {crpix=30 crval=1.41542E+09 cdelt=-78125 (HZ)}
```

Its spectral axis number is 3. The type is frequency. The extension tells us that an optical velocity in the heliocentric system is associated with the frequencies. In the header we found that the optical velocity is 1050 Km/s. The header is a legacy GIPSY header and module `wcs` can interpret it. We require the frequencies to be expressed as wavelengths.

Example: mu_wave.py - Plot a graticule in a position wavelength diagram.

```python
from kapteyn import maputils
from matplotlib import pylab as plt

# Make plot window wider if you don't see toolbar info

# Open FITS file and get header
f = maputils.FITSimage('mclean.fits')
f.set_imageaxes('freq','dec')
f.set_limits(pxlim=(35,45))

fig = plt.figure(figsize=f.get_figsize(ysize=12, cm=True))
frame = fig.add_subplot(1,1,1)
annim = f.Annotatedimage(frame)
```

```

grat = annim.Graticule(spectrans='WAVE')
grat.setp_ticklabel(plotaxis='bottom', fun=lambda x: x*100, fmt="%.3f")
grat.setp_axislabel(plotaxis='bottom', label="Wavelength (cm)")
grat.setp_gratline(wcsaxis=(0,1), color='g')

annim.Pixellabels(plotaxis=("right","top"), gridlines=False)
annim.plot()
annim.interact_toolbarinfo()
plt.show()
```

Explanation:

- With PyFITS we open the fits file on disk and read its header
- A Matplotlib Figure- and Axes instance are made
- The range in pixel coordinates in x is decreased
- A Graticule object is created and FITS axis 3 (FREQ) is associated with x and FITS axis 2 (DEC) with y. The spectral axis is expressed in wavelengths with method `wcs.Projection.spectra()`. Note that we omitted a code for the conversion algorithm and instead entered three question marks so that the *spectra()* method tries to find the appropriate code.
- The tick labels along the x axis (the wavelengths) are formatted. The S.I. unit is meter, but we want it to be plotted in cm. A function to convert the values is given with *fun=lambda x: x*100*. A format for the printed numbers is given with: *fmt="%.3f"*

Note: The spatial axis is expressed in offsets. By default it starts with an offset equal to zero in the middle of the plot. Then a suitable step size is calculated and the corresponding labels are plotted. For spatial offsets we need also a value for the missing spatial axis. If not specified with parameter *mixpix* in the constructor of class *Graticule*, a default value is assumed equal to CRPIX corresponding to the missing spatial axis (or 1 if CRPIX is outside interval [1,NAXIS])

For the next example we use the same FITS file (mclean.fits).

Example: mu_spectraltypes.py - Plot grid lines for different spectral translations

```
from kapteyn import maputils
from matplotlib import pyplot as plt

# Read header of FITS file
f = maputils.FITSimage('mclean.fits')

# Matplotlib
fig = plt.figure(figsize=(7,10))
fig.subplots_adjust(left=0.12, bottom=0.05, right=0.97,
                    top=0.97, wspace=0.22, hspace=0.90)
fig.text(0.05,0.5,"Radial offset latitude", rotation=90,
         fontsize=14, va='center')

# Get the projection object to get allowed spectral translations
altspec = f.proj.altspec
crpix = f.proj.crpix[f.proj.specaxnum-1]
altspec.insert(0, (None, ''))  # Add native to list
k = len(altspec) + 1
frame = fig.add_subplot(k,1,1)

# Limit range in x to neighbourhood of CRPIX
xlim = (crpix-5, crpix+5)
```

```python
f.set_imageaxes(3,2)
f.set_limits(pxlim=xlim)
mplim = f.Annotatedimage(frame)
mplim.set_aspectratio(0.002)

print "Native system", f.proj.ctype[f.proj.specaxnum-1], f.proj.cunit[f.proj.specaxnum-1],

print "Spectral translations"
for i, ast in enumerate(altspec):
   print i, ast
   frame = fig.add_subplot(k,1,i+1)
   mplim = f.Annotatedimage(frame)
   mplim.set_aspectratio(0.002)
   grat = mplim.Graticule(spectrans=ast[0], boxsamples=3)
   grat.setp_ticklabel(plotaxis="bottom", fmt="%g")
   unit = ast[1]
   ctype = ast[0]
   if ctype == None:
      ctype = "Frequency (Hz) without translation"
   grat.setp_axislabel(plotaxis="bottom",
                       label=ctype+' '+unit, color='b', fontsize=10)
   grat.setp_axislabel("left", visible=False)
   grat.setp_ticklabel(wcsaxis=(0,1), fontsize='8')
   mplim.plot()

plt.show()
```

Explanation:

- With PyFITS we open the FITS file on disk and read its header
- We created a `wcs.Projection` object for this header to get a list with allowed spectral translations (attribute *altspec*). We need this list before we create the graticules
- Matplotlib Figure- and Axes instances are made
- The native FREQ axis (top figure) differs from the FREQ axis in the next plot, because a legacy header was found and its freqencies were transformed to a barycentric/heliocentric system.

2.2.7 Rulers

Rulers are objects derived from an Annimatedimage object. The class description is found at `rulers.Ruler`. A ruler is always plotted as a straight line, whatever the projection (so it doesn't necessarily follow graticule lines). A ruler plots ticks and labels and the *spatial* distance between any two ticks is a constant given by a user in parameter *step*. This makes rulers ideal to put nearby a feature in your map to give an idea of the physical size of that feature. Rulers can be plotted in maps with one or two spatial axes. They are either defined by a start- and an end point (in pixel or world coordinates) or by a start point and a size and angle (w.r.t. the North). This size and angle are defined on a sphere.

Note: Rulers, Beams and Markers can be positioned using either pixel coordinates or world coordinates in a string. See the examples in module `positions`.

Example: mu_manyrulers.py - Ruler demonstration

```python
from kapteyn import maputils
from matplotlib import pylab as plt
```

```

header = {'NAXIS' : 2, 'NAXIS1': 800, 'NAXIS2': 800,
          'CTYPE1':'RA---TAN',
          'CRVAL1': 0.0, 'CRPIX1' : 1, 'CUNIT1' : 'deg', 'CDELT1' : -0.05,
          'CTYPE2':'DEC--TAN',
          'CRVAL2': 0.0, 'CRPIX2' : 1, 'CUNIT2' : 'deg', 'CDELT2' : 0.05,
         }

fitsobject = maputils.FITSimage(externalheader=header)

fig = plt.figure()
frame = fig.add_axes([0.1,0.1, 0.82,0.82])
annim = fitsobject.Annotatedimage(frame)
grat = annim.Graticule(header)
x1 = 10; y1 = 1
x2 = 10; y2 = annim.pylim[1]

ruler1 = annim.Ruler(x1=x1, y1=y1, x2=x2, y2=y2, lambda0=0.0, step=1.0)
ruler1.setp_label(color='g')
x1 = x2 = annim.pxlim[1]
ruler2 = annim.Ruler(x1=x1, y1=y1, x2=x2, y2=y2, lambda0=0.5, step=2.0,
                     fmt='%3d', mscale=-1.5, fliplabelside=True)
ruler2.setp_label(ha='left', va='center', color='b', clip_on=False)

ruler3 = annim.Ruler(x1=23*15, y1=30, x2=22*15, y2=15, lambda0=0.0,
                     step=2, world=True,
                     units='deg', addangle=90)
ruler3.setp_label(color='r')

ruler4 = annim.Ruler(pos1="23h0m 15d0m", pos2="22h0m 30d0m", lambda0=0.0,
                     step=1,
                     fmt="%4.0f^\prime",
                     fun=lambda x: x*60.0, addangle=0)
ruler4.setp_line(color='g')
ruler4.setp_label(color='m')

ruler5 = annim.Ruler(x1=1, y1=800, x2=800, y2=800, lambda0=0.5, step=2,
                     fmt="%4.1f", addangle=90)
ruler5.setp_label(color='c')

ruler6 = annim.Ruler(pos1="23h0m 15d0m", rulersize=15, step=2,
                     units='deg', lambda0=0, fliplabelside=True)
ruler6.setp_label(color='b')
ruler6.set_title("Size in deg", fontsize=10)

ruler7 = annim.Ruler(pos1="23h0m 30d0m", rulersize=5, rulerangle=90, step=1,
                     units='deg', lambda0=0)
ruler7.setp_label(color='#ffbb33')

ruler8 = annim.Ruler(pos1="23h0m 15d0m", rulersize=5, rulerangle=10, step=1.25,
                     units='deg', lambda0=0, fmt="%02g", fun=lambda x: x*8)
ruler8.setp_label(color='#3322ff')
ruler8.set_title("Size in kpc", fontsize=10)

# Increase size and lambda a bit to get all the labels
# from 5 to 0 and to 5 again plotted
# Show LaTeX in ruler label
ruler9 = annim.Ruler(pos1="23h0m 10d0m", rulersize=10.1, rulerangle=90, step=1,
                          units='deg', lambda0=0.51)
ruler9.setp_label(color='#33ff22')
ruler9.set_title("$\lambda = 0.5$", fontsize=10)

annim.plot()
annim.interact_toolbarinfo()
annim.interact_writepos()
plt.show()
```

Ruler tick labels can be formatted so that we can adjust the values near the ruler ticks with parameter *fmt*. With parameter *fun* it is possible to convert the spatial distance to some other physical quantity. Parameter *fun* accepts a function or a lambda expression. You can use method `rulers.Ruler.set_title()` to annotate alternative units. Note that the keyword arguments for this method are the same as for Matplotlib's `set_title()` method.

In the next plot we want offsets to be plotted in arcminutes.

Example: mu_arcminrulers.py - Rulers with non default labels

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

header = {'NAXIS'  : 2, 'NAXIS1': 100, 'NAXIS2': 100,
          'CTYPE1' : 'RA---TAN',
          'CRVAL1' : 80.0, 'CRPIX1' : 1,
          'CUNIT1' : 'arcmin', 'CDELT1' : -0.5,
          'CTYPE2' : 'DEC--TAN',
          'CRVAL2' : 400.0, 'CRPIX2' : 1,
          'CUNIT2' : 'arcmin', 'CDELT2' : 0.5,
          'CROTA2' : 30.0
         }

f = maputils.FITSimage(externalheader=header)

fig = plt.figure()
frame = fig.add_axes([0.1,0.15,0.8,0.75])
annim = f.Annotatedimage(frame)
grat = annim.Graticule()
grat.setp_ticklabel(plotaxis='bottom', rotation=90)
grat.setp_ticklabel(fmt='s') # Suppress the seconds in all labels

# Use pixel limits attributes of the FITSimage object

xmax = annim.pxlim[1]+0.5; ymax = annim.pylim[1]+0.5
annim.Ruler(x1=xmax, y1=0.5, x2=xmax, y2=ymax, lambda0=0.5, step=5.0/60.0,
            fun=lambda x: x*60.0, fmt="%4.0f^\prime",
            fliplabelside=True, color='r')

# The wcs methods that convert between pixels and world
# coordinates expect input in degrees whatever the units in the
# header are (e.g. arcsec, arcmin).
annim.Ruler(x1=60/60.0, y1=390/60.0, x2=60/60.0, y2=420/60.0,
            lambda0=0.0, step=5.0/60, world=True,
            fun=lambda x: x*60.0, fmt="%4.0f^\prime", color='g')

annim.Ruler(pos1='0h3m30s 6d30m', pos2='0h3m30s 7d0m',
            lambda0=0.0, step=5.0,
            units='arcmin', color='c')

annim.plot()
```

```
plt.show()
```

It is possible to put a ruler in a map with only one spatial coordinate (as long there is a matching axis in the header) like a Position-Velocity diagram (sometimes also called XV maps). It will take the pixel coordinate of the slice as a constant so even for XV maps we have reliable offsets. In the next example we created two rulers. The red ruler is in fact the same as the Y-axis offset labeling. The blue ruler show the same offsets in horizontal direction. That is because only the horizontal direction is spatial. Such a ruler is probably not very useful but is a nice demonstration of the flexibility of method `maputils.Annotatedimage.Ruler()`.

Note that we set Matplotlib's *clip_on* to *True* because if we pan the image in Matplotlib we don't want the labels to be visible outside the border of the frame.

Example: mu_xvruler.py - Ruler in a XV map

```
from kapteyn import maputils
from matplotlib import pylab as plt

# Open FITS file and get header
f = maputils.FITSimage('ngc6946.fits')
f.set_imageaxes(3,2)   # X axis is velocity, y axis is declination

fig = plt.figure(figsize=f.get_figsize(xsize=15, cm=True))
frame = fig.add_subplot(1,1,1)
annim = f.Annotatedimage(frame)

# Velocity - Dec
grat = annim.Graticule()
grat.setp_axislabel("right", label="Offset (Arcmin.)", visible=True)

xmax = annim.pxlim[1]+0.5; ymax = annim.pylim[1]+0.5
ruler = annim.Ruler(x1=xmax, y1=0.5, x2=xmax, y2=ymax,
                    lambda0 = 0.5, step=5.0/60.0,
                    fun=lambda x: x*60.0, fmt="%4.0f^\prime",
                    fliplabelside=True)
ruler.setp_line(lw=2, color='r')
ruler.setp_label(color='r')

ruler2 = annim.Ruler(x1=0.5, y1=0.5, x2=xmax, y2=ymax, lambda0 = 0.5,
                     step=5.0/60.0,
                     fun=lambda x: x*60.0, fmt="%4.0f^\prime",
                     fliplabelside=True)
ruler2.setp_line(lw=2, color='b')
ruler2.setp_label(color='b')


annim.plot()
annim.interact_writepos()

plt.show()
```

2.2.8 Contours

Example: mu_simplecontours.py - Simple plot with contour lines only

```
from kapteyn import maputils
from matplotlib import pyplot as plt

```

```
fitsobj = maputils.FITSimage("m101.fits")
fitsobj.set_limits((200,400), (200,400))

annim = fitsobj.Annotatedimage()
cont = annim.Contours()
annim.plot()

print "Levels=", cont.clevels

plt.show()
```

The example above shows how to plot contours without plotting an image. It also shows how one can retrieve the contour levels that are calculated as a default because no levels were specified.

Next we demonstrate how to use the three Matplotlib keyword arguments to set some global properties of the contours:

Example: mu_contourlinestyles.py - Setting global colors and line styles/widths

```
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj = maputils.FITSimage("m101.fits")
fitsobj.set_limits((200,400), (200,400))

annim = fitsobj.Annotatedimage()
annim.Image(alpha=0.5)
cont = annim.Contours(linestyles=('solid', 'dashed', 'dashdot', 'dotted'),
                      linewidths=(2,3,4), colors=('r','g','b','m'))
annim.plot()

print "Levels=", cont.clevels

plt.show()
```

Example: mu_annotatedcontours.py - Add annotation to contours

```
from kapteyn import maputils
from matplotlib import pyplot as plt

f = maputils.FITSimage("m101.fits")
f.set_limits(pxlim=(200,350), pylim=(200,350))

fig = plt.figure()
frame = fig.add_subplot(1,1,1)

mplim = f.Annotatedimage(frame)
cont = mplim.Contours(levels=range(10000,16000,1000))
cont.setp_contour(linewidth=1)
cont.setp_contour(levels=11000, color='g', linewidth=3)

# Second contour set only for labels
cont2 = mplim.Contours(levels=(8000,9000,10000,11000))
cont2.setp_label(11000, colors='b', fontsize=14, fmt="%.3f")
cont2.setp_label(fontsize=10, fmt="%g \lambda")

mplim.plot()

plt.show()
```

The plot shows two sets of contours. The first step is to plot all contours in a straightforward way. The second is to plot contours with annotation. For this second set we don't see any contours if a label could not be fitted that's why we first plot all the contours. Note that now we can use the properties methods for single contours because we can identify these contours by their corresponding level.

Example: mu_negativecontours.py - Contours with different line styles for negative values

```python
""" Show contour lines with different lines styles """
from kapteyn import maputils
from matplotlib import pyplot as plt

f = maputils.FITSimage("RAxDEC.fits")

fig = plt.figure(figsize=(8,6))
frame = fig.add_subplot(1,1,1)

mplim = f.Annotatedimage(frame)
cont = mplim.Contours(levels=[-500,-300, 0, 300, 500], negative="dotted")
cont.setp_label()
mplim.plot()
mplim.interact_toolbarinfo()

plt.show()
```

2.2.9 Colorbar

A colorbar is an image which shows colors and values which correspond to these colors. It is a tool that helps you to inspect the values in an image. The distribution of the colors depends on the selected color map and the selected clip levels. Next example shows how to setup a colorbar. The default position is calculated by Matplotlib. It borrows space from the current frame depending on the orientation ('vertical' or 'horizontal').

Example: mu_colbar.py - Add colorbar to plot

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj = maputils.FITSimage("m101.fits")

mplim = fitsobj.Annotatedimage(cmap="spectral")
mplim.Image()
units = r'$ergs/(sec.cm^2)$'
colbar = mplim.Colorbar(fontsize=8)
colbar.set_label(label=units, fontsize=24)
mplim.plot()
plt.show()
```

If you want more control over the position and size of the colorbar then specify a frame for the colorbar. In the next example we prepared a frame for both the image and the colorbar. If you don't enter a figure size, it can happen that the figure does not provide enough space in either width or height. In the example we want the colorbar to be as big as the width of the image. This will be a problem with the default figure size so we provided some extra space in height with *figsize=*:

Example: mu_colbarframe.py - Add colorbar with user's frame to plot

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

fitsobj = maputils.FITSimage("m101.fits")
fig = plt.figure(figsize=(5,7.5))
```

```
#fig = plt.figure()
frame = fig.add_axes((0.1, 0.2, 0.8, 0.8))
cbframe = fig.add_axes((0.1, 0.1, 0.8, 0.1))

annim = fitsobj.Annotatedimage(cmap="Accent", clipmin=8000, frame=frame)
annim.Image()
units = r'$ergs/(sec.cm^2)$'
colbar = annim.Colorbar(fontsize=8, orientation='horizontal', frame=cbframe)
colbar.set_label(label=units, fontsize=24)
annim.plot()
annim.interact_imagecolors()
plt.show()
```

Note that we entered a colormap (case sensitive names!) and a value for the lower clip value (below which all image pixels get the same color). The clip for the maximum is not entered so the default will be taken which is the maximum intensity in your image.

Note also that we added interaction to set other colormaps and to change the relation between colors and image values. Interaction is a topic in a later section of this tutorial.

Usually one associates colorbars with images but it can also be used in combination with contours. We demonstrate the use of Matplotlib's keyword parameters *visible=False* to make an image invisible. However, to make the contents of the colorbar invisible one should use *alpha=0.0* but we implemented keyword *visible* to simulate this effect.

Example: mu_colbarwithlines.py - Add lines representing contours in plot to dummy colorbar

```
from kapteyn import maputils
from matplotlib import pyplot as plt

f = maputils.FITSimage("m101.fits")
limy = limx=(160,360)
f.set_limits(limx,limy)
rows = 3
cols = 2

fig = plt.figure(figsize=(8,10))

frame = fig.add_subplot(rows,cols,1)
mplim = f.Annotatedimage(frame, cmap="spectral")
cont = mplim.Contours()
mplim.Colorbar(clines=True, fontsize=8, linewidths=5) # show only cont. lines
mplim.plot()
# Levels only known after plotted
print "Proposed levels:", cont.clevels

frame = fig.add_subplot(rows,cols,2)
mplim = f.Annotatedimage(frame, cmap="spectral")
cont = mplim.Contours(filled=True)
mplim.Colorbar(clines=True, fontsize=8) # show only cont. lines
mplim.plot()

frame = fig.add_subplot(rows,cols,3)
mplim = f.Annotatedimage(frame, cmap="spectral")
mplim.Image()
cont = mplim.Contours(colors='w', linewidths=1)
mplim.Colorbar(clines=True, ticks=(4000,8000,12000))
mplim.plot()
mplim.interact_imagecolors()

```

```
frame = fig.add_subplot(rows,cols,4)
mplim = f.Annotatedimage(frame, cmap="spectral")
mplim.Image()
# Give each contour its own color, instead of borrowing from the colormap
cont = mplim.Contours(levels=(6000,8000,10000,12000),
                      colors=('w','g','b', 'c'))
cont.setp_contour(levels=8000, color='m', linewidth=2)
mplim.Colorbar(clines=True, ticks=(4000,8000,12000), linewidths=6)
mplim.plot()
mplim.interact_imagecolors()
mplim.interact_toolbarinfo()

frame = fig.add_subplot(rows,cols,5)
mplim = f.Annotatedimage(frame, cmap="mousse.lut")
mplim.Image()
cont = mplim.Contours()
mplim.Colorbar(clines=True, orientation="horizontal", ticks=(4000,8000,12000))
mplim.plot()
mplim.interact_imagecolors()
mplim.interact_toolbarinfo()

# With given levels
frame = fig.add_subplot(rows,cols,6)
levels = (10000,11000,12000,13000)
mplim = f.Annotatedimage(frame, cmap="mousse.lut",
                         clipmin=min(levels)-500,
                         clipmax=max(levels)+500)
mplim.Image()
cont = mplim.Contours(levels=levels)
mplim.Colorbar(clines=True, orientation="horizontal",
               ticks=levels)
mplim.plot()
mplim.interact_imagecolors()
mplim.interact_toolbarinfo()

plt.show()
```

2.2.10 Adding pixel coordinate labels

In Matplotlib the axes in a frame are coupled. To get uncoupled axes a we stack frames at the same location. For each frame one can change properties of the pixel coordinate labels separately. The trick is implemented in a number of methods, but in the methods of class `maputils.Pixellabels` it is easy to demonstrate that it works. In the example we defined 4 plot axes for which we want to draw pixel coordinate labels. The constructor uses Matplotlib defaults but these can be overruled by parameters *major* and *minor*. These are numbers for which n*major major ticks and labels are plotted and m*minor minor ticks. Note that the default in Matplotlib is not to plot minor tick marks.

Example: mu_pixellabels.py - Add annotation for pixel coordinates

```
from kapteyn import maputils
from matplotlib import pyplot as plt

fig = plt.figure(figsize=(4,4))
frame = fig.add_subplot(1,1,1)

fitsobject = maputils.FITSimage("m101.fits")
annim = fitsobject.Annotatedimage(frame)
annim.Pixellabels(plotaxis="bottom", major=200, minor=10, color="r")
```

```
pl2 = annim.Pixellabels(plotaxis="right", color="b", markersize=10,
                        gridlines=True)
pl2.setp_marker(markersize=+15, color='b', markeredgewidth=2)
pl3 = annim.Pixellabels(plotaxis="top", color='g',
                        gridlines=False)
pl3.setp_marker(markersize=-10)
pl3.setp_label(rotation=90)
pl4 = annim.Pixellabels(plotaxis="left", major=150, minor=25)
pl4.setp_label(fontsize=10)

annim.plot()
plt.show()
```

2.2.11 Adding a beam

Objects from class Beam are graphical representations of the resolution of an instrument. The beam is plotted at a center position entered as a string that represents a position or as two world coordinates. The major axis of the beam is the FWHM of longest distance between two opposite points on the ellipse. The angle between the major axis and the North is the position angle of the beam. See also `maputils.Annotatedimage.Beam()`.

Note: Rulers, Beams and Markers are positioned using either pixel coordinates or world coordinates. See the examples in module `positions`.

In the next example we added two rulers to prove that the sizes of plotted ellipse are indeed the correct values on a sphere. Note also the use of parameter *units* to set the FWHM's to minutes of arc.

Example: mu_beam.py - Plot an ellipse representing a beam

```
from kapteyn import maputils
from matplotlib import pyplot as plt
from math import cos, radians

fitsobj = maputils.FITSimage('m101.fits')
annim = fitsobj.Annotatedimage()
annim.Image()
grat = annim.Graticule()
grat.setp_ticklabel(wcsaxis=1, fmt='s')    # Exclude seconds in label


# beam = annim.Beam(210.9619, 54.261039, 0.01, 0.01, 0, hatch='*')
# Hatching does not work in mpl 0.98.3

fwhm_maj = 5/60.0   # arcmin to degrees
fwhm_min = 4/60.0
lat = 54.347395233845
lon = 210.80254413455
beam = annim.Beam(fwhm_maj, fwhm_min, 90, xc=lon, yc=lat,
                  fc='g', fill=True, alpha=0.6)
pos = '210.80254413455 deg, 54.347395233845 deg'
beam = annim.Beam(7, 4, units='arcmin', pos=pos, fc='m', fill=True,
                  alpha=0.6)
pos = '14h03m12.6105s 54d20m50.622s'
beam = annim.Beam(fwhm_maj, fwhm_min, pos=pos, fc='y', fill=True, alpha=0.6)
pos = 'ga 102.0354152 {} 59.7725125'
beam = annim.Beam(fwhm_maj, fwhm_min, pos=pos, fc='g', fill=True, alpha=0.6)
```

```
pos = 'ga 102d02m07.494s {} 59.7725125'
beam = annim.Beam(fwhm_maj, fwhm_min, pos=pos, fc='b', fill=True, alpha=0.6)
pos = '{ecliptic,fk4, j2000} 174.3674627 {} 59.7961737'
beam = annim.Beam(fwhm_maj, fwhm_min, pos=pos, fc='r', fill=True, alpha=0.6)
pos = '{eq, fk4-no-e, B1950} 14h01m26.4501s {} 54d35m13.480s'
beam = annim.Beam(fwhm_maj, fwhm_min, pos=pos, fc='c', fill=True, alpha=0.6)
pos = '{eq, fk4-no-e, B1950, F24/04/55} 14h01m26.4482s {} 54d35m13.460s'
beam = annim.Beam(fwhm_maj, fwhm_min, pos=pos, fc='c', fill=True, alpha=0.6)
pos = '{ecl} 174.367764 {} 59.79623457'
beam = annim.Beam(fwhm_maj, fwhm_min, pos=pos, fc='c', fill=True, alpha=0.6)
pos = '53 58'      # Pixels
beam = annim.Beam(0.04, 0.02, pa=30, pos=pos, fc='y', fill=True, alpha=0.4)
pos = '14h03m12.6105s 58'
beam = annim.Beam(0.04, 0.02, pa=-30, pos=pos, fc='y', fill=True, alpha=0.4)

annim.Ruler(x1=lon, y1=lat, x2=lon+fwhm_min/1.99/cos(radians(54.20)), y2=lat,
           world=True, step=1, lambda0=0.0, units='arcmin', color='r')
annim.Ruler(x1=lon, y1=lat, x2=lon, y2=lat+fwhm_maj/1.99, world=True,
           step=1,  lambda0=0.0, units='arcmin',  color='b')

annim.plot()

annim.interact_toolbarinfo()
annim.interact_imagecolors()
plt.show()
```

2.2.12 Markers

Sometimes there are features in an image that you want to mark with a symbol. In other cases you want to plot positions from an external source (file or database etc.). Then you use objects from class `maputils.Annotatedimage.Marker()`. The use is straightforward. Positions can be entered in different formats: as pixel coordinates, as world coordinates or as strings with position information (see module `positions`).

Note: Rulers, Beams and Markers are positioned using either pixel coordinates or world coordinates. See the examples in module `positions`.

Note the use of Matplotlib keyword arguments to set the properties of the marker symbols. The most important are:

```
>>> marker=
>>> markersize=
>>> markeredgewidth=
>>> markeredgecolor=
>>> markerfacecolor=
```

Example: mu_markers.py - Different ways to define marker positions

```
from kapteyn import maputils, tabarray
from matplotlib import pyplot as plt
import numpy

f = maputils.FITSimage("m101.fits")
fig = plt.figure()
frame = fig.add_subplot(1,1,1)
annim = f.Annotatedimage(frame, cmap="binary")
annim.Image()
```

```
grat = annim.Graticule()
#annim.Marker(pos="210.80 deg 54.34 deg", marker='o', color='b')
annim.Marker(pos="pc", marker='o', markersize=10, color='r')
annim.Marker(pos="14h03m30 54d20m", marker='o', color='y')
annim.Marker(pos="ga 102.035415152 ga 59.772512522", marker='+',
             markersize=20, markeredgewidth=2, color='m')
annim.Marker(pos="{ecl,fk4,J2000} 174.367462651 {} 59.796173724",
             marker='x', markersize=20, markeredgewidth=2, color='g')
annim.Marker(pos="{eq,fk4-no-e,B1950,F24/04/55} 210.360200881 {} 54.587072397",
             marker='o', markersize=25, markeredgewidth=2, color='c',
             alpha=0.4)

# Use pos= keyword argument to enter sequence of
# positions in pixel coordinates. The syntax is described
# in the module positions.py
pos = "200+20*sin([100:199]/20), range(100,200)"

annim.Marker(pos=pos, marker='o', color='r')

# Use x= and y= keyword arguments to enter sequence of
# positions in pixel coordinates. Note that this is not parsed by
# module positions.py. Here we need list comprehension to
# get the same effect.
xp = [400+20*numpy.sin(x/20.0) for x in range(100,200)]
yp = range(100,200)
annim.Marker(x=xp, y=yp, mode='pixels', marker='o', color='g')

xp = yp = 150
annim.Marker(x=xp, y=yp, mode='pixels', marker='+', color='b')

annim.plot()
annim.interact_imagecolors()
annim.interact_toolbarinfo()
plt.show()
```

2.2.13 Sky polygons

Sometimes one needs to plot a shape which represents an area in the sky. Such a shape can be a small ellipse which represents the beam of a radio telescope or it is a rectangle representing a footprint of a survey. These shapes (ellipse, circle, rectangle, square, regular polygon) have a prescription. For example a circle is composed of a number of vertices with a constant distance to a center and all the vertices define a different angle. If you want to plot such a shape onto a projection of a sphere you need to recalculate the vertices so that for a given center (lon_c,lat_c) the set of distances and angles are preserved on the sphere. By distances we mean the distance along a great circle and not along a parallel.

So what we do is calculate vertices of an ellipse/rectangle/regular polygon in a plane and the center of the shape is at (0,0). For a sample of points on the shape we calculate the distance of the perimeter as function of the angle. Then with spherical trigonometry we solve for the missing (lon,lat) in the triangle (lon_c,lat_c)-Pole-(lon,lat). This is the position for which the distance along a great circle is the required one and also the angle is the required one.

Assume a triangle on a sphere connecting two positions *Q1* with $(\alpha_1, \delta 1)$ and *Q2* with $(\alpha_2, \delta 2)$ and on the sphere *Q2* is situated to the left of *Q1*. *Q1* is the new center of the polygon. *P* is the pole and *Q2* the position we need to find. This position has distance *d* along a great circle connecting *Q1* and *Q2* and the angle *PQ1Q2* is the required angle α. The sides of the triangle are $(90 - \delta_1)$ and $(90 - \delta_2)$

Then the distance between *Q1* and *Q2* is given by:

$$\cos(d) = \cos(90-\delta_1)\cos(90-\delta_2) + \sin(90-\delta_1)\sin(90-\delta_2)\cos(\alpha_2-\alpha_1) \tag{2.3}$$

from which we calculate $\cos(\alpha_2-\alpha_1)$

Angle *Q1PQ2* is equal to $\alpha_2-\alpha_1$. For this angle we have the sine formula:

$$\frac{\sin(d)}{\sin(\alpha_2-\alpha_1)} = \frac{\sin(90-\delta_2)}{\sin(\alpha)} \tag{2.4}$$

so that:

$$\sin(\alpha_2-\alpha_1) = \frac{\sin(d)\sin(\alpha)}{\cos(\delta_2)} \tag{2.5}$$

With $\cos(\alpha_2-\alpha_1)$ and the value of $\sin(\alpha_2-\alpha_1)$ we derive an unambiguous value for $\alpha_2-\alpha_1$ and because we started with α_1 we derive a value for α_2.

The angle *PQ1Q2* is α. This is not the astronomical convention, but that doesn't matter because we use the same definition for an angle in the 'flat space' polygon. For this situation we have another cosine rule:

$$\cos(90-\delta_2) = \cos(d)cos(90-\delta_1) + \sin(d)\sin(90-\delta_1)\cos(\alpha) \tag{2.6}$$

or:

$$\sin(\delta_2) = \cos(d)\sin(\delta_1) + \sin(d)\cos(\delta_1)\cos(\alpha) \tag{2.7}$$

which gives δ_2 and we found longitude and latitude (α_2, δ_2) of the transformed 'flat space' coordinate. The set of transformed vertices in world coordinates are then transformed to pixels which involves the projection of a map.

The next example shows some shapes plotted in a map of a part of the sky.

Example: mu_skypolygons.py - 'Sky polygons' in M101

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

f = maputils.FITSimage("m101.fits")

fig = plt.figure()
frame = fig.add_subplot(1,1,1)

annim = f.Annotatedimage(frame, cmap='gist_yarg')
annim.Image()
grat = annim.Graticule()
grat.setp_gratline(color='0.75')

# Ellipse centered on crossing of two graticule lines
annim.Skypolygon("ellipse", cpos="14h03m 54d20m", major=100, minor=50,
                  pa=-30.0, units='arcsec', fill=False)

# Ellipse at given pixel coordinates
annim.Skypolygon("ellipse", cpos="10 10", major=100, minor=50,
                  pa=-30.0, units='arcsec', fc='c')

# Circle with radius in arc minutes
annim.Skypolygon("ellipse", cpos="210.938480 deg 54.269206 deg",
                  major=1.50, minor=1.50, units='arcmin',
                  fc='g', alpha=0.3, lw=3, ec='r')

```

```
# Rectangle at the projection center
annim.Skypolygon("rectangle", cpos="pc pc", major=200, minor=50,
                  pa=30.0, units='arcsec', ec='g', fc='b', alpha=0.3)

# Regular polygon with 6 angles at some position in galactic coordinates
annim.Skypolygon("npoly", cpos="ga 102d11m35.239s ga 59d50m25.734",
                  major=150, nangles=6,
                  units='arcsec', ec='g', fc='y', alpha=0.3)

# Regular polygon
annim.Skypolygon("npolygon", cpos="ga 102.0354152 ga 59.7725125",
                  major=150, nangles=3,
                  units='arcsec', ec='g', fc='y', alpha=0.3)

lons = [210.969423, 210.984761, 210.969841, 210.934896, 210.894589,
        210.859949, 210.821008, 210.822413, 210.872040]
lats = [54.440575, 54.420249, 54.400778, 54.388611, 54.390166,
        54.396241, 54.416029, 54.436244, 54.454230]

annim.Skypolygon(prescription=None, lons=lons, lats=lats, fc='r', alpha=0.3)

annim.plot()
annim.interact_toolbarinfo()
annim.interact_imagecolors()
annim.interact_writepos(wcsfmt="%f",zfmt=None, pixfmt=None, hmsdms=False)

plt.show()
```

In ‘all sky’ plots the results can be a little surprising. For the family of cylindrical projections we give a number of examples.

Example: mu_skypolygons_allsky.py - ‘Sky polygons’ in a number of cylindrical projections

```
from kapteyn import maputils
from matplotlib import pyplot as plt
import numpy


def shapes(proj, fig, plnr, crval2=0.0, **pv):
   naxis1 = 800; naxis2 = 800
   header = {'NAXIS': 2,
             'NAXIS1': naxis1, 'NAXIS2': naxis2,
             'CRPIX1': naxis1/2.0, 'CRPIX2': naxis2/2.0,
             'CRVAL1': 0.0,    'CRVAL2': crval2,
             'CDELT1': -0.5,   'CDELT2': 0.5,
             'CUNIT1': 'deg',  'CUNIT2': 'deg',
             'CTYPE1': 'RA---%s'%proj, 'CTYPE2': 'DEC--%s'%proj}
   if len(pv):
      header.update(pv)

   print header
   X = numpy.arange(0,390.0,30.0); X[-1] = 180+0.00000001
   Y = numpy.arange(-90,91,30.0)
   f = maputils.FITSimage(externalheader=header)
   frame = fig.add_subplot(2,2,plnr)
   annim = f.Annotatedimage(frame)
   grat = annim.Graticule(axnum=(1,2),
                          wylim=(-90.0,90.0), wxlim=(-180,180),
                          startx=X, starty=Y)
   grat.setp_gratline(color='0.75')
   if plnr in [1,2]:
     grat.setp_axislabel(plotaxis='bottom', visible=False)
   print "Projection %d is %s" % (plnr, proj)
   # Ellipse centered on crossing of two graticule lines
   try:
      annim.Skypolygon("ellipse", cpos="5h00m 20d0m", major=50, minor=30,
                        pa=-30.0, fill=False)
   except:
      print "Failed to plot ellipse"
   # Ellipse at given pixel coordinates
   try:
      cpos = "%f %f"%(naxis1/2.0+20, naxis2/2.0+10)
      annim.Skypolygon("ellipse", cpos=cpos, major=20, minor=10,
                        pa=-30.0, fc='m')
   except:
      print "Failed to plot ellipse"
   # Circle with radius in arc minutes
   try:
      annim.Skypolygon("ellipse", cpos="0 deg 60 deg",
                     major=30, minor=30,
                     fc='g', alpha=0.3, lw=3, ec='r')
   except:
      print "Failed to plot circle"
   # Rectangle at the projection center
   try:
      annim.Skypolygon("rectangle", cpos="pc pc", major=50, minor=20,
                     pa=30.0, ec='g', fc='b', alpha=0.3)
   except:
      print "Failed to plot rectangle"
   # Square centered at 315 deg -45 deg and with size equal
   # to distance on sphere between 300,-30 and 330,-30 deg (=25.9)
   try:
      annim.Skypolygon("rectangle", cpos="315 deg -45 deg", major=25.9, minor=25.9,
                        pa=0.0, ec='g', fc='#ff33dd', alpha=0.8)
   except:
      print "Failed to plot square"
   # Regular polygon with 6 angles at some position in galactic coordinates
   try:
      annim.Skypolygon("npoly", cpos="ga 102d11m35.239s ga 59d50m25.734",
                        major=20, nangles=6,
                        ec='g', fc='y', alpha=0.3)
   except:
      print "Failed to plot regular polygon"
   # Regular polygon as a triangle
   try:
      annim.Skypolygon("npolygon", cpos="ga 0 ga 90",
                        major=70, nangles=3,
                        ec='g', fc='c', alpha=0.7)
   except:
      print "Failed to plot triangle"
   # Set of (absolute) coordinates, no prescription
   lons = [270, 240, 240, 270]
   lats = [-60, -60, -30, -30]
   try:
      annim.Skypolygon(prescription=None, lons=lons, lats=lats, fc='r', alpha=0.9)
   except:
      print "Failed to plot set of coordinates as polygon"

   grat.Insidelabels(wcsaxis=0,
                     world=range(0,360,30), constval=0, fmt='Hms',
                     color='b', fontsize=5)
   grat.Insidelabels(wcsaxis=1,
                     world=[-60, -30, 30, 60], constval=0, fmt='Dms',
                     color='b', fontsize=5)
   annim.interact_toolbarinfo()
   annim.interact_writepos(wcsfmt="%f",zfmt=None, pixfmt=None, hmsdms=False)
   frame.set_title(proj, y=0.8)
   annim.plot()

fig = plt.figure()
fig.subplots_adjust(left=0.03, bottom=0.05, right=0.97,
                    top=0.97, wspace=0.02, hspace=0.02)
shapes("AIT", fig, 1)
shapes("CAR", fig, 2)
shapes("BON", fig, 3, PV2_1=45)
shapes("PCO", fig, 4)
plt.show()
```

The code shows a number of lines with `try` and `except` clauses. This is to catch problems for badly chosen origins or polygon parameters. We also provide examples of similar polygons in a number of zenithal projections. The scaling is unaltered so different projections fill the plot differently.

Example: mu_skypolygons_zenith.py - 'Sky polygons' in a number of zenithal projections

```
from kapteyn import maputils
from matplotlib import pyplot as plt
import numpy

# This script shows that you can plot shapes that cross the pole.
# A shape is plotted with respect to its center and the border points
# are derived in a way that distance and angle are correct for a sphere.
# This makes it impossible to have objects centered at the pole because at
# the pole, longitudes are undefined. To avoid this problem, one can shift
# the center of such shapes a little as we have done with pcra and
# pcdec below.
# The try excepts in this program is to catch problems with special
# projections (e.g. NCP where dec > 0)

delta = 0.0001
pcra = delta
pcdec = 90. -delta

def shapes(proj, fig, plnr, crval2=0.0, **pv):
   naxis1 = 800; naxis2 = 800
   header = {'NAXIS': 2,
             'NAXIS1': naxis1, 'NAXIS2': naxis2,
             'CRPIX1': naxis1/2.0, 'CRPIX2': naxis2/2.0,
             'CRVAL1': 0.0,    'CRVAL2': crval2,
             'CDELT1': -0.5,   'CDELT2': 0.5,
             'CUNIT1': 'deg', 'CUNIT2': 'deg',
             'CTYPE1': 'RA---%s'%proj, 'CTYPE2': 'DEC--%s'%proj}
   if len(pv):
      header.update(pv)

   X = numpy.arange(0,390.0,30.0);
   Y = numpy.arange(-30,91,30.0)
```

```
    f = maputils.FITSimage(externalheader=header)
    frame = fig.add_subplot(2, 2, plnr)
    annim = f.Annotatedimage(frame)
    grat = annim.Graticule(axnum=(1,2),
                          wylim=(-30.0,90.0), wxlim=(-180,180),
                          startx=X, starty=Y)
    grat.setp_gratline(color='0.75')
    if plnr in [1,2]:
      grat.setp_axislabel(plotaxis='bottom', visible=False)
    print "Projection %d is %s" % (plnr, proj)
    # Ellipse centered on crossing of two graticule lines
    try:
       annim.Skypolygon("ellipse", cpos="5h00m 20d0m", major=50, minor=30,
                          pa=-30.0, fill=False)
       print "Plotted ellipse with cpos='5h00m 20d0m', major=50, minor=30, pa=-30.0, fill=False"
    except:
       print "Failed to plot ellipse"
    # Ellipse at given pixel coordinates
    try:
       cpos = "%f %f"%(naxis1/2.0+20, naxis2/2.0+10)
       annim.Skypolygon("ellipse", cpos=cpos, major=40, minor=10,
                          pa=0.0, fc='m')
       print "Plotted ellipse major=40, minor=10, pa=-30.0, fc='m'"
    except:
       print "Failed to plot ellipse"
    # Circle with radius in arc minutes
    try:
       annim.Skypolygon("ellipse", xc=pcra, yc = pcdec, #cpos="0 deg 60 deg",
                       major=30, minor=30,
                       fc='g', alpha=0.3, lw=3, ec='r')
       print "Plotted red circle, green with red border transparent"
    except:
       print "Failed to plot circle"
    # Rectangle at the projection center
    try:
       annim.Skypolygon("rectangle", xc=pcra, yc=pcdec, major=50, minor=20,
                       pa=30.0, ec='g', fc='b', alpha=0.3)
       print "Plotted blue rectangle at projection center"
    except:
       print "Failed to plot blue rectangle at projection center"
    # Square centered at 315 deg -45 deg and with size equal
    # to distance on sphere between 300,-30 and 330,-30 deg (=25.9)
    try:
       annim.Skypolygon("rectangle", cpos="315 deg -45 deg", major=25.9, minor=25.9,
                          pa=0.0, ec='g', fc='#ff33dd', alpha=0.8)
       print "Plotted square with color #ff33dd"
    except:
       print "Failed to plot square"
    # Regular polygon with 6 angles at some position in galactic coordinates
    try:
       annim.Skypolygon("npoly", cpos="ga 102d11m35.239s ga 59d50m25.734",
                          major=20, nangles=6,
                          ec='g', fc='y', alpha=0.3)
       print "Plotted npoly in yellow"
    except:
       print "Failed to plot regular polygon"
    # Regular polygon as a triangle
    try:
       annim.Skypolygon("npolygon", cpos="ga 0 ga 90",
                          major=70, nangles=3,
                          ec='g', fc='c', alpha=0.7)
       print "Plotted npoly triangle in cyan"
    except:
       print "Failed to plot triangle"
    # Set of (absolute) coordinates, no prescription
    lons = [270, 240, 240, 270]
    lats = [-30, -30, 0, 0]
    try:
       annim.Skypolygon(prescription=None, lons=lons, lats=lats, fc='r', alpha=0.9)
       print "Plotted polygon without prescription"
    except:
       print "Failed to plot set of coordinates as polygon"

    grat.Insidelabels(wcsaxis=0,
                       world=range(0,360,30), constval=0, fmt='Hms',
                       color='b', fontsize=5)
    grat.Insidelabels(wcsaxis=1,
                       world=[-60, -30, 30, 60], constval=0, fmt='Dms',
                       color='b', fontsize=5)
    annim.interact_toolbarinfo()
    annim.interact_writepos(wcsfmt="%f",zfmt=None, pixfmt=None, hmsdms=False)
    frame.set_title(proj, y=0.8)
    annim.plot()


 fig = plt.figure()
 fig.subplots_adjust(left=0.03, bottom=0.05, right=0.97,
                     top=0.97, wspace=0.02, hspace=0.02)

 shapes("STG", fig, 1, crval2=90)
 shapes("ARC", fig, 2, crval2=90)
 pvkwargs = {'PV2_0' : 0.05, 'PV2_1' : 0.975, 'PV2_2' : -0.807,
             'PV2_3' : 0.337, 'PV2_4' : -0.065,
             'PV2_5' : 0.01, 'PV2_6' : 0.003,' PV2_7' : -0.001}
 shapes("ZPN", fig, 3, crval2=90, **pvkwargs)
 shapes("NCP", fig, 4, crval2=90)
 #xi =  -1/numpy.sqrt(6); eta = 1/numpy.sqrt(6)
 #shapes("SIN", fig, 4, crval2=90, PV2_1=xi, PV2_2=eta)
 plt.show()
```

Note that some polygons could not be plotted for the NCP projection, simply because it is defined from declination 90 to 0 only. To avoid problems with divergence we limit the world coordinates to a declination of -30 degrees. The sky polygons are not aware of this limit and are plotted as long conversions between pixel- and world coordinates are possible. The ZPN example is a bit special. First, it is not possible to center a shape onto the pole (at least with the set of PV elements defined in the code) and second, we have a non zero PV2_0 element which breaks the relation between CRPIX and CRVAL.

For a detailed description of the input parameters of the used *Skypolygon()* method, read `maputils.Annotatedimage.Skypolygon()`.

2.2.14 Combining different plot objects

We arrived at a stage where one is challenged to apply different plot objects in one plot. Here is a practical example:

Example: mu_graticules.py - Combining plot with contours and a colorbar

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

f = maputils.FITSimage("m101.fits")
f.set_limits(pxlim=(50,440), pylim=(50,450))

fig = plt.figure(figsize=(8,5.5))
frame = fig.add_axes((0.05, 0.1, 0.8, 0.7))
fig.text(0.5, 0.96, "Combination of plot objects",
         horizontalalignment='center',
         fontsize=14, color='r')

annim = f.Annotatedimage(frame, clipmin=3000, clipmax=15000)
cont = annim.Contours(levels=range(8000,14000,1000))
cont.setp_contour(linewidth=1)
cont.setp_contour(levels=11000, color='g', linewidth=2)
cb = annim.Colorbar(clines=True, orientation='vertical', fontsize=8, linewidths=5)
gr = annim.Graticule()
gr.setp_ticklabel(wcsaxis=0, fmt='HMS')
ilab = gr.Insidelabels(color='b', ha='left')
ilab.setp_label(position='14h03m0s', fontsize=15)

# Plot a second graticule for the galactic sky system
gr2 = annim.Graticule(deltax=7.5/60, deltay=5.0/60,
                      skyout="galactic",
                      visible=True)
gr2.setp_axislabel(plotaxis=("top","right"), label="Galactic l,b",
                   color='g', visible=True)
gr2.setp_axislabel(plotaxis=("left","bottom"), visible=False)
gr2.set_tickmode(plotaxis=("top","right"), mode="Native")
gr2.set_tickmode(plotaxis=("left","bottom"), mode="NO")
gr2.setp_ticklabel(wcsaxis=(0,1), color='g')
gr2.setp_ticklabel(plotaxis='right', fmt='DMs')
gr2.setp_tickmark(plotaxis='right', markersize=8, markeredgewidth=2)
gr2.setp_gratline(wcsaxis=(0,1), color='g')
annim.Ruler(x1=120, y1=100, x2=120, y2=330, step=1/60.0)
r1 = annim.Ruler(pos1='ga 102d0m, 59d50m', pos2='ga 102d7m30s, 59d50m',
                 world=True, step=1/60.0)
r1.setp_line(color='#ff22ff', lw=6)
r1.setp_label(color='m')
annim.Pixellabels(plotaxis='top', va='top')
pl = annim.Pixellabels(plotaxis='right')
pl.setp_marker(color='c', markersize=10)
pl.setp_label(color='m')

annim.plot()
plt.show()
```

2.2.15 External headers and/or data

You are not restricted to FITS files to get plots of your data. The only requirement is that you must be able to get your data into a NumPy array and you need to supply a Python dictionary with FITS keywords. For both options we show an example.

Example: mu_externalheader.py - Header data from Python dictionary and setting a figure size

```
from kapteyn import maputils
from matplotlib import pylab as plt
import numpy

header = {'NAXIS' : 2, 'NAXIS1': 800, 'NAXIS2': 800,
          'CTYPE1' : 'RA---TAN',
          'CRVAL1' :0.0, 'CRPIX1' : 1, 'CUNIT1' : 'deg', 'CDELT1' : -0.05,
          'CTYPE2' : 'DEC--TAN',
          'CRVAL2' : 0.0, 'CRPIX2' : 1, 'CUNIT2' : 'deg', 'CDELT2' : 0.05,
         }

# Overrule the header value for pixel size in y direction
header['CDELT2'] = 0.3*abs(header['CDELT1'])
fitsobj = maputils.FITSimage(externalheader=header)
figsize = fitsobj.get_figsize(ysize=7, cm=True)

fig = plt.figure(figsize=figsize)
print "Figure size x, y in cm:", figsize[0]*2.54, figsize[1]*2.54
frame = fig.add_subplot(1,1,1)
mplim = fitsobj.Annotatedimage(frame)
gr = mplim.Graticule()
mplim.plot()

plt.show()
```

Example: mu_externaldata.py - Using external FITS header and data

```
from kapteyn import maputils
from matplotlib import pylab as plt
import numpy

header = {'NAXIS'  : 2, 'NAXIS1': 800, 'NAXIS2': 800,
          'CTYPE1' : 'RA---TAN',
          'CRVAL1' : 0.0, 'CRPIX1' : 1, 'CUNIT1' : 'deg', 'CDELT1' : -0.05,
          'CTYPE2' : 'DEC--TAN',
          'CRVAL2' : 0.0, 'CRPIX2' : 1, 'CUNIT2' : 'deg', 'CDELT2' : 0.05,
         }

nx = header['NAXIS1']
ny = header['NAXIS2']
sizex1 = nx/2.0; sizex2 = nx - sizex1
sizey1 = nx/2.0; sizey2 = nx - sizey1
x, y = numpy.mgrid[-sizex1:sizex2, -sizey1:sizey2]
edata = numpy.exp(-(x**2/float(sizex1*10)+y**2/float(sizey1*10)))

f = maputils.FITSimage(externalheader=header, externaldata=edata)
f.writetofits()
fig = plt.figure(figsize=(6,5))
frame = fig.add_axes([0.1,0.1, 0.82,0.82])
mplim = f.Annotatedimage(frame, cmap='pink')
mplim.Image()
gr = mplim.Graticule()
gr.setp_gratline(color='y')
mplim.plot()

mplim.interact_toolbarinfo()
mplim.interact_imagecolors()
mplim.interact_writepos()
```

```

plt.show()
```

Note: Manipulated headers and data can be written to a FITS file with method `maputils.FITSimage.writetofits()`. Its documentation shows how to manipulate the format in which the data is written. Also have a look at this example which creates a FITSobject from an external header and external data. It then writes the object to a FITS file. The first time in the original data format with the original comments and history cards. The second time it writes to a file with BITPIX=-32 and it skips all comment and history information:

```
from kapteyn import maputils

fitsobject = maputils.FITSimage(promptfie=maputils.prompt_fitsfile)
header = fitsobject.hdr
edata = fitsobject.dat
f = maputils.FITSimage(externalheader=header, externaldata=edata)

f.writetofits(history=True, comment=True,
              bitpix=fitsobject.bitpix,
              bzero=fitsobject.bzero,
              bscale=fitsobject.bscale,
              blank=fitsobject.blank)

f.writetofits("standard.fits", history=False, comment=False)

# or use parameter append to append to an existing file:
f.writetofits("existing.fits", append=True, history=False, comment=False)
```

We use the method with external headers also to create all sky maps. In the next example we demonstrate how a graticule is created for an all sky map. You will find examples about plotting data in these plots in the document about all sky maps.

Example: mu_allsky_single.py - Using Python dictionary to define all-sky map

```
from kapteyn import maputils
from numpy import arange
from matplotlib import pyplot as plt

dec0 = 89.9999999999   # Avoid plotting on the wrong side
header = {'NAXIS'  : 2,
          'NAXIS1' : 100, 'NAXIS2': 80,
          'CTYPE1' : 'RA---TAN',
          'CRVAL1' : 0.0, 'CRPIX1' : 50, 'CUNIT1' : 'deg',
          'CDELT1' : -5.0, 'CTYPE2' : 'DEC--TAN',
          'CRVAL2' : dec0, 'CRPIX2' : 40,
          'CUNIT2' : 'deg', 'CDELT2' : 5.0,
         }
X = arange(0,360.0,15.0)
Y = [20, 30,45, 60, 75, 90]

fig = plt.figure(figsize=(7,6))
frame = fig.add_axes((0.1,0.1,0.8,0.8))
f = maputils.FITSimage(externalheader=header)
annim = f.Annotatedimage(frame)
grat = annim.Graticule(wylim=(20.0,90.0), wxlim=(0,360), startx=X, starty=Y)
lon_world = range(0,360,30)
lat_world = [20, 30, 60, dec0]
```

```python
grat.setp_gratline(position=20, color='g', linestyle='--')

# Plot labels inside the plot
lon_constval = None
lat_constval = 18
il1 = grat.Insidelabels(wcsaxis=0,
                  world=lon_world, constval=lat_constval, fmt='Dms')
il1.setp_label(color='r', fontsize=15)
il2 = grat.Insidelabels(wcsaxis=1, deltapy=2,
                  world=lat_world, constval=lon_constval, fmt='Dms')
il2.setp_label(color='b', fontsize=10)
annim.plot()

# Set title for Matplotlib
title = r"Gnomonic projection (TAN) diverges at $\theta=0$. (Cal. fig.8)"
frame.set_title(title, color='g', y=1.02)

plt.show()
```

The data from a FITS file is stored in a NumPy array. Then it is straightforward to maniplate this data. NumPy has many methods for this. We apply a Fourier transform to an image of M101. We show how to use functions *abs* and *angle* with a complex array as argument to get images of the amplitude and the fase of the transform. With the transform we test the inverse procedure and show the residual. There seems to be some systematic structure in the residual map but notice that the maximum is very small compared to the smallest image value in the original (which is around 1500). We used NumPy's FFT functions to calculate the transform. Have a look at the code:

Example: mu_fft.py - Mosaic of plots showing FFT of image data and inverse transform

```python
from kapteyn import maputils
from matplotlib import pyplot as plt
from numpy import fft, log, abs, angle

f = maputils.FITSimage("m101.fits")

yshift = -0.1
fig = plt.figure(figsize=(8,6))
fig.subplots_adjust(left=0.01, bottom=0.1, right=1.0, top=0.98,
                    wspace=0.03, hspace=0.16)
frame = fig.add_subplot(2,3,1)
frame.text(0.5, yshift, "M101", ha='center', va='center',
           transform = frame.transAxes)
mplim = f.Annotatedimage(frame, cmap="spectral")
mplim.Image()

fftA = fft.rfft2(f.dat, f.dat.shape)
frame = fig.add_subplot(2,3,2)
frame.text(0.5, yshift, "Amplitude of FFT", ha='center', va='center',
           transform = frame.transAxes)
f = maputils.FITSimage("m101.fits", externaldata=log(abs(fftA)+1.0))
mplim2 = f.Annotatedimage(frame, cmap="gray")
im = mplim2.Image()

frame = fig.add_subplot(2,3,3)
frame.text(0.5, yshift, "Phase of FFT", ha='center', va='center',
           transform = frame.transAxes)
f = maputils.FITSimage("m101.fits", externaldata=angle(fftA))
mplim3 = f.Annotatedimage(frame, cmap="gray")
im = mplim3.Image()
```

```

frame = fig.add_subplot(2,3,4)
frame.text(0.5, yshift, "Inverse FFT", ha='center', va='center',
           transform = frame.transAxes)
D = fft.irfft2(fftA)
f = maputils.FITSimage("m101.fits", externaldata=D.real)
mplim4 = f.Annotatedimage(frame, cmap="spectral")
im = mplim4.Image()

frame = fig.add_subplot(2,3,5)
Diff = D.real - mplim.data
f = maputils.FITSimage("m101.fits", externaldata=Diff)
mplim5 = f.Annotatedimage(frame, cmap="spectral")
im = mplim5.Image()

frame.text(0.5, yshift, "M101 - inv. FFT", ha='center', va='center',
           transform = frame.transAxes)
s = "Residual with min=%.1g max=%.1g" % (Diff.min(), Diff.max())
frame.text(0.5, yshift-0.08, s, ha='center', va='center',
           transform = frame.transAxes, fontsize=8)

mplim.interact_imagecolors()
mplim2.interact_imagecolors()
mplim3.interact_imagecolors()
mplim4.interact_imagecolors()
mplim5.interact_imagecolors()

maputils.showall()
```

Fig. mu_fft.py - FFT: another use of external data

The example shows that we can use external data with the correct shape in combination with the original FITS header. Note that we used Matplotlib's *text()* method instead of *xlabel()*. The reason is that the primary frame has all its axes set to invisible. We can set them to visible but besides a label, one also get numbers along the axes and that was what we want to avoid.

2.2.16 Re-projections and image overlays

A simple example

There are several methods to compare data of the same part of the sky but from different sources. These different sources often have different world coordinate systems. If you want to compare two or more sources in one plot you need to define a base world coordinate system (wcs) and adjust the other sources so that their data fits the header description of the first. In other words: you need to re-project the data of the other sources. Module `wcs` provides a powerful coordinate transformation function called `wcs.coordmap()` which does the necessary coordinate transformations between two wcs systems. The transformation of the data is done with an interpolation function based Scipy's function `map_coordinates`. The two functions are combined in method `maputils.FITSimage.reproject_to()`. If you have a FITS data structure that contains more than one spatial map (in the same hdu), then the method will re-project all these maps to a new spatial structure given in a second FITSimage object. This is demonstrated in the next example

Example: mu_reproj.py - Use second FITSimage object to define re-projection

```
from kapteyn import maputils

# Read first image as base
Basefits = maputils.FITSimage("ra_pol_freq_dec.fits")
```

```
# Get data from a FITS file. This is the data that
# should be reprojected to fit the header of Basefits.
Secondfits = maputils.FITSimage("dec_pol_freq_ra.fits")

# Now we want to re-project the data of the Base object onto
# the wcs of the second object. This is done with the
# reproject_to() method of the first FITSimage object
# (the data object) with the header of the second FITSimage
# object as parameter. This results in a new FITSimage object
Reprojfits = Basefits.reproject_to(Secondfits.hdr)

# Write the result to disk
Reprojfits.writetofits("reproj.fits", clobber=True)
```

Note that the result has the same structure for all non spatial axes, while the spatial information is copied from the second object.

If you want only a selection of all the available spatial maps, then you can restrict the re-projection with parameters *plimlo* and *plimhi*. These parameters are single pixel coordinates or tuples with pixel coordinates and each pixel coordinate represents a position on a non-spatial axis (a repeat axis) similar to the definition of a slice. Also it is possible to set the pixel limits of the output spatial structure with *pxlim* and *pylim*. Note that these correspond to the axis order of the spatial map to which we want to re-project. With these parameters it is easy to extend a map e.g. so that it contains a rotated map without cutting the edges. For all these procedures, the appropriate values of CRPIX in the header are adjusted so that the output header describes a valid wcs.

Below we show how to decrease the output size for the spatial axes. Also we require two maps in the output: the first is at POL=1 and FREQ=7 and the second is at POL=1 and FREQ=8. Note that *plimlo* and *plimhi* describe contiguous ranges!

```
>>> Reprojfits = Basefits.reproject_to(Secondfits.hdr,
                                       pxlim=(3,30), pylim=(3,20),
                                       plimlo=(1,7), plimhi=(1,8))
```

You can also expand the output for the spatial maps by entering values outside the default ranges [1, NAXIS]. Negative values are allowed to expand beyond pixel coordinate 1. The next code fragment shows this for all spatial maps at POL=1 (i.e. for all pixels coordinates on the FREQ axis).

```
>>> Reprojfits = Basefits.reproject_to(Secondfits.hdr,
                                       pxlim=(-10,50), pylim=(-10,50),
                                       plimlo=1, plimhi=1)
```

Re-projecting to an adjusted header

As an alternative to re-projecting to an existing header of a different wcs, one can also make a copy of a header and adjust it by making changes to existing keywords or to add new keyword, value pairs. This is one of the more common applications for re-projection purposes. For instance, one can change the header value for CROTA (corresponding to the latitude axis of an image) to rotate a map. Or one can re-project to another projection e.g. from a Gnomonic projection (TAN) to a Mercator projection (MER). This is what we have done in the next script. In the familiar M101 FITS file, we increased the pixel sizes with a factor of 100 to demonstrate the effect of the re-projection.

There are two practical problems we have to address:

- The CRVAL's for a Mercator projection must be 0.0. If we don't change them, the projection will be oblique.
- We don't know how big the result must be (in terms of pixels) to fit the result.

These problems are solved by creating an intermediate FITSimage object with the new header where CRVAL is set to 0 and where the values of CTYPE were changed. Then the pixel positions of the border of the original image are used to find world coordinates in the original image and from these world coordinates we derive pixel coordinates in the new projection. From these positions we take the minimum and maximum to extend the output box so that it can hold the entire image without any clipping.

We end up with the rectangular system that we are used to from a Mercator projection. Note that the image is subject to a small rotation.

Example: mu_m1012mercator.py - Transforming a map to another projection

```python
from kapteyn import maputils
from matplotlib import pyplot as plt
import numpy as np

Basefits = maputils.FITSimage("m101big.fits")
hdr = Basefits.hdr.copy()

hdr['CTYPE1'] = 'RA---MER'
hdr['CTYPE2'] = 'DEC--MER'
hdr['CRVAL1'] = 0.0
hdr['CRVAL2'] = 0.0
naxis1 = Basefits.hdr['NAXIS1']
naxis2 = Basefits.hdr['NAXIS2']

# Sample the border in pixels
x = np.concatenate([np.arange(1, naxis1+1), naxis1*np.ones(naxis1),
                    np.arange(1, naxis1+1), np.ones(naxis1)])
y = np.concatenate([np.ones(naxis2), np.arange(1, naxis2+1),
                    naxis2*np.ones(naxis1), np.arange(1, naxis2+1)])
x, y = Basefits.proj.toworld((x,y))


# Create a dummy object to calculate pixel coordinates of border in the new system.
f = maputils.FITSimage(externalheader=hdr)
px, py = f.proj.topixel((x,y))
pxlim = [int(min(px))-1, int(max(px))+1]
pylim = [int(min(py))-1, int(max(py))+1]
# print "New limits:", pxlim, pylim

Reprojfits = Basefits.reproject_to(hdr, pxlim_dst=pxlim, pylim_dst=pylim)
#Reprojfits.writetofits("reproj.fits", clobber=True)

fig = plt.figure(figsize=(9,5))
frame1 = fig.add_subplot(1,2,1)
frame2 = fig.add_subplot(1,2,2)
im1 = Basefits.Annotatedimage(frame1)
im2 = Reprojfits.Annotatedimage(frame2)
im1.Image(); im1.Graticule()
im2.Image(); im2.Graticule()
im1.plot(); im2.plot()
fig.tight_layout()
plt.show()
```

Re-projecting to an adjusted header II

In the previous section we have dealt with a change in the projection only and we needed to set explicit values for CRVAL. But what if we want to change both projection system and sky system. For example, we want to transform our M101 FITS file to a map with a Galactic sky system and we want to change the projection type from `TAN` to `SIN`. Now we need an extra step to find the values for CRVAL in the galactic system that correspond to the values of CRVAL in the original file. We do this with class `wcs.Transformation` which transforms world coordinates from one celestial system to the other.

We apply the same trick with the border pixels, but now we need to convert the world coordinates of the border to world coordinates in the Galactic system.

Example: mu_skyAndProj.py - Transforming a map to another projection AND another sky system

```
from kapteyn import maputils, wcs
from matplotlib import pyplot as plt
import numpy as np

# The data you want to re-project
Basefits = maputils.FITSimage("m101.fits")
hdrIn = Basefits.hdr
projIn = Basefits.proj
crvalsI = Basefits.proj.crval

# We want to change the sky system, so we need to know the values of CRVAL in the new system
trans = wcs.Transformation(projIn.skysys, skyout=wcs.galactic)
crvalsO = trans(crvalsI)

# Adjust the new header which was derived from the input
hdrOut = hdrIn.copy()
hdrOut['CTYPE1'] = 'GLON-SIN'
hdrOut['CTYPE2'] = 'GLAT-SIN'
hdrOut['CRVAL1'] = crvalsO[0]
hdrOut['CRVAL2'] = crvalsO[1]

# Get an estimate of the new corners by converting ALL boundary pixels of the input map
naxis1 = hdrIn['NAXIS1']
naxis2 = hdrIn['NAXIS2']
x = np.concatenate([np.arange(1, naxis1+1), naxis1*np.ones(naxis1),
                    np.arange(1, naxis1+1), np.ones(naxis1)])
y = np.concatenate([np.ones(naxis2), np.arange(1, naxis2+1),
                    naxis2*np.ones(naxis1), np.arange(1, naxis2+1)])
x, y = projIn.toworld((x,y))

# But we need Galactic coordinates  before converting to pixels
# in the new system, not the original cooordinates:
x, y = trans((x,y))

# Create a dummy object to calculate pixel coordinates in the new system.
f = maputils.FITSimage(externalheader=hdrOut)
px, py = f.proj.topixel((x,y))
pxlim = [int(min(px))-1, int(max(px))+1]
pylim = [int(min(py))-1, int(max(py))+1]
#print "New limits:", pxlim, pylim

# Do the re-projection
Reprojfits = Basefits.reproject_to(hdrOut, pxlim_dst=pxlim, pylim_dst=pylim)
#Reprojfits.writetofits("reproj.fits", clobber=True)
crpixs = Reprojfits.proj.crpix
crvals = Reprojfits.proj.toworld(crpixs)
#print "Check that crpix values in output are adjusted correctly:"
#print "Crvals from initial transformation and from re-projection:", crvals, crvalsO

# Some plotting to check the result
fig = plt.figure(figsize=(9,5))
```

```
frame1 = fig.add_subplot(1,2,1)
frame2 = fig.add_subplot(1,2,2)
im1 = Basefits.Annotatedimage(frame1)
im2 = Reprojfits.Annotatedimage(frame2)
im1.Image(); im1.Graticule()
im2.Image(); im2.Graticule()
im1.interact_toolbarinfo()
im2.interact_toolbarinfo()
im1.plot(); im2.plot()
fig.tight_layout()
plt.show()
```

Transforming a WCS with CD or PC elements to a classic header

To facilitate legacy FITS readers which cannot process CD and PC elements we wrote a method that converts headers to classic headers, i.e. with the familiar header keywords CRVAL, CRPIX, CDELT, CROTA. When a CD or PC matrix is encountered, and the non diagonal elements are not zero, then skew can be expected. One derives then two values for the image rotation. This method averages these values as a 'best value' for CROTA. (see also section 6 in paper II by Calabretta & Greisen).

Example: mu_reproj2classic.py - Create a 'classic' header without CD or PC elements

```
from kapteyn import maputils, wcs
from math import *       # To support expression evaluation with 'eval()'



Basefits = maputils.FITSimage(promptfie=maputils.prompt_fitsfile)

classicheader, skew, hdrchanged = Basefits.header2classic()
if  hdrchanged:
   print "Original header:\n", Basefits.str_header()
   if skew != 0.0:
      print "Found two different rotation angles. Difference is %f deg." % skew
else:
   print "Header probably already 'classic'. Nothing changed"

print  """You can copy the data and replace the header by the classic header
or you can re-project it to get rid of skew or to rotate the data
using a rotation angle (keyword CROTAn=)."""

ok = raw_input("Do you want to remove skew or rotate image ... [Y]/N: ")
if ok in ['y', 'Y', '']:
   lat = Basefits.proj.lataxnum
   key = "CROTA%d"%lat
   crotaold = classicheader[key] # CROTA Is always available in this header
   mes = "Enter value for CROTA%d ... [%g]: " %(lat, crotaold)
   newcrota = raw_input(mes)
   if newcrota != '':
      crota = eval(newcrota)
      classicheader[key] = crota
   print "Classic header voor reproject:"
   print classicheader
   print "\n Re-projecting ..."
   fnew = Basefits.reproject_to(classicheader, insertspatial=False)
else:
   # A user wants to replace the header only. Leave data untouched.
   fnew = maputils.FITSimage(externalheader=classicheader,
                             externaldata=Basefits.dat)

filename_out = "classic.fits"
# ADD (!) to 'classic.fits'
print "A copy of the selected hdu in the FITS file is APPENDED to [%s] on disk"%filename_out
fnew.writetofits(filename_out, clobber=True, append=True)
```

Change FITS keywords to change current image

Method `maputils.FITSimage.reproject_to()` recognizes three input types for parameter *reprojobj*. We demonstrated the use of a FITSimage object and a header. It is also possible to set its value to *None*. Then the header of the current object is used to define the transformation. If you don't want to get the header and change it in your script (as in a previous example) then one can use keyword parameters with the same name as FITS keywords to change this current header. In the next example we show how to rotate an image using the *rotation* parameter and increase the size of the image using keyword parameters. It also shows a hint how to align an image with the direction of the north, which combines the use of parameter *rotation* to create a header for which *CROTAn* defines the rotation of the image and keyword *CROTA2* to set this header value to 0.0.

Example: mu_simplereproj.py - Rotate an image using keyword parameters

```
from kapteyn import maputils

Basefits = maputils.FITSimage(promptfie=maputils.prompt_fitsfile)
Rotfits = Basefits.reproject_to(rotation=40.0, naxis1=800, naxis2=800,
                                crpix1=400, crpix2=400)

# If you want alignment with direction of the north, use:
# Rotfits = Basefits.reproject_to(rotation=0.0, crota2=0.0)

# If copy on disk required:
# Rotfits.writetofits("aligned.fits", clobber=True, append=False)

annim = Rotfits.Annotatedimage()
annim.Image()
annim.Graticule()
annim.interact_toolbarinfo()
maputils.showall()
```

Re-projections for overlay

Re-projections are often used to enable the comparison of data of two different sources (i.e. with different world coordinate systems) in one plot. Then usually contours of a second FITS image are used upon an image of a base FITS image. The situation is a bit different compared to the examples above. We need only one spatial map to be re-projected and this spatial map is set by a slice (i.e. pixel positions on the repeat axes). The pixel limits (box) of the spatial axes are set by the first FITS image. Instead of a header we can use the *FITSimage* object to which we want to re-project as a parameter. Then all appropriate information is passed and the `maputils.FITSimage.reproject_to()` method returns a new FITSimage object with only one spatial map with sizes that fits the first spatial map. The attribute *boxdat* can be used to replace the contents of the first image using the *boxdat* parameter in method `maputils.FITSimage.Annotatedimage()`.

The example script below shows how this is implemented in a script. The situation is not very complicated because we deal with two 2-dimensional data structures. Note the use of histogram equalization to enhance some details.

Example: mu_overlayscuba.py - Overlay image with different world coordinate system

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

file1 = "scuba850_AFGL2591.fits"
file2 = "13CO_3-2_integ_regrid.fits"

# Read first image as base
Basefits = maputils.FITSimage(file1)
Secondfits = maputils.FITSimage(file2)
Reprojfits = Secondfits.reproject_to(Basefits)

fig = plt.figure()
frame = fig.add_subplot(1,1,1)

baseim = Basefits.Annotatedimage(frame)
baseim.Image()
baseim.set_histogrameq()
baseim.Graticule()

overlayim = Basefits.Annotatedimage(frame, boxdat=Reprojfits.boxdat)
levels = range(20,200,20)
overlayim.Contours(levels=levels, colors='w')

baseim.plot()
overlayim.plot()
baseim.interact_toolbarinfo()
baseim.interact_imagecolors()

plt.show()
```

It is also possible to mix two images using an alpha factor smaller than 1.0. That is what we did in the next example. The overlay image is smaller than the base image. Then the overlay is padded with NaN's which are not transparent. We can change the values for pixels that could not be interpolated from NaN to another value with parameter *cval* which is part of a dictionary with keywords and values to control the interpolation routine in `maputils.FITSimage.reproject_to()`. We also set the interpolation order to 1 (which is the default set by maputils). This order represents a bilinear interpolation.

Example: mu_overlayscuba_im.py - Overlay image with different world coordinate system, using transparancy factor

```python
from kapteyn import maputils
from matplotlib import pyplot as plt

file1 = "scuba850_AFGL2591.fits"
file2 = "13CO_3-2_integ_regrid.fits"

# Read first image as base
Basefits = maputils.FITSimage(file1)
Secondfits = maputils.FITSimage(file2)

pars = dict(cval=0.0, order=1)
Reprojfits = Secondfits.reproject_to(Basefits, interpol_dict=pars)

fig = plt.figure()
frame = fig.add_subplot(1,1,1)

baseim = Basefits.Annotatedimage(frame)
baseim.Image(alpha=1.0)
baseim.set_histogrameq()
baseim.set_blankcolor('k')
baseim.Graticule()

overlayim = Basefits.Annotatedimage(frame, cmap='OrRd',
                                    boxdat=Reprojfits.boxdat)
levels = range(50,200,20)
#overlayim.Contours(levels=levels, colors='w')
overlayim.Image(alpha=0.8)
baseim.set_histogrameq()

baseim.plot()
overlayim.plot()
baseim.interact_toolbarinfo()
baseim.interact_imagecolors()

plt.show()
```

The base image has scheduled a function to interactively change its colors while the second image remains fixed. This enables you do compare the two images.

Combining an all-sky image with a user defined projection system

We like to add an example where we combine an all-sky image with an all-sky graticule. If your goal is to plot an all-sky image with a graticules overlay then you have to use the method that plots labels inside the image area to get the best results. This method is from class Annotatedimage and is called Insidelabels(), see description at: `wcsgrat.Graticule.Insidelabels()`

But your mission is not always straightforward as it seems. For instance, when you want to plot a map in a different sky system and/or projection system, you need to re-project the image first. Here we present an example where we start with an all-sky map in galactic coordinates but without a projection system defined in its header. We want to show the data in an oblique Aitof projection.

First obtain an all sky map. This is what we have done for our example: Go to the Bonn 408-MHz All-Sky Survey

You need to enter information on a form. First, there is no need to enter coordinates for the center. For the map size in x and y take 360 180 (degrees), i.e. the whole sky For the tabular interval, take the original values (i.e. enter -1 -1) Select Galactic coordinates as the sky system and pixmap (no projection) as the projection system. In 'Select a survey' we selected 'All sky 408 Mhz'. Submit form and wait for the map to appear. Then download the FITS file (one of the buttons below the image).

In the script we change this FITS file's CTYPE's to 'GLON-CAR' and 'GLAT-CAR'. WCS projection CAR is a valid WCSLIB rectangular projection. We can do this safely because the values of CRVAL are 0.0 and they represent the center of the image (CRPIX=NAXIS/2). So the standard parallel is the equator. Now the previously linear projection has a WCS equivalent and we can use it for our reprojections. If you wonder why we needed to apply this trick, then you should try the example without adding the projection type 'CAR'. You will end up with only one half of the entire sky. The reason is that we sample the output in pixels. These pixels are converted into world coordinates of the output image. But these world coordinates are between 0 and 360 degrees in longitude. If you enter these coordinates in the linear system, you get pixels outside the range of pixels for the input set because there the range in longitude is between -180 and 180 degrees (CRVAL's are in the center) and a linear system does not wrap around.

Example: mu_allsky_reproj.py - Reproject all-sky map to fit graticules of self defined projection system

```
import numpy
from kapteyn import maputils
from matplotlib.pyplot import show, figure
import csv    # Read some poitions from file in Comma Separated Values format
```

```

# Some initializations
blankcol = "#334455"                  # Represent undefined values by this color
epsilon = 0.0000000001
figsize = (9,7)                       # Figure size in inches
plotbox = (0.1,0.05,0.8,0.8)
fig = figure(figsize=figsize)
frame = fig.add_axes(plotbox)

Basefits = maputils.FITSimage("allsky_raw.fits")  # Here is your downloaded FITS file in rectangular
Basefits.hdr['CTYPE1'] = 'GLON-CAR'               # For transformations we need to give it a project
Basefits.hdr['CTYPE2'] = 'GLAT-CAR'               # CAR is rectangular

# Use some header values to define reprojection parameters
cdelt1 = Basefits.hdr['CDELT1']
cdelt2 = Basefits.hdr['CDELT2']
naxis1 = Basefits.hdr['NAXIS1']
naxis2 = Basefits.hdr['NAXIS2']

# Header works only with a patched wcslib 4.3
# Note that changing CRVAL1 to 180 degerees, shifts the plot 180 deg.
header = {'NAXIS'  : 2, 'NAXIS1': naxis1, 'NAXIS2': naxis2,
          'CTYPE1' : 'GLON-AIT',
          'CRVAL1' : 0, 'CRPIX1' : naxis1//2, 'CUNIT1' : 'deg', 'CDELT1' : cdelt1,
          'CTYPE2' : 'GLAT-AIT',
          'CRVAL2' : 30.0, 'CRPIX2' : naxis2//2, 'CUNIT2' : 'deg', 'CDELT2' : cdelt2,
          'LONPOLE' :60.0,
          'PV1_1'  : 0.0, 'PV1_2' : 90.0,  # IMPORTANT. This is a setting from Cal.section 7.1, p 11
         }
Reprojfits = Basefits.reproject_to(header)
annim_rep = Reprojfits.Annotatedimage(frame)
annim_rep.set_colormap("heat.lut")               # Set color map before creating Image object
annim_rep.set_blankcolor(blankcol)               # Background are NaN's (blanks). Set color here
annim_rep.Image(vmin=30000, vmax=150000)         # Just a selection of two clip levels
annim_rep.plot()

# Draw the graticule, but do not cover near -90 to prevent ambiguity
X = numpy.arange(0,390.0,15.0);
Y = numpy.arange(-75,90,15.0)
f = maputils.FITSimage(externalheader=header)
annim = f.Annotatedimage(frame)
grat = annim.Graticule(axnum= (1,2), wylim=(-90,90.0), wxlim=(0,360),
                       startx=X, starty=Y)
grat.setp_lineswcs0(0, color='w', lw=2)
grat.setp_lineswcs1(0, color='w', lw=2)

# Draw border with standard graticule, just to make the borders look smooth
header['CRVAL1'] = 0.0
header['CRVAL2'] = 0.0
del header['PV1_1']
del header['PV1_2']
header['LONPOLE'] = 0.0
header['LATPOLE'] = 0.0
border = annim.Graticule(header, axnum= (1,2), wylim=(-90,90.0), wxlim=(-180,180),
                         startx=(180-epsilon, -180+epsilon), skipy=True)
border.setp_lineswcs0(color='w', lw=2)   # Show borders in arbitrary color (e.g. background color)
border.setp_lineswcs1(color='w', lw=2)

```

```
# Plot the 'inside' graticules
lon_constval = 0.0
lat_constval = 0.0
lon_fmt = 'Dms'; lat_fmt = 'Dms'  # Only Degrees must be plotted
addangle0 = addangle1=0.0
deltapx0 = deltapx1 = 1.0
labkwargs0 = {'color':'r', 'va':'center', 'ha':'center'}
labkwargs1 = {'color':'r', 'va':'center', 'ha':'center'}
lon_world = range(0,360,30)
lat_world = [-60, -30, 30, 60]

ilabs1 = grat.Insidelabels(wcsaxis=0,
                     world=lon_world, constval=lat_constval,
                     deltapx=1.0, deltapy=1.0,
                     addangle=addangle0, fmt=lon_fmt, **labkwargs0)
ilabs2 = grat.Insidelabels(wcsaxis=1,
                     world=lat_world, constval=lon_constval,
                     deltapx=1.0, deltapy=1.0,
                     addangle=addangle1, fmt=lat_fmt, **labkwargs1)

# Read marker positions (in 0h0m0s 0d0m0s format) from file
reader = csv.reader(open("positions.txt"), delimiter=' ',  skipinitialspace=True)
for line in reader:
    if line:
       hms, dms = line
       postxt = "{eq fk4-no-e} "+hms+" {} "+dms   # Define the sky system of the source
       print postxt
       annim.Marker(pos=postxt, marker='*', color='yellow', ms=20)


# Plot a title
titlepos = 1.02
title = r"""All sky map in Hammer Aitoff projection (AIT) oblique with:
$(\alpha_p,\delta_p) = (0^\circ,30^\circ)$, $\phi_p = 75^\circ$ also:
$(\phi_0,\theta_0) = (0^\circ,90^\circ)$."""
t = frame.set_title(title, color='g', fontsize=13, linespacing=1.5)
t.set_y(titlepos)

annim.plot()
annim.interact_toolbarinfo()
annim_rep.interact_imagecolors()
show()
```

In the plot we marked some positions with a yellow star. The method that we use here to read positions from a file is different compared to previous examples. That is because we have a file where the positions are represented by strings. So we implemented some code to read these positions. The positions must follow the syntax for positions as described in: <no title>. The contents of the text file with some marker positions is:

```
17h45m40.0409s   -29d0m28.118s
12h49m0s         27d24m
17h42m37s        -28d57m0s
3h10m07s  20d10m0s
3h11m07s  21d10m0s
3h12m07s  22d10m0s
3h13m07s  23d10m0s
3h14m07s  24d10m0s
3h15m07s  25d10m0s
```

As you can see, the positions are equatorial. But there is no way for the program to detect which sky- and reference system is used for the positions. Therefore we iterate over all position strings and add the right specification. In our example this is {eq fk4-no-e}.

Improving the quality of the re-projection

The interpolation routine in the Kapteyn Package is based on SciPy's `map_coordinates()`. This function has a parameter *order* which sets the interpolation mode. In the script below we create a contour overlay using a rotated version of a base image (also the pixel size differs). This version is re-projected onto the first. The difference map is used to calculate a mean and a standard deviation of the residual. In a table we show the calculated values as function of the interpolation order:

order	interpolation	mean	std	sum
0	Nearest int	0.174	194	35337
1	Linear	0.067	156	13728
2	Quadratic	0.034	113	6821
3	Spline order 3	0.032	111	6430
4	order 4	0.031	108	6238
5	order 5	0.030	107	6183

So *order=2* or *order=3* seems a reasonable choice.

If you zoom the third figure, you will see that the red contours closely follow the green contours that were drawn first. This is also a measure of the precision in the re-projection because the intensities in the two images are the same.

Example: mu_overlaym101.py - Re-projection test with overlay data

```python
from kapteyn import maputils
from matplotlib import pyplot as plt
import numpy


#order = input("Enter order of spline interpolation in range 0..5: ")
order = 1
if order < 0: order = 0
if order > 5: order = 5

file1 = "m101OPT.fits"
Basefits = maputils.FITSimage(file1)
pxlim = (50,500); pylim = (50,500)
Basefits.set_limits(pxlim, pylim)

file2 = "m101HI.fits"
Overfits = maputils.FITSimage(file2)
Overfits.set_imageaxes(1,2)
# Not necessary to set limits if an overlay is required

fig = plt.figure(figsize=(6,7))
frame1 = fig.add_subplot(2,2,1)
frame2 = fig.add_subplot(2,2,2)
frame3 = fig.add_subplot(2,2,3)
frame4 = fig.add_subplot(2,2,4)
fs = 10      # Font size for titles

levels = [8000, 12000]

# Plot 1: Base
baseim = Basefits.Annotatedimage(frame1)
baseim.Image()
```

```
frame1.set_title("WCS1", fontsize=fs)

# Plot 2: Data with different wcs
overlayim = Overfits.Annotatedimage(frame2)
overlayim.Image()
overlayim.set_blankcolor('y')
frame2.set_title("WCS2", fontsize=fs)

# Plot 3: Base with contours reprojected from other source
baseim2 = Basefits.Annotatedimage(frame3)
baseim2.Image()
baseim2.Contours(levels=levels, colors='g')

# Filter the NaN's. Replace by 0.0 to be able tu use spline order > 1
#Overfits.boxdat[numpy.where(numpy.isnan(Overfits.boxdat))] = 0.0

# Set parameters for the interpolation routine
pars = dict(cval=numpy.nan, order=order)
Reprojfits = Overfits.reproject_to(Basefits, interpol_dict=pars)
overlayim2 = Basefits.Annotatedimage(frame3, boxdat=Reprojfits.boxdat)
overlayim2.Contours(levels=levels, colors='r')

frame3.set_title("Image WCS1 + \ncontours reprojected WCS2", fontsize=fs)
# Plot 4: Plot the difference between base and reprojection
x = Basefits.boxdat - overlayim2.data
print "Residual min, max, mean, std, sum:", x.flatten().min(), x.flatten().max(),\
      x.flatten().mean(), x.flatten().std(), x.flatten().sum()
diff = Basefits.Annotatedimage(frame4, boxdat=x)
diff.Image()
diff.set_histogrameq()
frame4.set_title(r"$\Delta$ = WCS1 - reprojected WCS2", fontsize=fs)

# User interaction
diff.interact_toolbarinfo()
diff.interact_imagecolors()
overlayim.interact_imagecolors()

maputils.showall()
```

2.2.17 Plotting markers from file

There are many situations where you want to identify features using markers at positions listed in a file. These positions are world coordinates. and assumed to be in degrees. The format of the file we used to read positions is as follows:

```
segment 1  rank 4  points 169
      31.270000 32.268889
      31.148611 32.277500
      31.104722 32.171389
      31.061111 32.114444
      31.120833 32.056667
```

The first line is an example of a comment. Therefore we use in `maputils.Annotatedimage.positionsfromfile()` character 's' as indentifier of a line with comments. In this method, the numbers are read from file with a method from module `tabarray` and are transformed to pixel coordinates in the projection system of the image in the FITS file. We changed the header values of CDELT a bit to get a bigger area in world coordinates. The positions are plotted as small dots. The dots represent coastlines in the Caribbean.

Example: mu_markersfromfile.py - Use special method to read positions from file and mark those positions

```python
from kapteyn import maputils
from matplotlib import pyplot as plt
from kapteyn import tabarray
import numpy

# Get a header and change some values
f = maputils.FITSimage("m101.fits")
header = f.hdr
header['CDELT1'] = 0.1
header['CDELT2'] = 0.1
header['CRVAL1'] = 285
header['CRVAL2'] = 20

# Use the changed header as external source for new object
f = maputils.FITSimage(externalheader=header)
fig = plt.figure()
frame = fig.add_subplot(1,1,1)
annim = f.Annotatedimage(frame)
grat = annim.Graticule()

fn = 'WDB/smallworld.txt'
# Note that in this file the latitudes are in the first column
# (column 0). And the longitudes in the second (column=1)
xp, yp = annim.positionsfromfile(fn, 's', cols=[1,0])
annim.Marker(x=xp, y=yp, mode='pixels', marker=',', color='b')
annim.plot()
frame.set_title("Markers in the Carribbean")

plt.show()
```

Method `maputils.Annotatedimage.positionsfromfile()` is based on method `tabarray.readColumns()`. They share the same parameters which implies that you have many options to get your data from a file.

The next plot also uses `tabarray.tabarray` to read coast line data. But here we wanted the coast line dots to be connected to get more realistic coast lines. For this we use the comment lines in the file as segment separator. This gives us an option to process the data in segments using tabarray's segment attribute and avoid that distant segments are connected with straight lines. Again we used the adapted header of the M101 FITS file to scale things up and to set the eye of the 'hurricane' in the Caribbean. The example also shows the use of masked arrays for plotting.

Example: mu_hurricane - Hurricane 'M101' threatens the Caribbean

```python
from kapteyn import maputils
from matplotlib import pyplot as plt
from kapteyn import tabarray
import numpy

def plotcoast(fn, pxlim, pylim, col='k'):

   coasts = tabarray.tabarray(fn, comchar='s')  # Read two columns from file
   for segment in coasts.segments:
      coastseg = coasts[segment].T
      xw = coastseg[1]; yw = coastseg[0]        # First one appears to be Latitude
      xs = xw; ys = yw                          # Reset lists which store valid pos.
      if 1:
        # Mask arrays if outside plot box
        xp, yp = annim.projection.topixel((numpy.array(xs),numpy.array(ys)))
```

```
         xp = numpy.ma.masked_where(numpy.isnan(xp) |
                         (xp > pxlim[1]) | (xp < pxlim[0]), xp)
         yp = numpy.ma.masked_where(numpy.isnan(yp) |
                         (yp > pylim[1]) | (yp < pylim[0]), yp)
         # Mask array could be of type numpy.bool_ instead of numpy.ndarray
         if numpy.isscalar(xp.mask):
            xp.mask = numpy.array(xp.mask, 'bool')
         if numpy.isscalar(yp.mask):
            yp.mask = numpy.array(yp.mask, 'bool')
         # Count the number of positions in these list that are inside the box
         j = 0
         for i in range(len(xp)):
            if not xp.mask[i] and not yp.mask[i]:
               j += 1
         if j > 200:   # Threshold to prevent too much detail and big pdf's
            frame.plot(xp.data, yp.data, color=col)


# Get a header and change some values
f = maputils.FITSimage("m101.fits")
header = f.hdr
header['CDELT1'] = 0.1
header['CDELT2'] = 0.1
header['CRVAL1'] = 285
header['CRVAL2'] = 20

# Use the changed header as external source for new object
f = maputils.FITSimage(externalheader=header, externaldata=f.dat)
fig = plt.figure()
frame = fig.add_subplot(1,1,1)
annim = f.Annotatedimage(frame, cmap="YlGn")
annim.Image()
grat = annim.Graticule()
grat.setp_ticklabel(wcsaxis=0, fmt="%g^{\circ}")
grat.setp_ticklabel(wcsaxis=1, fmt='Dms')
grat.setp_axislabel(plotaxis='bottom', label='West - East')
grat.setp_axislabel(plotaxis='left', label='South - North')
annim.plot()
annim.projection.allow_invalid = True

# Plot coastlines in black, borders in red
plotcoast('WDB/namer-cil.txt', annim.pxlim, annim.pylim, col='k')
plotcoast('WDB/namer-bdy.txt', annim.pxlim, annim.pylim, col='r')
plotcoast('WDB/samer-cil.txt', annim.pxlim, annim.pylim, col='k')
plotcoast('WDB/samer-bdy.txt', annim.pxlim, annim.pylim, col='r')

annim.interact_imagecolors()
plt.show()
```

2.2.18 Mosaics of plots

We have two examples of a mosaic of plots. First a mosaic is presented with an image and two position-velocity diagrams. The second is a classic examples which shows channel maps from an HI data cube at different velocities.

The base of the image is a velocity for which we want to show data and a pixel coordinate to set the position of the slice (*slicepos=51*). This code can be used as a template for a more general application, e.g. with user input of parameters that set velocity and slice position.

Example: mu_channelmaps1.py - Adding two slices

```
from kapteyn import maputils
from matplotlib import pylab as plt

# This is our function to convert velocity from m/s to km/s
def fx(x):
  return x/1000.0

# Create an object from the FITSimage class:
fitsobj = maputils.FITSimage('ngc6946.fits')

# We want to plot the image that corresponds to a certain velocity,
# let's say a radio velocity of 100 km/s
# Find the axis number that corresponds to the spectral axis:
specaxnum = fitsobj.proj.specaxnum
spec = fitsobj.proj.sub(specaxnum).spectra("VRAD-???")
channel = spec.topixel1d(100*1000.0)
channel = round(channel)                         # We need an integer pixel coordinate
vel = spec.toworld1d(channel)                    # Velocity near 100 km/s

# Set for this 3d data set the image axes and the position
# for the slice, i.e. the frequency pixel
lonaxnum = fitsobj.proj.lonaxnum
lataxnum = fitsobj.proj.lataxnum
fitsobj.set_imageaxes(lonaxnum,lataxnum, slicepos=channel)

fig = plt.figure(figsize=(7,8))
frame = fig.add_axes([0.3,0.5,0.4,0.4])
annim = fitsobj.Annotatedimage(frame)
annim.Image()

# The FITSimage object contains all the relevant information
# to set the graticule for this image
grat = annim.Graticule()
ruler = annim.Ruler(x1=-51.1916, y1=59.9283, x2=-51.4877, y2=60.2821,
                    units='arcmin', step=3, mscale=5.0,
                    color='w', world=True, ha='right')

grat.setp_tick(plotaxis="right", color='r')
pixellabels = annim.Pixellabels(plotaxis=("right","top"), color='r', fontsize=7)

# First position-velocity plot at RA=51
fitsobj.set_imageaxes(lataxnum, specaxnum, slicepos=51)
frame2 = fig.add_axes([0.1,0.3,0.8,0.1])
annim2 = fitsobj.Annotatedimage(frame2)
annim2.set_aspectratio(0.15)
annim2.Image()
grat2 = annim2.Graticule()
grat2.setp_axislabel(plotaxis="right", label='Velocity (km/s)',
                    fontsize=9, visible=True)
grat2.set_tickmode(plotaxis="right", mode="native_ticks")
grat2.setp_ticklabel(plotaxis="right", fmt="%+5g", fun=fx)
grat2.setp_axislabel("bottom",
                     label=r"Offset in latitude (arcmin) at $\alpha$ = pixel 51",
                     fontsize=9)
grat2.setp_axislabel(plotaxis="left", visible=False)
grat2.set_tickmode(plotaxis="left", mode="no_ticks")
annim2.Pixellabels(plotaxis=("top", "left"))
```

```python

# Second position-velocity plot at DEC=51
fitsobj.set_imageaxes(lonaxnum, specaxnum, slicepos=51)
frame3 = fig.add_axes([0.1,0.1,0.8,0.1])
annim3 = fitsobj.Annotatedimage(frame3)
annim3.set_aspectratio(0.15)
annim3.Image()
grat3 = annim3.Graticule()
grat3.setp_axislabel("right",
                     label='Velocity (km/s)', fontsize=9, visible=True)
grat3.set_tickmode(plotaxis='right', mode="native_ticks")
# The next line forces labels to be right aligned, but one needs a shift
# in x to set the labels outside the plot
grat3.setp_ticklabel(plotaxis="right", fmt="%8g", fun=fx, ha="right", x=1.075)
grat3.setp_axislabel(plotaxis="left", visible=False)
grat3.set_tickmode(plotaxis="left",  mode="no_ticks")
grat3.setp_axislabel("bottom",
                     label=r"Offset in longitude (arcmin) at $\delta$ = pixel 51",
                     fontsize=9)
annim3.Pixellabels(plotaxis=("top", "left"))

# Set title and adjust position of title
frame.set_title('NGC 6946 at %g km/s (channel %d)' % (vel/1000.0, channel), y=1.1)

maputils.showall()
```

For a series of so called *channel maps* we use Matplotlib's *add_subplot()* to position the plots automatically. To set the same scale for the colors in each plot, we first calculate the minimum and maximum values in the data with `maputils.FITSimage.get_dataminmax()`. The scale itself is set with parameters *clipmin* and *clipmax* in the constructor of `maputils.Annotatedimage`.

Before you make a hardcopy, it is possible to fine tune the colors because for each plot both mouse and key interaction is added with `maputils.Annotatedimage.interact_imagecolors()`. Some channels seem not to contain any signal but when you fine tune the colors you discover that they show features. For inspection one can set histogram equalization on/off for each plot separately. Special attention is paid to put labels in the plots with velocity information.

Also this example turns out to be a small but practical tool to inspect data.

Example: mu_channelmosaic.py - A mosaic of channelmaps

```python
from kapteyn import maputils
from matplotlib import pylab as plt

# This is our function to convert velocity from m/s to km/s
def fx(x):
  return x/1000.0

# Create an object from the FITSimage class:
fitsobj = maputils.FITSimage('ngc6946.fits')
specaxnum = fitsobj.proj.specaxnum
lonaxnum = fitsobj.proj.lonaxnum
lataxnum = fitsobj.proj.lataxnum
spec = fitsobj.proj.sub(specaxnum).spectra("VRAD-???")

start = 5; end = fitsobj.proj.naxis[specaxnum-1]; step = 4
channels = range(start,end,step)
nchannels = len(channels)

```

```
fig = plt.figure(figsize=(7,8))
fig.subplots_adjust(left=0, bottom=0, right=1, top=1, wspace=0, hspace=0)

vmin, vmax = fitsobj.get_dataminmax()
print "Vmin, Vmax of data in cube:", vmin, vmax
cmap = 'spectral'              # Colormap
cols = 5
rows = nchannels / cols
if rows*cols < nchannels: rows += 1
for i, ch in enumerate(channels):
   fitsobj.set_imageaxes(lonaxnum, lataxnum, slicepos=ch)
   print "Min, max in this channel: ", fitsobj.get_dataminmax(box=True)
   frame = fig.add_subplot(rows, cols, i+1)
   mplim = fitsobj.Annotatedimage(frame,
                                   clipmin=vmin, clipmax=vmax,
                                   cmap=cmap)
   mplim.Image()

   vel = spec.toworld1d(ch)
   velinfo = "ch%d = %.1f km/s" % (ch, vel/1000.0)
   frame.text(0.98, 0.98, velinfo,
              horizontalalignment='right',
              verticalalignment='top',
              transform = frame.transAxes,
              fontsize=8, color='w',
              bbox=dict(facecolor='red', alpha=0.5))
   mplim.plot()
   if i == 0:
      cmap = mplim.cmap
   mplim.interact_imagecolors()

plt.show()
```

2.2.19 Interaction with the display

Matplotlib (v 0.99) provides a number of built-in keyboard shortcuts. These are available on any Matplotlib window. Most of these shortcuts start actions that can also be started with buttons on the canvas. Also some keys interfere with the system and others don't seem to work for certain combinations of Matplotlib version and backend. Therefore a filter is applied to those shortcuts and now you need to specify the keys from the table below for which you want to use the shortcut with a function:

```
>>> from kapteyn.mplutil import KeyPressFilter
>>> KeyPressFilter.allowed = ['f', 'g', 'l']
```

Note that the interactions defined in module `maputils` could interfere with some of these keys. By default, the keys 'f' and 'g' are allowed.

Some Matplotlib Navigation Keyboard Shortcuts

Command	Keyboard Shortcut(s)
Toggle fullscreen	f
Toggle grid	g
Toggle y axis scale (log/linear)	l

Three methods from `maputils.Annotatedimage` add mouse and keyboard interaction. These methods are described in the next sections:

Changing colors in an image

Method `maputils.Annotatedimage.interact_imagecolors()` adds keys and mouse interaction for color editing i.e. change color scheme and settings for image and colorbar. The active keys are:

Command	Keyboard Shortcut(s)
Colormap scaling linear	1
Colormap scaling logarithmic	2
Colormap scaling exponential	3
Colormap scaling square root	4
Colormap scaling square	5
Toggle between data and histogram equalized version	h
Loop forward through list with colormaps	page-up
Loop backwards through list with colormaps	page-down
Save current colormap to disk	m
Toggle between inverse and normal scaling	9 (nine)
reset the colors to start values	0 (zero)
Change color of bad pixels (blanks)	b

The **right mouse button** must be pressed to change slope and offset of the function that maps image values to colors in the current color map.

Example: mu_smooth.py - Apply Gaussian filter

Smoothing of images is a technique that is often used to enhance the contrast of extended emission. Maputils provides a method for smoothing using a gaussian kernel. The method expects values for the dispersion of the Gauss in both directions x and y. To show how this can be used interactively, we give a small script where a Matplotlib Slider object changes the value of sigma (which is copied for both the x and y direction).

```python
from kapteyn import maputils
from matplotlib import pyplot as plt
from matplotlib.widgets import Slider

def func(x):
   nx = ny = x
   annim.set_blur(True, nx, ny, new=True)

f = maputils.FITSimage("m101.fits")
fig = plt.figure(figsize=(9,7))
frame = fig.add_subplot(1,1,1)
fr2 = fig.add_axes([0.3,0.01,0.4,0.03])
valinit = 1.0
sl = Slider(fr2, "Sigma: ", 0.1, 5.0, valinit=valinit)
sl.on_changed(func)

annim = f.Annotatedimage(frame)
#annim.set_colormap("mousse.lut")
annim.Image()
annim.plot()
func(valinit)
annim.interact_toolbarinfo()
annim.interact_imagecolors()
annim.interact_writepos()

plt.show()
```

Adding messages with position information

Method `maputils.Annotatedimage.interact_toolbarinfo()` connects movements of your mouse to messages in the toolbar of your canvas. The message shows pixel position, the corresponding world coordinates, and the image value of the pixel.

Note: There is a minimum width for the window to be able to display the message. If you see any imcomplete text, then resize the window until it is wide enough to show the message.

A programmer can change the formatting of the informative string using parameters with the same name as the attributes of an object from class `maputils.Annotatedimage.Positionmessage` If a format is set to *None*, its corresponding number(s) will not appear in the informative message. Here is an example how to skip the world coordinates (*wcsfmt*=*None*) and to add a format for the image values (*zfmt*).

```
>>> interact_toolbarinfo(wcsfmt=None, zfmt="%g")
```

Writing position information to the terminal

Method `maputils.Annotatedimage.interact_writepos()` writes the toolbar message with information about coordinates and image values to the terminal. This is a primitive way to collect positional information about features in a map.

```
>>> x,y= 196.4, 303.5  wcs=210.858423, 54.365281  Z=+8.74e+03
>>> x,y= 260.5, 378.3  wcs=210.806913, 54.400918  Z=+4.75e+03
>>> x,y= 391.1, 231.1  wcs=210.700135, 54.331856  Z=+6.08e+03
```

The first two numbers are the x and y pixel coordinate. Then two numbers follow which represent the pixel position in world coordinates. The units are the S.I. versions of the units found in the header. For spatial maps these units are degrees. The values are the real longitude and latitude even when the labels along the axes represent offsets. For spectral axes, the units depend on the selected spectral translation.

Here is a minimalistic example how to add user interaction:

```
"""Show interaction options"""
from kapteyn import maputils
from matplotlib import pyplot as plt
from kapteyn.mplutil import KeyPressFilter

KeyPressFilter.allowed = ['f','g', 'l']


f = maputils.FITSimage("m101.fits")
#f.set_limits((100,500),(200,400))

fig = plt.figure(figsize=(9, 7))
frame = fig.add_subplot(1, 1, 1)

mycmlist = ["mousse.lut", "ronekers.lut"]
maputils.cmlist.add(mycmlist)
print "Colormaps: ", maputils.cmlist.colormaps

mplim = f.Annotatedimage(frame, cmap="mousse.lut")
mplim.cmap.set_bad('w')
ima = mplim.Image()
mplim.Pixellabels()
mplim.Colorbar()
mplim.plot()

mplim.interact_toolbarinfo()
mplim.interact_imagecolors()
mplim.interact_writepos()

plt.show()
```

For a formatted output one could add parameters to *interact_writepos()*. The next line writes no pixel coordinates, writes spatial coordinates in degrees (not in HMS/DMS format) and adds a format for the world coordinates and the image value(s).

```
>>> interact_writepos(pixfmt=None, wcsfmt="%.12f", zfmt="%.3e", hmsdms=False)
```

Or if you need a lot of precision in the seconds of a HMS/DMS format:

```
>>> interact_writepos(dmsprec=3)
```

Interactive plotting of shapes for flux etc.

Another powerful tool of the `maputils` module is (together with module `shapes`) the option to propagate geometric shapes plotted in one image to other images from a list with images. A user selects a shape from the list Polygon, Ellipse, Circle, Rectangle and Spline, and interactively changes the geometry of a shape in an arbitray image in a mosaic of images. Then this shape duplicates itself in pixel coordinates or world coordinates on the other images. If we use the default setting then the duplication is in world coordinates, meaning that for different projections the geometry is different, but both represent the same area in the sky.

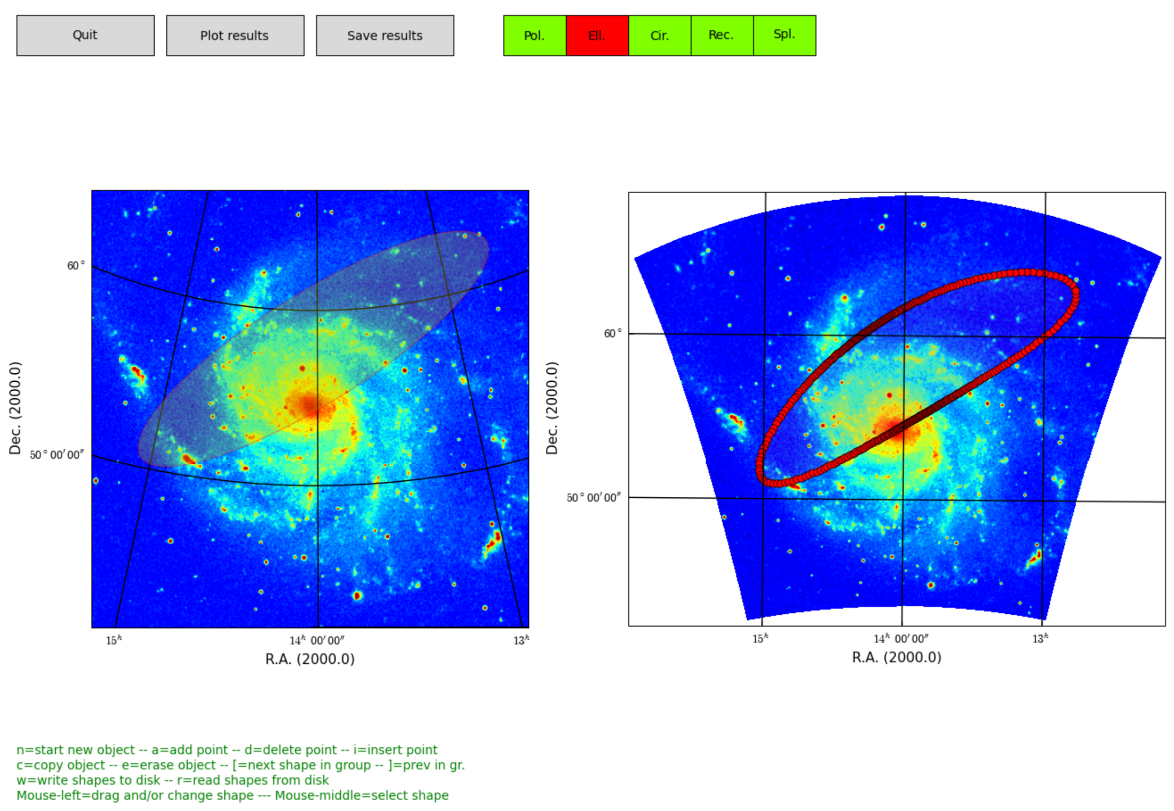
Note: If we change the image for interaction with circles or ellipses then their shape will change to the best approach of a circle or an ellipse for the new image and the deviation in geometry appears in the other image(s). This does not apply to the situation where we duplicate shapes in pixel coordinates.

For these areas the flux can be calculated. By default the flux is given by the (lambda) expression *s/a* where *s* represents the sum of the intensities of all the pixels enclosed by area *a*. One can supply a user defined function or lambda expression using Annotatedimage attribute *fluxfie*. Sometimes more precision is required. Then one can subdivide pixels using Annotatedimage attribute *pixelstep*.

```
from kapteyn import maputils, shapes
from matplotlib import pyplot as plt

Basefits = maputils.FITSimage("m101big.fits")
hdr = Basefits.hdr.copy()

hdr['CTYPE1'] = 'RA---MER'
hdr['CTYPE2'] = 'DEC--MER'
hdr['CRVAL1'] = 0.0
hdr['CRVAL2'] = 0.0
naxis1 = Basefits.hdr['NAXIS1']
naxis2 = Basefits.hdr['NAXIS2']

# Get an estimate of the new corners
x = [0]*5; y = [0]*5
x[0], y[0] = Basefits.proj.toworld((1,1))
x[1], y[1] = Basefits.proj.toworld((naxis1,1))
x[2], y[2] = Basefits.proj.toworld((naxis1,naxis2))
```

```
x[3], y[3] = Basefits.proj.toworld((1,naxis2))
x[4], y[4] = Basefits.proj.toworld((naxis1/2.0,naxis2))

# Create a dummy object to calculate pixel coordinates
# in the new system. Then we can find the area in pixels
# that corresponds to the area in the sky.
f = maputils.FITSimage(externalheader=hdr)
px, py = f.proj.topixel((x,y))
pxlim = [int(min(px))-10, int(max(px))+10]
pylim = [int(min(py))-10, int(max(py))+10]

Reprojfits = Basefits.reproject_to(hdr, pxlim_dst=pxlim, pylim_dst=pylim)

fig = plt.figure(figsize=(14,9))
frame1 = fig.add_axes([0.07,0.1,0.35, 0.8])
frame2 = fig.add_axes([0.5,0.1,0.43, 0.8])
im1 = Basefits.Annotatedimage(frame1)
im1.set_blankcolor('k')
im2 = Reprojfits.Annotatedimage(frame2)
im1.Image(); im1.Graticule()
im2.Image(); im2.Graticule()
im1.plot(); im2.plot()
im1.interact_imagecolors(); im1.interact_toolbarinfo()
im2.interact_imagecolors(); im2.interact_toolbarinfo()
#im1.fluxfie = lambda s, a: s/a
#im2.fluxfie = lambda s, a: s/a
im1.pixelstep = 0.2
im2.pixelstep = 0.5
images = [im1, im2]
shapes = shapes.Shapecollection(images, fig, wcs=True, inputwcs=True)

plt.show()
```

Your Matplotlib figure with one or more images gets a number of buttons at the top of your figure. You should anticipate on this when setting the figure size. Ofcourse one can also resize the plot window to make space for the buttons. The gui is simple. Here is an example. It corresponds to the example script above. A re-projection (to Mercator) of M101 (with exaggerated values for the pixel sizes) is displayed together with the original image. One ellipse is plotted to demonstrate that the same area in the re-projection looks different. If enough resolution (pixelstep=0.2) is used, then the flux in both shapes is comparable.



n=start new object -- a=add point -- d=delete point -- i=insert point
c=copy object -- e=erase object -- [=next shape in group --]=prev in gr.
w=write shapes to disk -- r=read shapes from disk
Mouse-left=drag and/or change shape --- Mouse-middle=select shape

Fig. – Calculate flux in user defined shapes in images with different world coordinate systems.

Adding and using external color maps

In the constructor of `maputils.Annotatedimage` one can set a color map with keyword *cmap*. There are four options here:

Option	Example
Matplotlib color map (string)	cmap='jet'
Path and filename of color map on disk	cmap='/home/user/myluts/rainbow4.lut'
A Color map from the Kapteyn Package	cmap='ronekers.lut'
Instance of class `mplutil.VariableColormap`	cmap=myimage.cmap

Module `maputils` has a global list called *cmlist* which contains the colormaps provided by Matplotlib. You can add an external colormap to this list as follows:

```
>>> maputils.cmlist.add('/home/user/luts/rainbow4.lut')
```

It will be prepended to the existing list. One can also prepend multiple external colormaps assembled in a list. This list can also be compiled from the color maps available in the Kapteyn Package. If you have a number of local color maps then use Python's glob function to read them all (or a selection) as in the next example:

```
from kapteyn import maputils
from matplotlib import pyplot as plt
from kapteyn import mplutil
import glob

f = maputils.FITSimage("m101.fits")

fig = plt.figure(figsize=(9,7))
```

```
frame = fig.add_axes([0.1,0.2,0.85, 0.75])

extralist = mplutil.VariableColormap.luts()
print "Extra luts from Kapteyn Package", extralist
maputils.cmlist.add(extralist)

mycmlist = glob.glob("*.lut")
print "\nFound private color maps:", mycmlist
maputils.cmlist.add(mycmlist)

print "\nAll color maps now available: ", maputils.cmlist.colormaps

annim = f.Annotatedimage(frame) #,cmap="mousse.lut")
annim.set_colormap("mousse.lut")
annim.Image()
annim.Pixellabels()
annim.Colorbar()
annim.plot()

annim.interact_toolbarinfo()
annim.interact_imagecolors()
annim.interact_writepos()

units = 'unknown'
if 'BUNIT' in f.hdr:
   units = f.hdr['BUNIT']
helptext  = "File: [%s]  Data units:  [%s]\n" % (f.filename, units)
helptext += annim.get_colornavigation_info()
tdict = dict(color='g', fontsize=10, va='bottom', ha='left')
fig.text(0.01,0.01, helptext, tdict)

plt.show()
```

The format of a colormap on disk (also called a color LookUp Table or *lut*) is simple. There should be a number (e.g. 256) lines with three floating point numbers between 0 and 1 which represent Red, Green and Blue.

More color resolution

```
from kapteyn import maputils
from matplotlib import pyplot as plt
from numpy import mgrid,exp

f = maputils.FITSimage("m101.fits")
n1, n2 = f.proj.naxis
X,Y = mgrid[0:n1, 0:n2]
Z = exp( -(X**2)/1.0e5-(Y**2)/1.0e5 )
f2 = maputils.FITSimage(externalheader=f.hdr, externaldata=Z)

fig = plt.figure()
frame = fig.add_subplot(1,2,1)
annim = f2.Annotatedimage(frame)
annim.set_colormap("rainbow.lut")
annim.cmap.set_length(64)
annim.Image()
annim.Pixellabels()
annim.Colorbar(fontsize=7, orientation='horizontal')
annim.plot()
annim.interact_toolbarinfo()
annim.interact_imagecolors()
annim.interact_writepos()

frame2 = fig.add_subplot(1,2,2)
annim2 = f2.Annotatedimage(frame2)
annim2.set_colormap("rainbow.lut")
annim2.cmap.set_length(1021)
annim2.Image()
annim2.Pixellabels()
annim2.Colorbar(fontsize=7, orientation='horizontal')
annim2.plot()
annim2.interact_toolbarinfo()
annim2.interact_imagecolors()
annim2.interact_writepos()

plt.show()
```

In this plot we demonstrate the difference between a small color map (64 color entries) and a big color map (1021 entries). The plot on the left uses the small color map. What you should observe are the false contours because that color map does not have the enough resolution to show smooth transitions between colors. The plot on the right has a big color map and there you don't see these transitions.

Note: The default length of a color map is 256. With this length the effect of steps in the color gradient is less obvious but still there. You should only extend your color map if a high resolution is required.

Reuse of modified colormap

If you modify the default colors in an image and want to be able to restore the exact color scheme in a second run, then save the modified colormap to disk with key 'm'. Note that we assume you connected mouse and keyboard interaction with `maputils.Annotatedimage.interact_imagecolors()`. The file on disk will get the name of the FITS file that you opened (or an alternative if you used external header and image data) followed by the extension *.lut*. This colormap can be used as external colormap to restore the adjusted colors.

Sharing the same colormap

There are at least two circumstances where one wants to share a colormap between multiple images. The first is the movie loop of images and the second is when we have a mosaic of channel maps. A change in the colormap settings affects all images in the movie or in the mosaic. There could be exceptions where you want each image to have its own colormap, but usually it is more convenient to share it. The trick is to use a loop over all images and to set the colormap for the first image and copy this colormap for the others. Have a look at the examples that illustrate movies and mosaics:

```
cmap = 'spectral'
for i, ch in enumerate(channels):
   fitsobj.set_imageaxes(lonaxnum, lataxnum, slicepos=ch)
   frame = fig.add_subplot(rows, cols, i+1)
   mplim = fitsobj.Annotatedimage(frame, clipmin=vmin, clipmax=vmax,
                                  cmap=cmap)
   mplim.Image()
   mplim.plot()
   if i == 0:
    cmap = mplim.cmap      # Copy map to set for other images
  mplim.interact_imagecolors()
```

Blanks

An image can contain some ‘bad pixels’. A bad pixel is a location where a physical value is missing. These pixels are represented by the value *NaN*. For FITS files where the data are integers (i.e. keyword BITPIX has a positive value) one needs to set an integer value for a bad pixel with FITS keyword *BLANK*. For the extraction of data the package *PyFITS* is used. PyFITS should take care of blanks automatically.

Some FITS writers use for BITPIX=-32 a blank value equal to -inf. To avoid problems with plotting images and contours we replace these values in the data with NaN’s first before anything is plotted.

In the next example we inserted some blank values. They appear as a square in the middle of the image. The color of a blank pixel is either set in the constructor of `maputils.Annotatedimage` with keyword *blankcolor*, or it can be changed with key ‘b’ if we applied user interaction with `maputils.Annotatedimage.interact_imagecolors()`.

```
from kapteyn import maputils
from matplotlib import pyplot as plt
from matplotlib.colors import LogNorm
import glob

f = maputils.FITSimage("blanksetmin32.fits")
#f = maputils.FITSimage("blankset16.fits")
f.set_imageaxes(1,2)

fig = plt.figure(figsize=(9,7))
frame = fig.add_subplot(1,1,1)

mycmlist = ["mousse.lut", "ronekers.lut"]
maputils.cmlist.add(mycmlist)
print "Colormaps: ", maputils.cmlist.colormaps

mplim = f.Annotatedimage(frame, cmap="mousse.lut", blankcolor='w')
mplim.Image()
#mplim.Image()
#mplim.set_blankcolor('c')
mplim.Pixellabels()
mplim.Colorbar()
mplim.plot()

mplim.interact_toolbarinfo()
mplim.interact_imagecolors()
mplim.interact_writepos()

plt.show()
```

If you change the input FITS file from *blanksetmin32.fits* to *blankset16.fits*, then you get the same image and the same blanks, which proves that the blanks can also be read from a FITS file with scaled data.

2.2.20 Movies

In version 2.3 of the Kapteyn Package, a lot of effort was spent on developing a class for access to- and visualization of N dimensional data. This visualization is restricted to a movieloop of images and side panels with slices from a data cube. One can change the color mapping interactively and graticules can be overlayed.

The base class for interactive image viewing is `maputils.Cubes`. It needs only a Figure instance for its initialization. There are a couple of extra parameters. One is *helptext* which shows a informative text on your plot to help you with the interaction. The other is *imageinfo* which must be set to True if you want a line in your plot with text showing which slice is on display by giving the name and world coordinate of the repeat axes.

The core of interactive image display is `maputils.Cubes`. An object of this class is constructed with:

```
>>>   myCubes = Cubes(fig, toolbarinfo=True, printload=False,
>>>                         helptext=False, imageinfo=True)
```

```
#!/usr/bin/env python
from kapteyn import wcsgrat, maputils
from matplotlib.pyplot import figure, show

fig = figure()
myCubes = maputils.Cubes(fig, toolbarinfo=True, printload=False,
                              helptext=False, imageinfo=True)

# Create a maputils FITS object from a FITS file on disk
fitsobject = maputils.FITSimage('ngc6946.fits')
naxis3 = fitsobject.hdr['NAXIS3']
# Note that slice positions follow FITS syntax, i.e. start at 1
slicepos = range(1,naxis3+1)

frame = fig.add_subplot(1,1,1)
vmin, vmax = fitsobject.get_dataminmax()
cube = myCubes.append(frame, fitsobject, (1,2), slicepos,
                      vmin=vmin, vmax=vmax, hasgraticule=True)

show()
```

2.2.21 Coordinate transformations

An object from class `maputils.Annotatedimage` has a so called wcs projection object (see module `wcs`) as attribute (called *projection*) This projection object has methods for the transformation between pixel- and world coordinates. In the context of module `maputils` there are two convenience methods with the same name and same functionality i.e. `maputils.Annotatedimage.toworld()` and `maputils.Annotatedimage.topixel()`. But in maputils we deal with two dimensional structures only, so these methods are easier to use. Look at the next example to find out how you can use them

```
from kapteyn import maputils

# The set is 3-dim. but default the first two axes are
# used to extract the image data
f = maputils.FITSimage("ngc6946.fits")

annim = f.Annotatedimage()
x = 200; y = 350
lon, lat  = annim.toworld(x,y)
print "lon, lat =", lon, lat

x, y = annim.topixel(lon, lat)
print "x, y = ", x, y
```

Module `maputils` supports the extraction of data slices in any direction. This enables you to inspect maps with for example one spatial axis and one spectral axis (so called position velocity map). For conversions between pixel- and world coordinates we need a matching spatial axis to process the transformation. The methods used in the previous example can be used to get values for the matching axis also. One only needs to set parameter *matchspatial* to True:

```
from kapteyn import maputils

f = maputils.FITSimage("ngc6946.fits")
# Get an XV slice at DEC=51
f.set_imageaxes(1,3, slicepos=51)
annim = f.Annotatedimage()

# Which pixel coordinates correspond to CRVAL's?
crpix = annim.projection.crpix
print "CRPIX from header", crpix

# Convert these to world coordinates
x = crpix[0]; y = crpix[1]
lon, velo, lat  = annim.toworld(x, y, matchspatial=True)
print "lon, velo, lat =", lon, velo, lat
print "Should be equivalent to CRVAL:", annim.projection.crval

x, y, slicepos = annim.topixel(lon, velo, matchspatial=True)
print "Back to pixel coordinates: x, y =", x, y, slicepos
```

Note that we can modify the FITS object 'f' in the example so that instead of velocities we get corresponding frequencies. Add a spectral translation with method `maputils.FITSimage.set_spectrans()` as in:

```
>>> f = maputils.FITSimage("ngc6946.fits")
>>> f.set_imageaxes(1,3, slicepos=51)
>>> f.set_spectrans("FREQ-???")
```

2.2.22 Glossary

graticule The network of lines of latitude and longitude upon which a map is drawn.

all-sky plot Plot of the sky in arbitrary projection showing a range in longitudes between [-180,180) degrees and a range in latitudes between [-90,90).

prompt function Function supplied by a user (or one of the pre defined functions in module `maputils` which prompts a user to enter relevant data. The functions need to return their data in a special format. See documentation in `maputils`.

CRVAL The reference world coordinate that corresponds to a reference pixel. Its value is extracted from a FITS header or a Python dictionary.

2.3 Least squares fitting with kmpfit

2.3.1 Introduction

We like code examples in our documentation, so let's start with an example:

```
#!/usr/bin/env python
# Short demo kmpfit (04-03-2012)

import numpy
from kapteyn import kmpfit
```

```
def residuals(p, data):  # Residuals function needed by kmpfit
   x, y = data           # Data arrays is a tuple given by programmer
   a, b = p              # Parameters which are adjusted by kmpfit
   return (y-(a+b*x))

d = numpy.array([42, 6.75, 25, 33.8, 9.36, 21.8, 5.58, 8.52, 15.1])
v = numpy.array([1294, 462, 2562, 2130, 750, 2228, 598, 224, 971])

paramsinitial = [0, 70.0]
fitobj = kmpfit.Fitter(residuals=residuals, data=(d,v))
fitobj.fit(params0=paramsinitial)

print "\nFit status kmpfit:"
print "===================="
print "Best-fit parameters:        ", fitobj.params
print "Asymptotic error:           ", fitobj.xerror
print "Error assuming red.chi^2=1: ", fitobj.stderr
print "Chi^2 min:                  ", fitobj.chi2_min
print "Reduced Chi^2:              ", fitobj.rchi2_min
print "Iterations:                 ", fitobj.niter
print "Number of free pars.:       ", fitobj.nfree
print "Degrees of freedom:         ", fitobj.dof
```

If you run the example, you should get output similar to:

```
Fit status kmpfit:
====================
Best-fit parameters:         [414.71769219487254, 44.586628080854609]
Asymptotic error:            [ 0.60915502  0.02732865]
Error assuming red.chi^2=1:  [ 413.07443146   18.53184367]
Chi^2 min:                   3218837.22783
Reduced Chi^2:               459833.889689
Iterations:                  2
Number of free pars.:        2
Degrees of freedom:          7
```

In this tutorial we try to show the flexibility of the least squares fit routine in `kmpfit` by showing examples and some background theory which enhance its use. The *kmpfit* module is an excellent tool to demonstrate features of the (non-linear) least squares fitting theory. The code examples are all in Python. They are not complex and almost self explanatory.

kmpfit is the Kapteyn Package Python binding for a piece of software that provides a robust and relatively fast way to perform non-linear least-squares curve and surface fitting. The original software called MPFIT was translated to IDL from Fortran routines found in MINPACK-1 and later converted to a C version by Craig Markwardt *[Mkw]*. The routine is stable and fast and has additional features, not found in other software, such as model parameters that can be fixed and boundary constraints that can be imposed on parameter values. We will show an example in section *Fitting Voigt profiles*, where this feature is very helpful to keep the profile width parameters from becoming negative.

kmpfit has many similar features in common with SciPy's Fortran-based `scipy.optimize.leastsq()` function, but *kmpfit*'s interface is more friendly and flexible and it is a bit faster. It provides also additional routines to calculate confidence intervals. And most important: you don't need Fortran to build it because it is based on code written in C. Mark Rivers created a Python version from Craig's IDL version (*mpfit.py*). We spent a lot of time in debugging this pure Python code (after converting its array type from Numarray to NumPy). It it not fast and we couldn't get the option of using derivatives to work properly. So we focused on the C version of *mpfit* and used Cython to build the C extension for Python.

A least squares fit method is an algorithm that minimizes a so-called *objective function* for N data points $(x_i, y_i), i =$

$0, ..., N-1$. These data points are measured and often y_i has a measurement error that is much smaller than the error in x_i. Then we call *x* the independent and *y* the dependent variable. In this tutorial we will also deal with examples where the errors in x_i and y_i are comparable.

Objective function

The method of least squares adjusts the parameters of a model function *f(parameters, independent_variable)* by finding a minimum of a so-called *objective function*. This objective function is a sum of values:

$$S = \sum_{i=0}^{N-1} r_i^2 \tag{2.8}$$

Objective functions are also called *merit* functions. Least squares routines also predict what the range of best-fit parameters will be if we repeat the experiment, which produces the data points, many times. But it can do that only for objective functions if they return the (weighted) sum of squared residuals (WSSR). If the least squares fitting procedure uses measurement errors as weights, then the objective function *S* can be written as a maximum-likelihood estimator (MLE) and *S* is then called chi-squared (χ^2).

If we define $\mathbf{p}$ as the set of parameters and take *x* for the independent data then we define a residual as the difference between the actual dependent variable y_i and the value given by the model:

$$r(\mathbf{p}, [x_i, y_i]) = y_i - f(\mathbf{p}, x_i) \tag{2.9}$$

A model function $f(\mathbf{p}, x_i)$ could be:

```python
def model(p, x):          # The model that should represent the data
   a, b = p               # p == (a,b)
   return a + b*x         # x is explanatory variable
```

A residual function $r(\mathbf{p}, [x_i, y_i])$ could be:

```python
def residuals(p, data):        # Function needed by fit routine
   x, y, err = data            # The values for x, y and weights
   a, b = p                    # The parameters for the model function
   return (y-model(p,x))/err   # An array with (weighted) residuals)
```

The arguments of the residuals function are *p* and *data*. You can give them any name you want. Only the order is important. The first parameter is a sequence of model parameters (e.g. slope and offset in a linear regression model). These parameters are changed by the fitter routine until the best-fit values are found. The number of model parameters is given by a sequence of initial estimates. We will explain this in more detail in the section about initial estimates.

The second parameter of the *residuals()* function contains the data. Usually this is a tuple with a number of arrays (e.g. x, y and weights), but one is not restricted to tuples to pass the data. It could also be an object with arrays as attributes. The parameter is set in the constructor of a *Fitter* object. We will show some examples when we discuss the *Fitter* object.

One is not restricted to one independent (*explanatory*) variable. For example, for a plane the dependent (*response*) variable y_i depends on two independent variables (x_{1_i}, x_{2_i})

```python
>>>    x1, x2, y, err = data
```

kmpfit needs only a specification of the residuals function (2.9). It defines the objective function *S* itself by squaring the residuals and summing them afterwards. So if you pass an array with weights w_i which are calculated from $1/\sigma_i^2$, then you need to take the square root of these numbers first as in:

```python
def residuals(p, data):        # Function needed by fit routine
   x, y, w = data              # The values for x, y and weights
   a, b = p                    # The parameters for the model function
```

```
    w = numpy.sqrt(w)           # kmpfit does the squaring
    return w*(y-model(p,x))     # An array with (weighted) residuals)
```

It is more efficient to store the square root of the weights beforehand so that it is not necessary to repeat this (often many times) in the residuals function itself. This is different if your weights depend on the model parameters, which are adjusted in the iterations to get a best-fit. An example is the residuals function for an orthogonal fit of a straight line:

```
def residuals(p, data):
    # Residuals function for data with errors in both coordinates
    a, theta = p
    x, y = data
    B = numpy.tan(theta)
    wi = 1/numpy.sqrt(1.0 + B*B)
    d = wi*(y-model(p,x))
    return d
```

Note: For *kmpfit*, you need only to specify a residuals function. The least squares fit method in *kmpfit* does the squaring and summing of the residuals.

Linearity

For many least squares fit problems we can use analytical methods to find the best-fit parameters. This is the category of linear problems. For linear least-squares problems (LLS) the second and higher derivatives of the fitting function with respect to the parameters are zero. If this is not true then the problem is a so-called non-linear least-squares problem (NLLS). We use *kmpfit* to find best-fit parameters for both problems and use the analytical methods of the first category to check the output of *kmpfit*. An example of a LLS problem is finding the best fit parameters of the model:

$$\begin{gathered} f(a,x) = a\ \sin(x) \\ \frac{\partial f}{\partial a} = \sin(x)\ \Rightarrow \frac{\partial^2 f}{\partial a^2} = 0 \end{gathered} \tag{2.10}$$

An example of a NLLS problem is finding the best fit parameters of the model:

$$\begin{gathered} f(a,x) = \sin(a\,x) \\ \frac{\partial f}{\partial a} = x\cos(a\,x)\ \Rightarrow \frac{\partial^2 f}{\partial a^2} \neq 0 \end{gathered} \tag{2.11}$$

A well-known example of a model that is non-linear in its parameters, is a function that describes a Gaussian profile as in:

```
def my_model(p, x):
    A, mu, sigma, zerolev = p
    return( A * numpy.exp(-(x-mu)*(x-mu)/(2.0*sigma*sigma)) + zerolev )
```

Note: In the linear case, parameter values can be determined analytically with straightforward linear algebra. *kmpfit* finds best-fit parameters for models that are either linear or non-linear in their parameters. If efficiency is an issue, one should find and apply an analytical method.

In the linear case, parameter values can be determined by comparatively simple linear algebra, in one direct step.

Goal

The function that we choose is based on a model which should describe the data so that *kmpfit* finds best-fit values for the free parameters in this model. These values can be used for interpolation or prediction of data based on the measurements and the best-fit parameters. *kmpfit* varies the values of the free parameters until it finds a set of values which minimize the objective function. Then, either it stops and returns a result because it found these *best-fit* parameters, or it stops because it met one of the stop criteria in *kmpfit* (see next section). Without these criteria, a fit procedure that is not converging would never stop.

Later we will discuss a familiar example for astronomy when we find best-fit parameters for a Gaussian to find the characteristics of a profile like the position of the maximum and the width of a peak.

Stop criteria

LLS and NLLS problems are solved by *kmpfit* by using an iterative procedure. The fit routine attempts to find the minimum by doing a search. Each iteration gives an improved set of parameters and the sum of the squared residuals is calculated again. *kmpfit* is based on the C version of *mpfit* which uses the Marquardt-Levenberg algorithm to select the parameter values for the next iteration. The Levenberg-Marquardt technique is a particular strategy for iteratively searching for the best fit. These iterations are repeated until a criterion is met. Criteria are set with parameters for the constructor of the *Fitter* object in *kmpfit* or with the appropriate attributes:

- `ftol` - a nonnegative input variable. Termination occurs when both the actual and predicted relative reductions in the sum of squares are at most `ftol`. Therefore, `ftol` measures the relative error desired in the sum of squares. The default is: 1e-10
- `xtol` - a nonnegative input variable. Termination occurs when the relative error between two consecutive iterates is at most `xtol`. therefore, `xtol` measures the relative error desired in the approximate solution. The default is: 1e-10
- `gtol` - a nonnegative input variable. Termination occurs when the cosine of the angle between *fvec* (is an internal input array which must contain the functions evaluated at x) and any column of the Jacobian is at most `gtol` in absolute value. Therefore, `gtol` measures the orthogonality desired between the function vector and the columns of the Jacobian. The default is: 1e-10
- `maxiter` - Maximum number of iterations. The default is: 200
- `maxfev` - Maximum number of function evaluations. The default is: 0 (no limit)

A `Fitter` object

After we defined a residuals function, we need to create a Fitter object. A Fitter object is an object of class **Fitter**. This object tells the fit procedure which data should be passed to the residuals function. So it needs the name of the residuals function and an object which provides the data. In most of our examples we will use a tuple with references to arrays. Assume we have a residuals function called *residuals* and two arrays *x* and *y* with data from a measurement, then a `Fitter` object is created by:

```
fitobj = kmpfit.Fitter(residuals=residuals, data=(x,y))
```

Note that *fitobj* is an arbitrary name. You need to store the result to be able to retrieve the results of the fit. The real fit is started when we call method `fit`. The fit procedure needs start values. Often the fit procedure is not sensitive to these values and you can enter 1 as a value for each parameter. But there are also examples where these *initial estimates* are important. Starting with values that are not close to the best-fit parameters could result in a solution that is a local minimum and not a global minimum.

If you imagine a surface which is a function of parameter values and heights given by the the sum of the residuals as function of these parameters and this surface shows more than one minimum, you must be sure that you start your fit nearby the global minimum.

Example: `kmpfit_chi2landscape_gauss.py` **- Chi-squared landscape for model that represents a Gaussian profile**

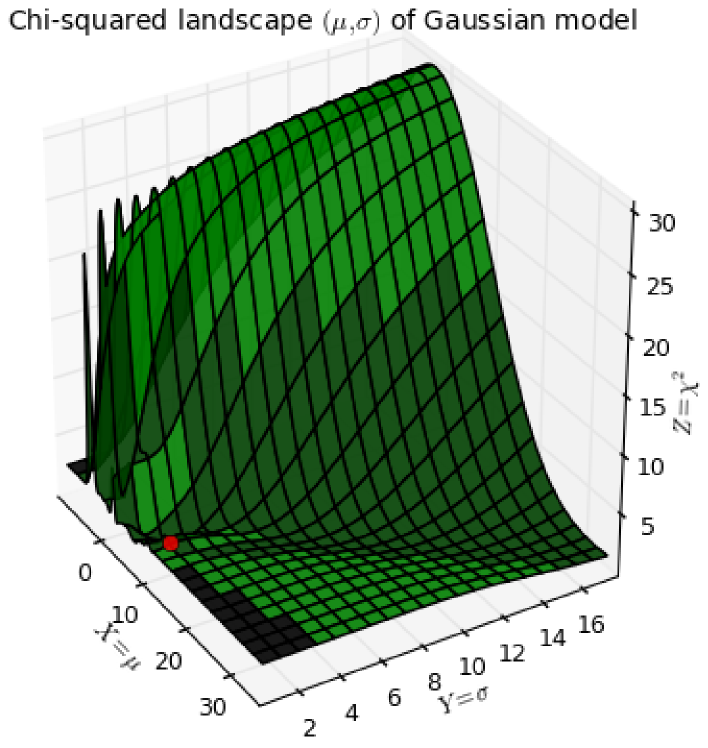


Fig. 2.1: Chi-squared parameter landscape for Gaussian model. The value of chi-squared is plotted along the z-axis.

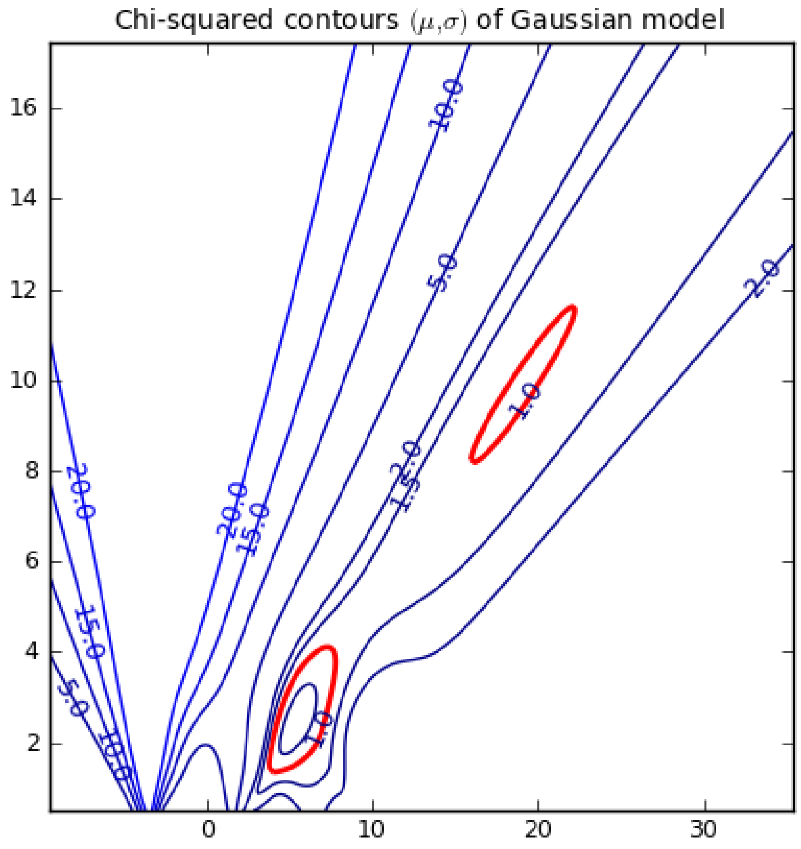
The figure shows the chi-squared parameter landscape for a model that represents a Gaussian. The landscape axes are model parameters: the position of the peak μ and σ which is a measure for the width of the peak (half width at 1/e of peak). The relation between σ and the the full width at half maximum (FWHM) is: $\mathrm{FWHM} = 2\sigma\sqrt{2ln2} \approx 2.35\,\sigma$. If you imagine this landscape as a solid surface and release a marble, then it rolls to the real minimum (red dot in the figure) only if you are not too far from this minimum. If you start for example in the front right corner, the marble will never end in the real minimum. Note that the parameter space is in fact 4 dimensional (4 free parameters) and therefore more complicated than this example. In the figure we scaled the value for chi-squared to avoid labels with big numbers.

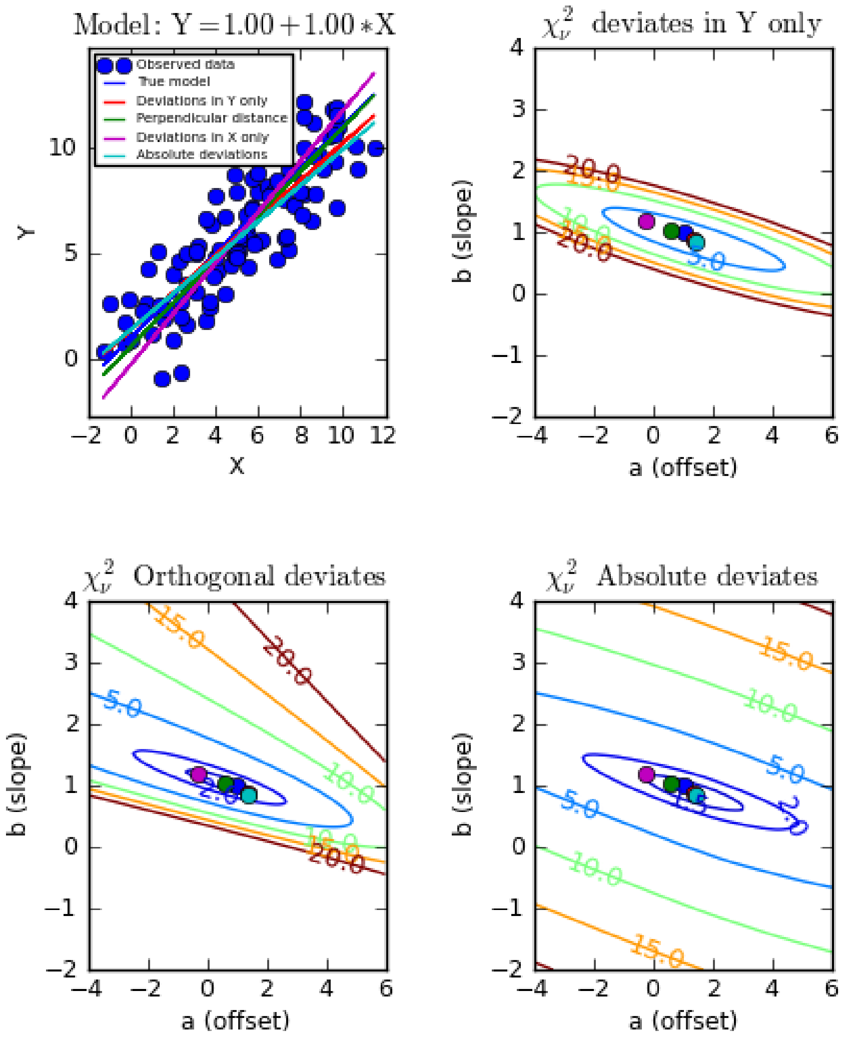
Another representation of the parameter space is a contour plot. It is created by the same example code:

These contour plots are very useful when you compare different objective functions. For instance if you want to compare an objective function for orthogonal fitting with an an objective function for robust fitting.

Example: `kmpfit_contours_objfunc.py` **- Comparing objective functions with contour plots**

A model which represents a straight line, always shows a very simple landscape with only one minimum. Wherever you release the marble, you will always end up in the real minimum. Then, the quality of the values of the initial estimates are not important to the quality of the fit result.


Chi-squared contours (μ,σ) of Gaussian model
16
14
12
10
8
6
4
2
0
10
20
30
20.0
15.0
10.0
5.0
2.0
1.5
1.0


Model: Y = 1.00 + 1.00 * X
Observed data
True model
Deviations in Y only
Perpendicular distance
Deviations in X only
Absolute deviations
X
Y
χ^2_ν deviates in Y only
χ^2_ν Orthogonal deviates
χ^2_ν Absolute deviates
a (offset)
b (slope)

The initial estimates are entered in parameter `params0`. You can enter this either in the constructor of the `Fitter` object or in the method `fit()`. In most examples we use the latter because then one can repeat the same fit with different initial estimates as in:

```
fitobj.fit(params0=[1,1])
```

The results are stored in the attributes of *fitobj*. For example the best-fit parameters are stored in `fitobj.params`. For a list of all attributes and their meaning, see the documentation of `kmpfit`.

An example of an overview of the results could be:

```
print "Fit status: ", fitobj.message
print "Best-fit parameters:      ", fitobj.params
print "Covariance errors:        ", fitobj.xerror
print "Standard errors           ", fitobj.stderr
print "Chi^2 min:                ", fitobj.chi2_min
print "Reduced Chi^2:            ", fitobj.rchi2_min
print "Iterations:               ", fitobj.niter
print "Number of function calls: ", fitobj.nfev
print "Number of free pars.:     ", fitobj.nfree
print "Degrees of freedom:       ", fitobj.dof
print "Number of pegged pars.:   ", fitobj.npegged
```

There is a section about the use and interpretation of parameter errors in *Standard errors of best-fit values*. In the next chapter we will put the previous information together and compile a complete example.

2.3.2 A Basic example

In this section we explain how to setup a residuals function for *kmpfit*. We use vectorized functions written with *NumPy*.

The residual function

Assume we have data for which we know that the relation between X and Y is a straight line with offset *a* and slope *b*, then a model $f(\mathbf{p},\mathbf{x})$ could be written in Python as:

```
def model(p, x):
   a,b = p
   y = a + b*x
   return y
```

Parameter `x` is a NumPy array and `p` is a NumPy array containing the model parameters *a* and *b*. This function calculates response Y values for a given set of parameters and an array with explanatory X values.

Then it is simple to define the residuals function $r(\mathbf{p},[x_i,y_i])$ which calculates the residuals between data points and model:

```
def residuals(p, data):
   x, y = data
   return y - model(p,x)
```

This residuals function has always two parameters. The first one `p` is an array with parameter values in the order as defined in your model, and `data` is an object that stores the data arrays that you need in your residuals function. The object could be anything but a list or tuple is often most practical to store the required data. We will explain a bit more about this object when we discuss the constructor of a *Fitter* object. We need not worry about the sign of the residuals because the fit routine calculates the the square of the residuals itself.

Of course we can combine both functions `model` and `residuals` in one function. This is a bit more efficient in Python, but usually it is handy to have the model function available if you need to plot the model using different sets of best-fit parameters.

The objective function which is often used to fit the best-fit parameters of a straight line model is for example:

$$\chi^2([a,b],x) = \sum_{i=0}^{N-1} \left(\frac{y_i - a - bx_i}{\sigma_i}\right)^2 \tag{2.12}$$

Assume that the values σ_i are given in array *err*, then this objective function translates to a residuals function:

```
def residuals(p, data):
   x, y, err = data
   ym = a + b*x              # Model data
   return (y-ym)/err         # Squaring is done in Fitter routine
```

Another example is an objective function for *robust* (i.e. less sensitive to outliers) for a straight line model without weights. For robust fitting one does not use the square of the residuals but the absolute value.

$$S = \sum |y_i - a - bx_i| \tag{2.13}$$

We cannot avoid that the Fitter routine squares the residuals so to undo this squaring we need to take the square-root as in:

```
def residuals(p, data):
   x, y = data
   ym = a + b*x              # Model data
   r = abs(y - ym)           # Absolute residuals for robust fitting
   return numpy.sqrt(r)      # Squaring is done in Fitter routine
```

Note: A residuals function should always return a NumPy double-precision floating-point number array (i.e. dtype='d').

Note: It is also possible to write residual functions that represent objective functions used in orthogonal fit procedures where both variables **x** and **y** have errors. We will give some examples in the section about orthogonal fitting.

Artificial data for experiments

For experiments with least square fits, it is often convenient to start with artificial data which resembles the model with certain parameters, and add some Gaussian distributed noise to the y values. This is what we have done in the next couple of lines:

The number of data points and the mean and width of the normal distribution which we use to add some noise:

```
N = 50
mean = 0.0; sigma = 0.6
```

Finally we create a range of x values and use our model with arbitrary model parameters to create y values:

```
xstart = 2.0; xend = 10.0
x = numpy.linspace(3.0, 10.0, N)
paramsreal = [1.0, 1.0]
noise = numpy.random.normal(mean, sigma, N)
y = model(paramsreal, x) + noise
```

Initial parameter estimates

Now we have to tell the constructor of the *Fitter* object what the residuals function is and which arrays the residuals function needs. To create a Fitter object we use the line:

```
fitobj = kmpfit.Fitter(residuals=residuals, data=(x,y))
```

Least squares fitters need initial estimates of the model parameters. As you probably know, our problem is an example of 'linear regression' and this category of models have best fit parameters that can be calculated analytically. Then the fit results are not very sensitive to the initial values you supply. So set the values of our initial parameters in the model (a,b) to (0,0). Use these values in the call to `Fitter.fit()`. The result of the fit is stored in attributes of the Fitter object (*fitobj*). We show the use of attributes *status*, *message*, and *params*. This last attribute stores the 'best fit' parameters, it has the same type as the sequence with the initial parameter (i.e. NumPy array, list or tuple):

```
paramsinitial = (0.0, 0.0)
fitobj.fit(params0=paramsinitial)
if (fitobj.status <= 0):
   print 'Error message = ', fitobj.message
else:
   print "Optimal parameters: ", fitobj.params
```

Below we show a complete example. If you run it, you should get a plot like the one below the source code. It will not be exactly the same because we used a random number generator to add some noise to the data. The plots are created with Matplotlib. A plot is a simple but effective tool to qualify a fit. For most of the examples in this tutorial a plot is included.

Example: kmpfit_example_simple.py - Simple use of kmpfit

```
#!/usr/bin/env python
#------------------------------------------------------------
# Purpose: Demonstrate simple use of fitter routine
#
# Vog, 12 Nov 2011
#------------------------------------------------------------
import numpy
from matplotlib.pyplot import figure, show, rc
from kapteyn import kmpfit


# The model
#==========
def model(p, x):
   a,b = p
   y = a + b*x
   return y


# The residual function
#======================
def residuals(p, data):
   x, y = data                    # 'data' is a tuple given by programmer
   return y - model(p,x)


# Artificial data
#================
N = 50                            # Number of data points
mean = 0.0; sigma = 0.6           # Characteristics of the noise we add
xstart = 2.0; xend = 10.0
```

```
x = numpy.linspace(3.0, 10.0, N)
paramsreal = [1.0, 1.0]
noise = numpy.random.normal(mean, sigma, N)
y = model(paramsreal, x) + noise


# Prepare a 'Fitter' object'
#===========================
paramsinitial = (0.0, 0.0)
fitobj = kmpfit.Fitter(residuals=residuals, data=(x,y))

try:
   fitobj.fit(params0=paramsinitial)
except Exception, mes:
   print "Something wrong with fit: ", mes
   raise SystemExit

print "Fit status: ", fitobj.message
print "Best-fit parameters:      ", fitobj.params
print "Covariance errors:        ", fitobj.xerror
print "Standard errors           ", fitobj.stderr
print "Chi^2 min:                ", fitobj.chi2_min
print "Reduced Chi^2:            ", fitobj.rchi2_min
print "Iterations:               ", fitobj.niter
print "Number of function calls: ", fitobj.nfev
print "Number of free pars.:     ", fitobj.nfree
print "Degrees of freedom:       ", fitobj.dof
print "Number of pegged pars.:   ", fitobj.npegged
print "Covariance matrix:\n", fitobj.covar


# Plot the result
#================
rc('font', size=10)
rc('legend', fontsize=8)
fig = figure()
xp = numpy.linspace(xstart-1, xend+1, 200)
frame = fig.add_subplot(1,1,1, aspect=1.0)
frame.plot(x, y, 'ro', label="Data")
frame.plot(xp, model(fitobj.params,xp), 'm', lw=1, label="Fit with kmpfit")
frame.plot(xp, model(paramsreal,xp), 'g', label="The model")
frame.set_xlabel("X")
frame.set_ylabel("Response data")
frame.set_title("Least-squares fit to noisy data using KMPFIT", fontsize=10)
s = "Model: Y = a + b*X    real:(a,b)=(%.2g,%.2g), fit:(a,b)=(%.2g,%.2g)"%\
     (paramsreal[0],paramsreal[1], fitobj.params[0],fitobj.params[1])
frame.text(0.95, 0.02, s, color='k', fontsize=7,
           ha='right', transform=frame.transAxes)
frame.set_xlim(0,12)
frame.set_ylim(0,None)
frame.grid(True)
leg = frame.legend(loc=2)
show()
```

2.3.3 Function `simplefit()`

For simple fit problems we provide a simple interface. It is a function which is used as follows:

```
>>> p0 = (0,0)
>>> fitobj = kmpfit.simplefit(model, p0, x, y, err=err, xtol=1e-8)
>>> print fitobj.params
```

Argument `model` is a function, just like the model in the previous section. `p0` is a sequence with initial values with a length equal to the number of parameters that is defined in your model. Argument `x` and `y` are the arrays or lists that represent your measurement data. Argument `err` is an array with 1 σ errors, one for each data point. Then you can enter values to tune the fit routine with keyword arguments (e.g. *gtol*, *xtol*, etc.). In the next example we demonstrate how to use lists for your data points, how to make an unweighted fit and how to print the right parameter uncertainties. For an explanation of parameter uncertainties, see section *Standard errors of best-fit values*.

The advantages of this function:

- You need only to worry about a model function
- No need to create a *Fitter* object first
- Direct input of relevant arrays
- As a result you get a Fitter object with all the attributes
- It is (still) possible to tune the fit routine with keyword arguments, no limitations here.

Example: `kmpfit_example_easyinterface.py` **- Simple function**

```
#!/usr/bin/env python
#------------------------------------------------------------
# Purpose: Demonstrate simple use of fitter routine
#
# Vog, 24 Nov 2011
#------------------------------------------------------------
import numpy
from matplotlib.pyplot import figure, show
from kapteyn import kmpfit

# The model:
def model(p, x):
   a, b = p
   y = a + b*x
   return y


# Artificial data
N = 50                               # Number of data points
mean = 0.0; sigma = 0.6              # Characteristics of the noise we add
x = numpy.linspace(2, 10, N)
paramsreal = [1.0, 1.0]
noise = numpy.random.normal(mean, sigma, N)
y = model(paramsreal, x) + noise
err = numpy.random.normal(mean, sigma, N)


# Simple interface
p0 = (0,0)
xl = range(10)
yl = [k*0.5 for k in xl]
fitobj = kmpfit.simplefit(model, p0, xl, yl)
print "Best fit parameters:", fitobj.params
print "Parameter errors:  :", fitobj.stderr

fitobj = kmpfit.simplefit(model, p0, x, y, err=err, xtol=1e-8)
```

```
print "Best fit parameters:", fitobj.params
print "Parameter errors:  :", fitobj.xerror

fitobj = kmpfit.simplefit(model, p0, x, y, maxiter=100)
print "Best fit parameters:", fitobj.params
print "Parameter errors:  :", fitobj.stderr
```

2.3.4 Standard errors of best-fit values

With the estimation of errors on the best-fit parameters we get an idea how good a fit is. Usually these errors are called standard errors, but often programs call these errors also standard deviations. For nonlinear least-squares routines, these errors are based on mathematical simplifications and are therefore often called *asymptotic* or *approximate* standard errors.

The standard error (often denoted by SE) is a measure of the average amount that the model over- or under-predicts.

According to *[Bev]* , the standard error is an uncertainty which corresponds to an increase of χ^2 by 1. That implies that if we we add the standard error σ_i to its corresponding parameter, fix it in a second fit and fit again, the value of χ^2 will be increased by 1.

$$\chi^2(p_i + \sigma_i) = \chi^2(p_i) + 1 \tag{2.14}$$

The next example shows this behaviour. We tested it with the first parameter fixed and a second time with the second parameter fixed. The example also shows how to set parameters to 'fixed' in *kmpfit*. The model is a straight line. If you run the example you will see that it shows exactly the behaviour as in (2.14). This proves that the covariance matrix (explained later) of *kmpfit* can be used to derive standard errors. Note the use of the `parinfo` attribute of the *Fitter* object to fix parameters. One can use an index to set values for one parameter or one can set the values for all parameters. These values are given as a Python dictionary. An easy way to create a dictionary is to use Python's `dict()` function.

Example: `kmpfit_errors_chi2delta.py` - Meaning of asymptotic errors

```
#!/usr/bin/env python
#------------------------------------------------------------
# Purpose: Demonstrate, using kmpfit, that if you find best-fit
# parameters, the errors derived from the covariance matrix
# correspond to an increase in chi^2 of 1.
# Vog, 23 Nov 2011
#------------------------------------------------------------
import numpy
from matplotlib.pyplot import figure, show, rc
from numpy.random import normal, randint
from kapteyn import kmpfit


def residuals(p, data):
   x, y, err = data
   a, b = p
   model = a + b*x
   return (y-model)/err


# Artificial data
#----------------
N = 100
a0 = 2; b0 = 3
x = numpy.linspace(0.0, 2.0, N)
```

```
y = a0 + b0*x + normal(0.0, 0.4, N)  # Mean,sigma,N
derr = normal(0.0, 0.5, N)
err = 0.9+derr

fitobj = kmpfit.Fitter(residuals=residuals, data=(x, y, err))
fitobj.fit(params0=[1,1])

if (fitobj.status <= 0):
   print 'error message =', fitobj.errmsg
   raise SystemExit

print "\n\n======== Results kmpfit for Y = A + B*X ========="
print "Params:        ", fitobj.params
print "Errors from covariance matrix         : ", fitobj.xerror
print "Uncertainties assuming reduced Chi^2=1: ", fitobj.stderr
print "Chi^2 min:     ", fitobj.chi2_min

p1, p2 = fitobj.params
e1, e2 = fitobj.xerror
# Next we take one of the parameters to be fixed and change its value
# with the amount of one of the estimated errors (covariance, scaled or bootstrap)
# If we fit again, then, according to Bevington (Data Reduction and Error
# Analysis for the Physical Sciences  Section 11-5), one should expect the
# Chi square value to increase with 1.0

fitobj.parinfo[0] = dict(fixed=True)
fitobj.fit(params0=[p1+e1,1])
print "\nFix first parameter and set its value to fitted value+error"
print "Params:        ", fitobj.params
print "Chi^2 min:     ", fitobj.chi2_min
print "Errors from covariance matrix         : ", fitobj.xerror

fitobj.parinfo = [{'fixed':False}, {'fixed':True}]
fitobj.fit(params0=[1, p2+e2])
print "\nFix second parameter and set its value to fitted value+error"
print "Params:        ", fitobj.params
print "Chi^2 min:     ", fitobj.chi2_min
print "Errors from covariance matrix         : ", fitobj.xerror
```

The results for an arbitrary run:

```
======== Results kmpfit for Y = A + B*X =========
Params:          [2.0104270702631712, 2.94745915643011]
Errors from covariance matrix         :  [ 0.05779471  0.06337059]
Uncertainties assuming reduced Chi^2=1:  [ 0.04398439  0.04822789]
Chi^2 min:      56.7606029739

Fix first parameter and set its value to fitted value+error
Params:          [2.0682217814912143, 2.896736695408106]
Chi^2 min:      57.7606030002
Errors from covariance matrix         :  [ 0.          0.03798767]

Fix second parameter and set its value to fitted value+error
Params:          [1.9641675954511788, 3.0108297500339498]
Chi^2 min:      57.760602835
Errors from covariance matrix         :  [ 0.0346452  0.        ]
```

As you can see, the value of chi-square has increased with ~1.

Standard errors in weighted fits

In the literature *[Num]* we can find analytical expressions for the standard errors of weighted fits for standard linear regression. We want to discuss the derivation of analytical errors for weighted fits to demonstrate that these errors are also represented by the elements of the so-called variance-covariance matrix (or just covariance matrix), which is also a result of a fit with *kmpfit* (attribute `Fitter.covar`). How should we interpret these errors? For instance in Numerical Recipes, *[Num]* we find the expressions for the best fit parameters of a model $y = a + bx$ Use the *chi-squared* objective function:

$$\chi^2([a,b],x) = \sum_{i=0}^{N-1} \left(\frac{y_i - a - bx_i}{\sigma_i}\right)^2 \tag{2.15}$$

To find analytical expressions for the best-fit values of *a* and *b*, we need to take derivatives of this objective function:

$$\begin{aligned} \frac{\partial \chi^2}{\partial a} &= -2 \sum_{i=0}^{N-1} \frac{y_i - a - bx_i}{\sigma_i^2} \\ \frac{\partial \chi^2}{\partial b} &= -2 \sum_{i=0}^{N-1} \frac{x_i(y_i - a - bx_i)}{\sigma_i^2} \end{aligned} \tag{2.16}$$

Define:

$$\begin{aligned} S \equiv \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} \;\; S_x \equiv \sum_{i=0}^{N-1} \frac{x_i}{\sigma_i^2} \;\; S_y \equiv \sum_{i=0}^{N-1} \frac{y_i}{\sigma_i^2} \\ S_{xx} \equiv \sum_{i=0}^{N-1} \frac{x_i^2}{\sigma_i^2} \;\; S_{xy} \equiv \sum_{i=0}^{N-1} \frac{x_i y_i}{\sigma_i^2} \end{aligned} \tag{2.17}$$

Then one can rewrite (2.16) into:

$$\begin{aligned} aS + bS_x &= S_y \\ aS_x + bS_{xx} &= S_{xy} \end{aligned} \tag{2.18}$$

which is in matrix notation:

$$\begin{bmatrix} S & S_x \\ S_x & S_{xx} \end{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} S_y \\ S_{xy} \end{pmatrix} \tag{2.19}$$

If we define:

$$C = \frac{1}{SS_{xx} - (S_x)^2} \begin{bmatrix} S_{xx} & -S_x \\ -S_x & S \end{bmatrix} \tag{2.20}$$

which gives the solution:

$$\begin{pmatrix} a \\ b \end{pmatrix} = C \begin{pmatrix} S_y \\ S_{xy} \end{pmatrix} \tag{2.21}$$

Define:

$$\Delta \equiv SS_{xx} - (S_x)^2 \tag{2.22}$$

The solutions for *a* and *b* are:

$$\begin{aligned} a &= \frac{S_{xx}S_y - S_x S_{xy}}{\Delta} \\ b &= \frac{SS_{xy} - S_x S_y}{\Delta} \end{aligned} \tag{2.23}$$

For the standard errors we will derive the error in parameter *a* and *b*. The error in *a* is by the law of propagation of errors:

$$\sigma_a^2 = \sum_i \sigma_i^2 \left(\frac{\partial a}{\partial y_i}\right)^2 \tag{2.24}$$

>From (2.23) and (2.17) we derive:

$$\frac{\partial a}{\partial y_i} = \frac{\frac{S_{xx}}{\sigma_i^2} - \frac{S_x x_i}{\sigma_i^2}}{\Delta} = \frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta} \tag{2.25}$$

With (2.24) we find

$$\begin{aligned}
\sigma_a^2 &= \sum_i \sigma_i^2 \left(\frac{\partial a}{\partial y_i}\right)^2 \\
&= \sum_i \sigma_i^2 \left(\frac{S_{xx} - S_x x_i}{\sigma_i^2 \Delta}\right)^2 \\
&= \frac{1}{\Delta^2}\left\{S_{xx}^2 \Sigma \frac{1}{\sigma_i^2} - 2 S_x S_{xx} \Sigma \frac{x_i}{\sigma_i^2} + S_x^2 \Sigma \frac{x_i^2}{\sigma_i^2}\right\} \\
&= \frac{1}{\Delta^2}\left\{S_{xx}^2 S - 2 S_x S_{xx} S_x + S_{xx} S_x^2\right\} \\
&= \frac{1}{\Delta^2}\left\{S_{xx}(S_{xx} S - S_x^2)\right\} \\
&= \frac{1}{\Delta^2} S_{xx} \Delta \\
&= \frac{S_{xx}}{\Delta}
\end{aligned} \tag{2.26}$$

Applying the same procedure to *b*:

$$\frac{\partial b}{\partial y_i} = \frac{\frac{S x_i}{\sigma_i^2} - \frac{S_x}{\sigma_i^2}}{\Delta} = \frac{S x_i - S_x}{\sigma_i^2 \Delta} \tag{2.27}$$

With (2.24) we find

$$\begin{aligned}
\sigma_b^2 &= \sum_i \sigma_i^2 \left(\frac{\partial b}{\partial y_i}\right)^2 \\
&= \sum_i \sigma_i^2 \left(\frac{S x_i - S_x}{\sigma_i^2 \Delta}\right)^2 \\
&= \frac{1}{\Delta^2}\left\{S^2 \Sigma \frac{x_i^2}{\sigma_i^2} - 2 S_x S \Sigma \frac{x_i^2}{\sigma_i^2} + S_x^2 \Sigma \frac{x_i^2}{\sigma_i^2}\right\} \\
&= \frac{1}{\Delta^2}\left\{S^2 S - 2 S_x S S_x + S_x^2 S\right\} \\
&= \frac{1}{\Delta^2}\left\{S(S_{xx} S - S_x^2)\right\} \\
&= \frac{1}{\Delta^2} S \Delta \\
&= \frac{S}{\Delta}
\end{aligned} \tag{2.28}$$

To summarize:

$$\boxed{\sigma_a = \sqrt{\frac{S_{xx}}{\Delta}}} \tag{2.29}$$

$$\boxed{\sigma_b = \sqrt{\frac{S}{\Delta}}}$$

A classical implementation to find analytical best-fit parameters using NumPy is as follows:

```python
def lingres(xa, ya, err):
   w = numpy.where(err==0.0, 0.0, 1.0/(err*err))
   Sum   =  w.sum()
   sumX  = (w*xa).sum()
   sumY  = (w*ya).sum()
   sumX2 = (w*xa*xa).sum()
   sumY2 = (w*ya*ya).sum()
   sumXY = (w*xa*ya).sum()
   delta = Sum * sumX2 - sumX * sumX
   a = (sumX2*sumY - sumX*sumXY) / delta
   b = (sumXY*Sum - sumX*sumY) / delta
   siga = numpy.sqrt(abs(sumX2/delta))
   sigb = numpy.sqrt(abs(Sum/delta))
   return a, b, siga, sigb, delta, Sum, sumX2, sumX
```

Note that these formulas are susceptible to roundoff error and Numerical Recipes derives alternative formulas (Section 15.2). However, our functions work with double precision numbers and we didn't (yet) encounter a situation where roundoff errors were obvious.

If we compare these results with the elements of the covariance matrix in (2.20), then we observe that the expressions for the parameter variances, are the square root of the diagonal values of this matrix. The co-variance between *a* and *b* can be calculated also and the formula turns out to be the same as the off-diagonal elements of the covariance matrix. This value is:

$$\mathrm{Cov}(a,b) = C_{12} = C_{21} = \frac{-S_x}{\Delta} \tag{2.30}$$

It is easy to demonstrate that these errors are the same as those we find with *kmpfit* in attribute `xerror`, which are the square-root diagonal values of the covariance matrix in attribute `covar`.

The covariance matrix elements C_{jk} for best-fit parameters **p** can be written as:

$$C_{jk} = \sum_{i=0}^{i=N} \sigma_i^2 \left(\frac{\partial p_j}{\partial y_i}\right)\left(\frac{\partial p_k}{\partial y_i}\right) \tag{2.31}$$

where we used *j* to indicate the matrix row and *k* the matrix column. If *j=k* then:

$$C_{jj} = \sum_{i=0}^{i=N} \sigma_i^2 \left(\frac{\partial p_j}{\partial y_i}\right)^2 \tag{2.32}$$

from which follows that the square root of the diagonal elements of the covariance matrix are the estimates of the best-fit parameter uncertainties.

Note:

- Parameter variances and covariance between parameters can be read from a covariance matrix. This is true for any model, not just a straight line. It is also true for models that are non-linear in their parameters.
- The covariance matrix C is in stored as an attribute of the 'kmpfit.Fitter' object The attribute is called `covar`.

- Error estimates for best-fit parameter are stored as an attribute of the 'kmpfit.Fitter' object. The attribute is called `xerror`

Example program `kmpfit_linearreg.py` compares the analytical covariance matrix with the *kmpfit* version for linear regression, using the previously derived formulas in this section. The output of an arbitrary example run demonstrates the similarity between the analytical and the *kmpfit* method:

Example: `kmpfit_linearreg.py` **- Compare output analytical method and kmpfit**

```
-- Results analytical solution:
Best fit parameters:                         [0.57857142857143595, 5.5285714285714258]
Parameter errors weighted fit:               [0.84515425472851657, 0.1889822365046136]
Parameter errors un-/relative weighted fit:  [1.0696652156022404, 0.2391844135253578]
Minimum chi^2:                               8.00928571429
Covariance matrix:
0.714285714286 -0.142857142857
-0.142857142857 0.0357142857143

-- Results kmpfit:
Best-fit parameters:                         [0.57857145533008425, 5.5285714226701863]
Parameter errors weighted fit:               [ 0.84515434  0.18898225]
Parameter errors un-/relative weighted fit:  [ 1.06966532  0.23918443]
Minimum chi^2:                               8.00928571429
Covariance matrix:
[[ 0.71428585 -0.14285717]
[-0.14285717  0.03571429]]
```

We observe:

- The analytical values of the best-fit parameters and those from *kmpfit* correspond. The same applies to the errors for the unweighted fit/fit with relative weights.

When to use weights?

Sometimes there is a good reason to use a fit method that can deal with weights. Usually you assign weights if you have additional knowledge about your measurements. Some points get more weight if they are more reliable than others. Therefore you should expect that the best-fit parameters are different between weighted and un-weighted fits. Also the accuracy of the results will improve, because besides the data you are using the quality of the data. The difference in best-fit parameters and the quality of the results is shown with program `kmpfit_compare_wei_unwei.py`

Example: `kmpfit_compare_wei_unwei.py` **- Compare output for unweighted (unit weighting) and weighted fit**

```
Data x: [ 1.  2.  3.  4.  5.  6.  7.]
Data y: [  6.9   11.95  16.8   22.5   26.2   33.5   41.  ]
Errors: [ 0.05  0.1   0.2   0.5   0.8   1.5   4.  ]

-- Results kmpfit unit weighting wi=1.0:
Best-fit parameters:                         [0.57857145533008425, 5.5285714226701863]
Parameter errors weighted fit:               [ 0.84515434  0.18898225]
Minimum chi^2:                               8.00928571429
Covariance matrix:
[[ 0.71428585 -0.14285717]
[-0.14285717  0.03571429]]

-- Results kmpfit with weights:
Best-fit parameters:                         [1.8705399823164173, 5.0290902421858439]
Parameter errors weighted fit:               [ 0.09922304  0.06751229]
```

```
Minimum chi^2:                                    4.66545480308
Covariance matrix:
[[ 0.00984521 -0.00602421]
[-0.00602421  0.00455791]]
```

If you examine the residuals function in this program, you will observe that we use a weight of $1/err_i$ in the residuals function, which is squared by *kmpfit*, so in fact the weighting is $1/\sigma_i^2$. First we set all the errors to 1.0. This is called *unit weighting* and effectively this fit does not weight at all. The second fit has different weights. Important is the observation that these weights can be relative. Then they contain information about the quality of the data but do not necessarily contain correct information about the errors on the data points and therefore give incorrect errors on the parameter estimates. This is shown in the same program `kmpfit_compare_wei_unwei.py` where we scaled the errors with a factor 10. The errors in the parameter estimates are increased by a factor 10.

Example: `kmpfit_compare_wei_unwei.py` **- Compare output for unweighted (unit weighting) and weighted fit**

```
-- Results kmpfit with scaled individual errors (factor=10):
Best-fit parameters:                              [1.870539984453957, 5.0290902408769238]
Parameter errors weighted fit:                    [ 0.99223048  0.6751229 ]
Minimum chi^2:                                    0.0466545480308
Covariance matrix:
[[ 0.98452132 -0.60242076]
[-0.60242076  0.45579092]]
```

This demonstrates that if weights are relative or when unit weighting is applied, one cannot rely on the covariance errors to represent real errors on the parameter estimates. The covariance errors are still based on a change in χ^2 of 1.0, but the weights do not represent the variances of the data correctly.

To summarize the weighting schemes:

- *Unweighted* or *unit weighting*. Set $w_i = 1/\sigma_i^2$ to 1.0
- *Relative weighting*. Set $w_i = 1/\sigma_i^2$ but the errors on the parameter estimates in *kmpfit*'s attribute `xerror` cannot be used.
- *Statistical weighting*. Set $w_i = 1/\sigma_i^2$. The errors on the parameter estimates in *kmpfit*'s attribute `xerror` are correct. An important assumption of this method is that the error distribution of the measured data is Gaussian and that the data errors are measured accurately (absolute uncertainties).
- Other weighting schemes like Poisson weighting $w_i = 1/y_i$

Reduced chi squared

>From the theory of maximum likelihood we find that for a least squares solution we need to maximize the probability that a measurement y_i with given σ_i is in a a small interval dy_i around y_i by minimizing the sum chi squared *[Ds1]* :

$$\chi^2 = \sum_{i=0}^{N-1} \left(\frac{\Delta y_i}{\sigma_i}\right)^2 = \sum_{i=0}^{N-1} \frac{(y_i - f(x_i))^2}{\sigma_i^2} \tag{2.33}$$

with:

- N is the number of data points
- y_i the measured data at x_i
- σ_i is the standard deviation of measurement i
- f is the model for which we want to find the best-fit parameters.

The sum is often called chi squared because it follows the χ^2 distribution if we repeat the experiment to get new measurements. The expectation value of χ^2 is (see proof in *[Ds3]*):

$$\langle\chi^2\rangle = N - n \tag{2.34}$$

where *n* is the number of free parameters in the fit. The *reduced* chi squared χ^2_ν is defined as:

$$\chi^2_\nu = \frac{\chi^2}{N-n} = \frac{\chi^2}{\nu} \tag{2.35}$$

where $\nu = N - n$. From (2.34) we derive for the expectation value of χ^2_ν:

$$\langle\chi^2_\nu\rangle = 1 \tag{2.36}$$

Fitting with (2.33) as objective function is often called chi squared fitting. The value of χ^2_ν is a measure of the *goodness of fit* and is returned by *kmpfit* in a Fitter object as attribute `rchi2_min`. The number of degrees of freedom is stored in attribute `dof`.

Note:

- χ^2_ν follows the chi square statistic. This statistic measures both the spread of the data and the accuracy of the fit.
- The reduced chi squared χ^2_ν is a measure of the goodness of fit. Its expectation value is 1.
- A value of $\chi^2_\nu \approx 1$ indicates that there is a match between measurements, best-fit parameters and error variances.
- A large value of χ^2_ν (e.g. > 1.5) indicates a poor model fit.
- A $\chi^2_\nu < 1$ indicates that probably the error variance has been over-estimated.
- A $\chi^2_\nu > 1$ indicates that probably the error variance has been under-estimated.

In the literature we find relations between the standard deviation of the sample and the true standard deviation of the underlying distribution . For least squares analysis we replace the average value of *y* (i.e. $\bar{y}$) in those formulas by the model with the best-fit parameters $f(p, x)$.

What should we expect of the variance σ_i compared to the sample deviations for each sample point? Assume we have N data points and each data point has an individual error of σ_i. >From (2.34) we have:

$$\left\langle \sum_{i=0}^{N-1} \frac{\left(y_i - f(x_i)\right)^2}{\sigma_i^2} \right\rangle = N - n \tag{2.37}$$

With the observation that the expectation value of each of the *N* terms is the same we derive for each data point:

$$\left\langle \left(y_i - f(x_i)\right)^2 \right\rangle = (1 - \frac{n}{N})\sigma_i \tag{2.38}$$

So for a good fit the true deviation of a measurement σ_i for large *N* is almost equal to the deviation between data point and fit. The less the scatter of data about the best fit, the smaller σ_i should be.

The *sample variance*, s_y^2 is then written as *[Ds2]* :

$$s_y^2 = \frac{1}{N-n}\sum_i (y_i - f(x_i))^2 \tag{2.39}$$

If we replace all σ_i with σ_y in equation (2.37), then we derive a familiar relationship:

$$\frac{s_y^2}{\sigma_y^2} = \chi_\nu \rightarrow \langle s_y^2\rangle = \sigma_y^2 \tag{2.40}$$

so that the value of s_y^2 of the measurements is an unbiased estimate of the true variance σ_y^2 of the underlying distribution. For an unbiased estimator, the expected value and the true value are the same.

The weighted version of the sample variance is defined as:

$$sw_y^2 = \frac{\frac{1}{N-n}\sum\limits_i w_i(y_i - f(x_i))^2}{\frac{1}{N}\sum\limits_i w_i} \tag{2.41}$$

If we use $1/\sigma_i^2$ as weight, then:

$$sw_y^2 \times \frac{1}{N}\sum_i \frac{1}{\sigma_i^2} = \chi_\nu^2 \tag{2.42}$$

Bevington *[Bev]* defines the weighted average of the individual variances $\bar{\sigma}_i^2$ as:

$$\bar{\sigma}_i^2 = \frac{\frac{1}{N}\sum\limits_i \left(\frac{1}{\sigma_i^2}\sigma_i^2\right)}{\frac{1}{N}\sum\limits_i \frac{1}{\sigma_i^2}} = \frac{1}{\frac{1}{N}\sum\limits_i \frac{1}{\sigma_i^2}} \tag{2.43}$$

Then:

$$\frac{sw_y^2}{\bar{\sigma}_i^2} = \chi_\nu^2 \tag{2.44}$$

If we set all weights to the same value $w_i = 1/\sigma_y^2$ then $sw_y = s_y$ and:

$$\frac{s_y^2}{\sigma_y^2} = \chi_\nu^2 \tag{2.45}$$

which is consistent with (2.40).

For chi squared fitting it is therefore important to have correct values for σ_i. Over-estimated values give a χ_ν^2 which is smaller than 1 and under-estimated values give a value bigger than 1 (If you get very large values, then probably fit and data are not in agreement). If the values for σ_i are unreliable then also the error estimates of the best-fit parameters are unreliable, because they are functions of σ_i (see e.g. the analytical expressions for these errors in a linear regression in (2.29)). According to equations (2.44) and (2.45) it is reasonable then to scale the values of σ_i in a way that we force χ_ν^2 to take its expectation value of 1. Then one gets values for the errors in `stderr` which are insensitive to arbitrary scaling factors of the weights.

We noted earlier that scaling the weights does not change the values of the best-fit parameters but they affect the values of the parameter error estimates because they depend on the values of σ_i. If for example values of σ_i are all too small with a factor 2 with respect to those that make $\chi_\nu = 1$. Then the errors in the parameter estimates are to small with a factor 2x2=4 (see e.g. (2.26) and (2.28) for the straight line model). The value of χ_ν will be 2x2=4. So to correct the errors on the parameter estimates, we can multiply the variances with the value of χ_ν. If we recall equation (2.32), then we see that this scaling can be applied to arbitrary models. This scaling is exactly what happens in *kmpfit* for the values in attribute `stderr`.

In *kmpfit* we use the unit- or relative weights as given by the user and calculate the value of χ_ν. The asymptotic standard errors in `xerror` are then multiplied by the square root of the value of χ_ν and stored in attribute `stderr`. We demonstrate this with the output of a small example (`kmpfit_compare_wei_unwei.py`) with data from *[Wol]*:

Example: `kmpfit_compare_wei_unwei.py` **- Compare output for unweighted (unit weighting) and weighted fit**

```
Data x: [ 1.  2.  3.  4.  5.  6.  7.]
Data y: [  6.9   11.95  16.8   22.5   26.2   33.5   41.  ]
Errors: [ 0.05  0.1   0.2   0.5   0.8   1.5   4.  ]

New array with measurement errors, scaled with factor 0.933091 to give
a reduced chi-squared of 1.0:
[ 0.04829832  0.09659663  0.19319327  0.48298317  0.77277307  1.4489495
3.86386534]

-- Results kmpfit with scaled individual errors to force red_chi2=1:
Best-fit parameters:                              [1.8705399822570359, 5.029090242191204]
Parameter errors using measurement uncertainties:  [ 0.09584612  0.0652146 ]
Parameter errors unit-/relative weighted fit:      [ 0.09584612  0.0652146 ]
Minimum chi^2:                                    5.0
Minimum reduced chi^2:                            1.0
Covariance matrix:
[[ 0.00918648 -0.00562113]
[-0.00562113  0.00425294]]
```

The next code example is a small script that shows that the scaled error estimates in attribute `stderr` for unit- and relative weighting are realistic if we compare them to errors found with a Monte Carlo method. We start with values of σ_i that are under-estimated. This results in a value for χ_ν which is too low. The re-scaled errors in `stderr` match with those that are estimated with the Monte-Carlo method. In the example we used the Bootstrap Method. The plot shows the fit and the bootstrap distributions of parameter *A* and *B*. We will explain the Bootstrap Method in the next section.

Example: kmpfit_unweighted_bootstrap_plot.py - How to deal with unweighted fits

```python
#!/usr/bin/env python
#------------------------------------------------------------
# Purpose: Demonstrate that the scaled covariance errors for
# unweighted fits are comparable to errors we find with
# a bootstrap method.
# Vog, 24 Nov 2011
#------------------------------------------------------------

import numpy
from matplotlib.pyplot import figure, show, rc
from numpy.random import normal, randint
from kapteyn import kmpfit

# Residual and model in 1 function. Model is straight line
def residuals(p, data):
   x, y, err = data
   a, b = p
   model = a + b*x
   return (y-model)/err

# Artificial data
N = 100
a0 = 0; b0 = 1.2
x = numpy.linspace(0.0, 2.0, N)
y = a0 + b0*x + normal(0.0, 0.4, N)   # Mean,sigma,N
err = numpy.ones(N)                   # All weights equal to 1

# Prepare fit routine
fitobj = kmpfit.Fitter(residuals=residuals, data=(x, y, err))
try:
   fitobj.fit(params0=[1,1])
except Exception, mes:
   print "Something wrong with fit: ", mes
   raise SystemExit

print "\n\n======== Results kmpfit unweighted fit ========="
print "Params:          ", fitobj.params
print "Errors from covariance matrix          : ", fitobj.xerror
print "Uncertainties assuming reduced Chi^2=1: ", fitobj.stderr
print "Chi^2 min:       ", fitobj.chi2_min
print "Reduced Chi^2: ", fitobj.rchi2_min
print "Iterations:      ", fitobj.niter
print "Function ev:     ", fitobj.nfev
print "Status:          ", fitobj.status
print "Status Message:", fitobj.message

# Bootstrap method to find uncertainties
A0, B0 = fitobj.params
xr = x.copy()
yr = y.copy()
ery = err.copy()
fitobj = kmpfit.Fitter(residuals=residuals, data=(xr, yr, ery))
slopes = []
offsets = []
trials = 10000                # Number of synthetic data sets
for i in range(trials):       # Start loop over pseudo sample
   indx = randint(0, N, N)    # Do the resampling using an RNG
   xr[:] = x[indx]
   yr[:] = y[indx]
   ery[:] = err[indx]

   # Only do a regression if there are at least two different
   # data points in the pseudo sample
   ok = (xr != xr[0]).any()

   if (not ok):
      print "All elements are the same. Invalid sample."
      print xr, yr
   else:
      fitobj.fit(params0=[1,1])
      offs, slope = fitobj.params
      slopes.append(slope)
      offsets.append(offs)

slopes = numpy.array(slopes) - B0
offsets = numpy.array(offsets) - A0
sigmaA0, sigmaB0 = offsets.std(), slopes.std()
print "Bootstrap errors in A, B:", sigmaA0, sigmaB0

# Plot results
rc('font', size=7)
rc('legend', fontsize=6)
fig = figure(figsize=(7,4))
fig.subplots_adjust(left=0.08, wspace=0.3, right=0.94)
frame = fig.add_subplot(1,3,1, aspect=1.0, adjustable='datalim')
frame.plot(x, y, 'bo', label='Observed data')
frame.plot(x, a0+b0*x, 'r', label='True: Y=%.1f+%.1fX'%(a0,b0))
frame.plot(x, A0+B0*x, '--c', alpha=0.5, lw=4, label='kmpfit')
frame.set_xlabel("X"); frame.set_ylabel("Y")
frame.set_title("Unweighted fit Y=A+B*X")
frame.grid(True)
frame.legend(loc='upper left')

ranges = [(offsets.min(), offsets.max()),(slopes.min(), slopes.max())]
nb = 40                                        # Number of bins in histogram
for i,sigma in enumerate([sigmaA0, sigmaB0]):
   framehist = fig.add_subplot(1, 3, 2+i)
   range = ranges[i]                           # (X) Range in histogram
   framehist.hist(slopes, bins=nb, range=range, fc='g')
   binwidth  = (range[1]-range[0])/nb          # Get width of one bin
   area = trials * binwidth                    # trials is total number of counts
   mu = 0.0
   amplitude = area / (numpy.sqrt(2.0*numpy.pi)*sigma)
   x = numpy.linspace(range[0], range[1], 100)
   y = amplitude * numpy.exp(-(x-mu)*(x-mu)/(2.0*sigma*sigma))
   framehist.plot(x, y, 'r')
   if i == 0:
      lab = "$A_i-A_0$"
      title = "Distribution synthetic A"
   else:
      lab = "$B_i-B_0$"
      title = "Distribution synthetic B"
   framehist.set_xlabel(lab)
   framehist.set_ylabel("Counts")
   framehist.set_title(title)

show()
```

Bootstrap Method

We need to discuss the bootstrap method, that we used in the last script, in some detail. Bootstrap is a tool which estimates standard errors of parameter estimates by generating synthetic data sets with samples drawn with replacement from the measured data and repeating the fit process with this synthetic data.

Your data realizes a set of best-fit parameters, say $p_{(0)}$. This data set is one of many different data sets that represent the 'true' parameter set p_{true} . Each data set will give a different set of fitted parameters $p_{(i)}$. These parameter sets follow some probability distribution in the *n* dimensional space of all possible parameter sets. To find the uncertainties in the fitted parameters we need to know the distribution of $p_{(i)} - p_{true}$ *[Num]*. In Monte Carlo simulations of synthetic data sets we assume that the shape of the distribution of Monte Carlo set $p_{(i)} - p_0$ is equal to the shape of the real world set $p_{(i)} - p_{true}$

The *Bootstrap Method* *[Num]* uses the data set that you used to find the best-fit parameters. We generate different synthetic data sets, all with *N* data points, by randomly drawing *N* data points, with replacement from the original data. In Python we realize this as follows:

```
indx = randint(0, N, N)     # Do the re-sampling using an RNG
xr[:] = x[indx]
yr[:] = y[indx]
ery[:] = err[indx]
```

We create an array with randomly selected array indices in the range 0 to *N*. This index array is used to create new arrays which represent our synthetic data. Note that for the copy we used the syntax xr[:] with the colon, because we want to be sure that we are using the same array `xr`, `yr` and `ery` each time, because the fit routine expects the data in these arrays (and not copies of them with the same name). The synthetic data arrays will consist of about 37 percent duplicates. With these synthetic arrays we repeat the fit and find our $p_{(i)}$. If we repeat this many times (let's say 1000), then we get the distribution we needed. The standard deviation of this distribution (i.e. for one parameter), gives the uncertainty.

Note: The bigger the data set, the higher the number of bootstrap trials should be to get accurate statistics. The best way to find a minimum number is to plot the Bootstrap results as in the example.

Jackknife method

Another Monte Carlo method is the Jackknife method. The Jackknife method finds errors on best-fit parameters of a model and *N* data points using *N* samples. In each sample a data point is left out, starting with the first, then the second and so on. For each of these samples we do a fit and store the parameters. For example, for a straight line we store the slopes and offsets. If we concentrate on one parameter and call this parameter θ then for each run *i* we find the estimated slope θ_i. The average of all the slopes is $\bar{\theta^*})$. Then the Jackknife error is:

$$\sigma_{jack} = \sqrt{\frac{N-1}{N}\sum_{i=0}^{N-1}\left(\theta_i - \bar{\theta^*}\right)^2} \tag{2.46}$$

Notes about weighting

Unweighted (i.e. unit weighting) and relative weighted fits

- For unit- or relative weighting, we find errors that correspond to attribute `stderr` in *kmpfit*.
- The errors on the best-fit parameters are scaled (internally) which is equivalent to scaling the weights in a way that the value of the reduced chi-squared becomes 1.0
- For unweighted fits, the standard errors from `Fitter.stderr` are comparable to errors we find with Monte Carlo simulations.

Alper, *[Alp]* states that for some combinations of model, data and weights, *the standard error estimates from diagonal elements of the covariance matrix neglect the interdependencies between parameters and lead to erroneous results.* Often the measurement errors are difficult to obtain precisely, sometimes these errors are not normally distributed. For this category of weighting schemes, one should always inspect the covariance matrix (attribute `covar`) to get an idea how big the covariances are with respect to the variances (diagonal elements of the matrix). The off-diagonal elements of the covariance matrix should be much lower than the diagonal.

Weighted fits with weights derived from real measurement errors

- For weighted fits where the weigths are derived from measurement errors, the errors correspond to attribute `xerror` in *kmpfit*. Only for this type of weights, we get a value of (reduced) chi-squared that can be used as a measure of **goodness of fit**.
- The fit results depend on the accuracy of the measurement errors σ_i.
- A basic assumption of the chi-squared objective function is that the error distribution of the measured data is Gaussian. If this assumption is violated, the value of chi squared does not make sense.
- The uncertainties given in attribute `xerror` and `stderr` are the same, only when $\chi^2_\nu = 1$

>From *[And]* we summarize the conditions which must be met before one can safely use the values in `stderr` (i.e. demanding that $\chi_\nu = 1$): In this approach of scaling the error in the best-fit parameters, we make some assumptions:

1. The error distribution has to be Gaussian.
2. The model has to be linear in all parameters. If the model is nonlinear, we cannot demand that $\chi_\nu = 1$, because the derivation of $\langle\chi\rangle^2 = N - n$ implicitly assumes linearity in all parameters.

3. By demanding $\chi_\nu = 1$, we explicitly claim that the model we are using is the **correct** model that corresponds to the data. This is a rather optimistic claim. This claim requires justification.
4. Even if all these assumptions above are met, the method is in fact only applicable if the degrees of freedom *N-n* is large. The reason is that the uncertainty in the measured data data does not only cause an uncertainty in the model parameters, but also an uncertainty in the value of χ^2 itself. If *N-n* is small, χ^2 may deviate substantially from *N-n* even though the model is linear and correct.

The conclusion is that one should be careful with the use of standard errors in `stderr`. A Monte Carlo method should be applied to prove that the values in `stderr` can be used. For weighted fits it is advertised not to use the Bootstrap method. In the next example we compare the Bootstrap method with and without weights. The example plots all trial results in the Bootstrap procedure. The yellow lines represent weighted fits in the Bootstrap procedure. The green lines represent unweighted fits in the Bootstrap procedure. One can observe that the weighted version shows errors that are much too big.

Example: `kmpfit_weighted_bootstrap.py` **- Compare Bootstrap with weighted and unweighted fits**

```
======== Results kmpfit UNweighted fit =========
Params:         [-0.081129823700123893, 2.9964571786959704]
Errors from covariance matrix         :  [ 0.12223491  0.0044314 ]
Uncertainties assuming reduced Chi^2=1:  [ 0.21734532  0.00787946]
Chi^2 min:      626.001387167
Reduced Chi^2:  3.16162316751
Iterations:     2
Function ev:    7
Status:         1


======== Results kmpfit weighted fit =========
Params:         [-1.3930156818836363, 3.0345053718712571]
Errors from covariance matrix         :  [ 0.01331314  0.0006909 ]
Uncertainties assuming reduced Chi^2=1:  [ 0.10780843  0.00559485]
Chi^2 min:      12984.0423449
Reduced Chi^2:  65.575971439
Iterations:     3
Function ev:    7
Status:         1
Covariance matrix:  [[  1.77239564e-04  -6.78626129e-06]
[ -6.78626129e-06   4.77344773e-07]]


===== Results kmpfit weighted fit with reduced chi^2 forced to 1.0 =====
Params:         [-1.3930155828717012, 3.034505368057717]
Errors from covariance matrix         :  [ 0.10780841  0.00559485]
Uncertainties assuming reduced Chi^2=1:  [ 0.10780841  0.00559485]
Chi^2 min:      198.0
Reduced Chi^2:  1.0
Iterations:     3
Function ev:    7
Status:         1
Bootstrap errors in A, B for procedure with weighted fits: 0.949585141866 0.0273199443168
Bootstrap errors in A, B for procedure with unweighted fits: 0.217752459166 0.00778497229684
```

The same conclusion applies to the Jackknife method. For unweighted fits, the Jackknife error estimates are very good, but for weighted fits, the method can not be used. This can be verified with the example script below. *[Sha]* proposes a modified Jackknife method to improve the error estimates.

Example: `kmpfit_weighted_jackknife.py` **- Compare Jackknife with weighted and unweighted fits**

2.3.5 Goodness of fit

Chi-squared test

As described in a previous section, the value of the reduced chi-squared is an indication for the goodness of fit. If its value is near 1 then your fit is probably good. With the value of chi-squared we can find a threshold value for which we can accept or reject the hypothesis that the data and the fitted model are consistent. The assumption is that the value of chi-squared follows the χ^2 distribution with ν degrees of freedom. Let's examine chi-squared in more detail.

In a chi-squared fit we sum the relative size of the deviation Δ_i and the error bar δ_i. Data points that are near the fit with the best-fit parameters have a small value Δ_i/δ_i. Bad points have a ratio that is bigger than 1. At those points the fitted curve does not go through the error bar. For a reasonable fit, there will be both small and big deviations but on average the value will be near 1. Remember that chi-squared is defined as:

$$\chi^2 = \left(\frac{\Delta_1}{\delta_1}\right)^2 + \left(\frac{\Delta_2}{\delta_2}\right)^2 + \left(\frac{\Delta_3}{\delta_3}\right)^2 + \cdots + \left(\frac{\Delta_N}{\delta_N}\right)^2 \tag{2.47}$$

So if we expect that on average the ratios are 1, then we expect that this sum is equal to *N*. You can always add more parameters to a model. If you have as many parameters as data points, you can find a curve that hits all data points, but usually these curves have no significance. In this case you don't have any *degrees of freedom*. The degrees of freedom for a fit with *N* data points and *n* adjustable model parameters is:

$$\nu = N - n \tag{2.48}$$

To include the degrees of freedom, we define the reduced chi squared as:

$$\chi^2_\nu = \frac{\chi^2}{\nu} \tag{2.49}$$

In the literature (*[Ds3]*) we can find prove that the expectation value of the reduced chi squared is 1. If we repeat a measurement many times, then the measured values of χ^2 are distributed according to the chi-squared distribution with ν degrees of freedom. See for example http://en.wikipedia.org/wiki/Chi-squared_distribution.

We reject the null hypothesis (data is consistent with the model with the best fit parameters) if the value of chi-squared is bigger than some threshold value. The threshold value can be calculated if we set a value of the probability that we make a wrong decision in rejecting a true null hypothesis (H0). This probability is denoted by α and it sets the significance level of the test. Usually we want small values for α like 0.05 or 0.01. For a given value of α we calculate $1-\alpha$, which is the left tail area under the cumulative distribution function. This probability is calculated with `scipy.stats.chi2.cdf()`. If α is given and we want to know the threshold value for chi-squared, then we use the Percent Point Function `scipy.stats.chi2.ppf()` which has $1-\alpha$ as its argument.

The recipe to obtain a threshold value for χ^2 is as follows.

1. **Set the hypotheses:**
    - H_0: The data are consistent with the model with the best fit parameters
    - H_α: The data are *not* consistent with the model with the best fit parameters
2. Make a fit and store the calculated value of χ^2
3. Set a p-value (α)
4. Use the χ^2 cumulative distribution function for ν degrees of freedom to find the threshold χ^2 for $1-\alpha$. Note that α is the right tailed area in this distribution while we use the left tailed area in our calculations.
5. Compare the calculated χ^2 with the threshold value.
6. If the calculated value is bigger, then reject the hypothesis that the data and the model with the best-fit parameters are consistent.

In the next figure we show these steps graphically. Note the use of the statistical functions and methods from SciPy.

Example: kmpfit_goodnessoffit1.py - Goodness of fit based on the value of chi-squared

Kolmogorov-Smirnov test

Another goodness-of-fit test is constructed by using the critical values of the Kolmogorov distribution (Kolmogorov-Smirnov test *[Mas]*).

For this test we need the normalized cumulative versions of the data and the model with the best-fit parameters. We call the cumulative distribution function of the model $F_0(x)$ and the observed cumulative distribution function of our data sample $S_n(x)$ then the sampling distribution of $D = \max|F_0(x) - S_n(x)|$ follows the Kolmogorov distribution which is independent of $F_0(x)$ if $F_0(x)$ is continuous, i.e. has no jumps.

The cumulative distribution of the sample is called the *empirical distribution function* (ECDF). To create the ECDF we need to order the sample values $y_0, y_1, ..., y_n$ from small to high values. Then the ECDF is defined as:

$$S_N = \frac{n(i)}{N} \tag{2.50}$$

The value of $n(i)$ is the number of sample values y that are smaller than or equal to y_i. So the first value would be *1/N*, the second *2/N* etc.

The cumulative distribution function (CDF) of the model can be calculated in the same way. First we find the best-fit parameters for a model using *kmpfit*. Select a number of *X* values to find *Y* values of your model. Usually the number of model samples is much higher than the number of data samples. With these (model) *Y* values we create a CDF using the criteria (ordered *Y* values) of the data. If *dat1* are the ordered sample *Y* values and *dat2* are the ordered model *Y* values, then a function that calculates the CDF could be:

```
def cdf(Y_ord_data, Y_ord_model):
   cdfnew = []
   n = len(Y_ord_model)
   for yy in Y_ord_data:
      fr = len(Y_ord_model[Y_ord_model <= yy])/float(n)
      cdfnew.append(fr)
   return numpy.asarray(cdfnew)
```

which is not the most efficient procedure but it is simple and it just works.

For hypotheses testing we define:

- H_0: The data are consistent with the model with the best fit parameters
- H_α: The data are *not* consistent with the model with the best fit parameters

Note that the ECDF is a step function and this step function could be interpreted in two ways. Therefore the Kolmogorov-Smirnov (KS) test statistic is defined as:

$$D_n = \max_{0\leq i\leq N-1} \left(\frac{i+1}{N} - F_0(y_i),\, F_0(y_i) - \frac{i}{N}\right) \tag{2.51}$$

where we note that F_0 is a continuous distribution function (a requirement for the KS-test).

The null hypothesis is rejected at a critical probability α (confidence level) if $D_n > D_\alpha$. The value D_α is a threshold value. Given the value of α, we need to find D_α by solving:

$$Pr(D_n < D_\alpha) = 1 - \alpha \tag{2.52}$$

To find this probability we use the Kolmogorov-Smirnov **two**-sided test which can be approximated with SciPy's method `scipy.stats.kstwobign()`. This test uses $D_n/\sqrt(N)$ as input and the output of `kstwobign.ppf()` is $D_n * \sqrt(N)$. Given a value for *N*, we find threshold values for D_n for frequently used values of confidence level α, as follows:

```
N = ...
from scipy.stats import kstwobign
# Good approximation for the exact distribution if N>4
dist = kstwobign()
alphas = [0.2, 0.1, 0.05, 0.025, 0.01]
for a in alphas:
   Dn_crit = dist.ppf(1-a)/numpy.sqrt(N)
   print "Critical value of D at alpha=%.3f(two sided):  %g"%(a, Dn_crit)
```

In the next script we demonstrate that the Kolmogorov-Smirnov test is useful if we have reasonable fits, but bad values of chi-squared due to improperly scaled errors on the data points. The χ^2 test will immediately reject the hypothesis that data and model are consistent. The Kolmogorov-Smirnov test depends on the difference between the cumulative distributions and does not depend on the scale of these errors. The empirical and model cdf's show where the fit deviates most from the model. A plot with these cdf's can be a starting point to reconsider a model if the deviations are too large.

Example: kmpfit_goodnessoffit2.py - Kolmogorov-Smirnov goodness of fit test

2.3.6 Profile fitting

Gaussian profiles

There are many examples where an astronomer needs to know the characteristics of a Gaussian profile. Fitting best parameters for a model that represents a Gauss function, is a way to obtain a measure for the peak value, the position of the peak and the width of the peak. It does not reveal any skewness or kurtosis of the profile, but often these are not important. We write the Gauss function as:

$$f(x) = Ae^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} + z_0 \tag{2.53}$$

Here A represents the peak of the Gauss, μ the mean, i.e. the position of the peak and σ the width of the peak. We added z_0 to add a background to the profile characteristics. In the early days of fitting software, there were no implementations that did not need partial derivatives to find the best fit parameters.

Partial derivatives for a Gaussian

In the documentation of the IDL version of *mpfit.pro*, the author states that it is often sufficient and even faster to allow the fit routine to calculate the derivatives numerically. In contrast with this we usually gain an increase in speed of about 20% if we use explicit partial derivatives, at least for fitting Gaussian profiles. The real danger in using explicit partial derivatives seems to be that one easily makes small mistakes in deriving the necessary equations. This is not always obvious in test-runs, but *kmpfit* is capable of providing diagnostics. For the Gauss function in (2.53) we derived the following partial derivatives:

$$\begin{aligned}
\frac{\partial f(x)}{\partial A} &= e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\\
\frac{\partial f(x)}{\partial \mu} &= Ae^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.\frac{(x-\mu)}{\sigma^2}\\
\frac{\partial f(x)}{\partial \sigma} &= Ae^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.\frac{(x-\mu)^2}{\sigma^3}\\
\frac{\partial f(x)}{\partial z_0} &= 1
\end{aligned} \tag{2.54}$$

If we want to use explicit partial derivatives in *kmpfit* we need the external residuals to return the derivative of the model *f(x)* at *x*, with respect to any of the parameters. If we denote a parameter from the set of parameters $P = (A, \mu, \sigma, z_0)$ with index i, then one calculates the derivative with a function `FGRAD(P,x,i)`. In fact, kmpfit needs the derivative of the **residuals** and if we defined the residuals as `residuals = (data-model)/err`, the residuals function should return:

$$\frac{\partial f(x)}{\partial P(i)} = \frac{-FGRAD(P,x,i)}{err} \tag{2.55}$$

where *err* is the array with weights.

Below, we show a code example of how one can implement explicit partial derivatives. We created a function, called `my_derivs` which calculates the derivatives for each parameter. We tried to make the code efficient but you should be able to recognize the equations from (2.54). The return value is equivalent with (2.55). The function has a fixed signature because it is called by the fitter which expects that the arguments are in the right order. This order is:

- *p* -List with model parameters, generated by the fit routine
- *data* -A reference to the `data` argument in the constructor of the Fitter object.
- *dflags* -List with booleans. One boolean for each model parameter. If the value is `True` then an explicit partial derivative is required. The list is generated by the fit routine.

There is no need to process the `dflags` list in your code. There is no problem if you return all the derivatives even when they are not necessary.

Note: A function which returns derivatives should create its own work array to store the calculated values. The shape of the array should be (parameter_array.size, x_data_array.size).

The function `my_derivs` is then:

```
def my_derivs(p, data, dflags):
   #-----------------------------------------------------------------------
   # This function is used by the fit routine to find the values for
   # the explicit partial derivatives. Argument 'dflags' is an array
   # with booleans. If an element is True then an explicit partial
   # derivative is required.
   #-----------------------------------------------------------------------
   x, y, err = data
   A, mu, sigma, zerolev = p
   pderiv = numpy.zeros([len(p), len(x)])  # You need to create the required array
   sig2 = sigma*sigma
   sig3 = sig2 * sigma
   xmu  = x-mu
   xmu2 = xmu**2
   expo = numpy.exp(-xmu2/(2.0*sig2))
   fx = A * expo
   for i, flag in enumerate(dflags):
      if flag:
         if i == 0:
            pderiv[0] = expo
         elif i == 1:
            pderiv[1] = fx * xmu/(sig2)
         elif i == 2:
            pderiv[2] = fx * xmu2/(sig3)
         elif i == 3:
            pderiv[3] = 1.0
   return pderiv/-err
```

Note that all the values per parameter are stored in a row. A minus sign is added to to the error array to fulfill the requirement in equation (2.55). The constructor of the Fitter object is as follows (the function `my_residuals` is not given here):

```
fitobj = kmpfit.Fitter(residuals=my_residuals, deriv=my_derivs, data=(x, y, err))
```

The next code and plot show an example of finding and plotting best fit parameters given a Gauss function as model. If you want to compare the speed between a fit with explicit partial derivatives and a fit using numerical derivatives, add a second Fitter object by omitting the `deriv` argument. In our experience, the code with the explicit partial derivatives is about 20% faster because it needs considerably fewer function calls to the residual function.

Example: kmpfit_example_partialdervs.py - Finding best fit parameters for a Gaussian model

```python
#!/usr/bin/env python
#------------------------------------------------------------
# Script compares efficiency of automatic derivatives vs
# analytical in mpfit.py
# Vog, 31 okt 2011
#------------------------------------------------------------

import numpy
from matplotlib.pyplot import figure, show, rc
from kapteyn import kmpfit

def my_model(p, x):
   #-----------------------------------------------------------------------
   # This describes the model and its parameters for which we want to find
   # the best fit. 'p' is a sequence of parameters (array/list/tuple).
   #-----------------------------------------------------------------------
   A, mu, sigma, zerolev = p
   return( A * numpy.exp(-(x-mu)*(x-mu)/(2.0*sigma*sigma)) + zerolev )


def my_residuals(p, data):
   #-----------------------------------------------------------------------
   # This function is the function called by the fit routine in kmpfit
   # It returns a weighted residual. De fit routine calculates the
   # square of these values.
   #-----------------------------------------------------------------------
   x, y, err = data
   return (y-my_model(p,x)) / err


def my_derivs(p, data, dflags):
   #-----------------------------------------------------------------------
   # This function is used by the fit routine to find the values for
   # the explicit partial derivatives. Argument 'dflags' is a list
   # with booleans. If an element is True then an explicit partial
   # derivative is required.
   #-----------------------------------------------------------------------
   x, y, err = data
   A, mu, sigma, zerolev = p
   pderiv = numpy.zeros([len(p), len(x)])  # You need to create the required array
   sig2 = sigma*sigma
   sig3 = sig2 * sigma
   xmu  = x-mu
   xmu2 = xmu**2
   expo = numpy.exp(-xmu2/(2.0*sig2))
   fx = A * expo
   for i, flag in enumerate(dflags):
      if flag:
         if i == 0:
            pderiv[0] = expo
         elif i == 1:
            pderiv[1] = fx * xmu/(sig2)
         elif i == 2:
            pderiv[2] = fx * xmu2/(sig3)
         elif i == 3:
            pderiv[3] = 1.0
   pderiv /= -err
   return pderiv


# Artificial data
N = 100
x = numpy.linspace(-5, 10, N)
truepars = [10.0, 5.0, 1.0, 0.0]
p0 = [9, 4.5, 0.8, 0]
y = my_model(truepars, x) + 0.3*numpy.random.randn(len(x))
err = 0.3*numpy.random.randn(N)

# The fit
fitobj = kmpfit.Fitter(residuals=my_residuals, deriv=my_derivs, data=(x, y, err))
try:
   fitobj.fit(params0=p0)
except Exception, mes:
   print "Something wrong with fit: ", mes
   raise SystemExit

print "\n\n======== Results kmpfit with explicit partial derivatives ========="
print "Params:        ", fitobj.params
print "Errors from covariance matrix         : ", fitobj.xerror
print "Uncertainties assuming reduced Chi^2=1: ", fitobj.stderr
print "Chi^2 min:     ", fitobj.chi2_min
print "Reduced Chi^2: ", fitobj.rchi2_min
print "Iterations:    ", fitobj.niter
print "Function ev:   ", fitobj.nfev
print "Status:        ", fitobj.status
print "Status Message:", fitobj.message
print "Covariance:\n", fitobj.covar

# Plot the result
rc('font', size=9)
rc('legend', fontsize=8)
fig = figure()
frame = fig.add_subplot(1,1,1)
frame.errorbar(x, y, yerr=err, fmt='go', alpha=0.7, label="Noisy data")
frame.plot(x, my_model(truepars,x), 'r', label="True data")
frame.plot(x, my_model(fitobj.params,x), 'b', lw=2, label="Fit with kmpfit")
frame.set_xlabel("X")
frame.set_ylabel("Measurement data")
frame.set_title("Least-squares fit to noisy Gaussian data using KMPFIT",
                fontsize=10)
leg = frame.legend(loc=2)
show()
```

Automatic initial estimates for profiles with multi component Gaussians

For single profiles we can obtain reasonable initial estimates by inspection of the profile. Processing many profiles, e.g. in a data cube with two spatial axes and one spectral axis, needs another approach. If your profile has more than 1

Gaussian component, the problem becomes even more complicated. So what we need is a method that automates the search for reasonable initial estimates.

Gauest

Function `profiles.gauest()` is a function which can be used to get basic characteristics of a Gaussian profile. The number of Gaussian components in that profile can be greater than 1. These characteristics are *amplitude*, *position of the maximum* and *dispersion*. They are very useful as initial estimates for a least squares fit of this type of multi-component Gausian profiles. For `gauest()`, the profile is represented by intensities y_i, expressed as a function of the independent variable x at equal intervals $\Delta x = h$ *[Sch]*. A second order polynomial is fitted at each x_i by using moments analysis (this differs from the method described in *[Sch]*), using q points distributed symmetrically around x_i, so that the total number of points in the fit is $2q + 1$. The coefficient of the second-order term is an approximation of the second derivative of the profile. For a Gaussian model, the position of the peak and the dispersion are calculated from the main minima of the second derivative. The amplitude is derived from the profile intensities. The function has parameters to set thresholds in minimum amplitude and dispersion to discriminate against spurious components.

Thresholds

Function `gauest()` uses an automatic window method to find the signal region of a profile. If the maximum of the entire profile is below the (user) given cutoff in amplitude (*cutamp*), then no signal is found and the process of finding Gaussian components is aborted. Otherwise, the position of the maximum is selected as the center of the first component and from this point on, a region is increased until the difference between the total flux and the flux in the region is smaller than or equal to the value of parameter *rms*, the noise in the profile. Then the method in *[Sch]* is used to find the characteristics of the Gaussian. This method is based on fitting (using moments analysis) of a second-order polynomial. The distance between the maxima of this polynomial is a measure for the width of the peak. If this width is greater than the threshold value given by the user in parameter *cutsig*, then there is a second check using the amplitude threshold (*cutamp*) given by the user. The reason for this is that the amplitude is also derived from moment analysis and can give a result that is greater than the maximum value in the profile. If both tests are passed then the Gaussian is stored as a valid component. This component is subtracted from the profile and the procedure is repeated until *ncomp* components are found or a signal region could not be found anymore.

Smoothing factor

The parameter q is a bit tricky. If q is big (e.g. 20) then the routine is less effective as with for example 5. But if q is too small, you don't always find the number of required components. Therefore it is important to find an optimum. In the script below we apply an iteration, starting with a reasonable value of q and increasing it until we found the required number of components or until q becomes too big. Parameter q is also called the *smoothing parameter*. If you take more points in the moments analysis of the polynomial, the effect will be that you apply smoothing of the data which gives better results if you have noisy data.

Note: Function `gauest()` requires parameters of which the optimal values depend on the profile data. You need to estimate the noise (*rms*) in the profile, a critical amplitude (*cutamp*) and dispersion (*cutdisp*). Also the smoothing factor q has an optimal value that depends on the profile data. Usually it is not difficult to obtain reasonable values for all these parameters.

Example: `kmpfit_gauest_multicomp.py` **- Function gauest() finds initial estimates in profiles with multi component Gaussians**

Messy samples

The original C version of function `gauest()` works with the assumption that your x values run from 0 .. N where N is the number of data points. Many profiles have different x values. Sometimes they are not sorted and sometimes they are not equally spaced. The current function `gauest()` inspects the data in argument *x*. If necessary, it sorts the data and forces it to be equally spaced by linear interpolation. This could be dangerous if your samples are distributed in a messy way, but usually `gauest()` will be able to find reasonable estimates. The procedure which modifies the data to make it usable for `gauest()` is based on the code in the next example.

Example: `kmpfit_gauest_prepare.py` **- Demonstrate how profile data needs to be prepared for gauest()**

Fitting Voigt profiles

The line-shapes of spectroscopic transitions depend on the broadening mechanisms of the initial and final states, and include natural broadening, collisional broadening, power broadening, and Doppler broadening. Natural, collisional, and power broadening are homogeneous mechanisms and produce Lorentzian line-shapes. Doppler broadening is a form of inhomogeneous broadening and has a Gaussian line-shape. Combinations of Lorentzian and Gaussian line-shapes can be approximated by a Voigt profile. In fact, the Voigt profile is a convolution of Lorentzian and Doppler line broadening mechanisms:

$$\phi_{Lorentz}(\nu) = \frac{1}{\pi}\frac{\alpha_L}{(\nu-\nu_0)^2+\alpha_L^2} \tag{2.56}$$

$$\phi_{Doppler}(\nu) = \frac{1}{\alpha_D}\sqrt{\frac{\ln 2}{\pi}}e^{-\ln 2\frac{(\nu-\nu_0)^2}{\alpha_D^2}} \tag{2.57}$$

Both functions are normalized, α_D and α_L are **half** widths at **half** maximum *[Scr]*. Convolution is given by the relation:

$$f(\nu)\star g(\nu) = \int\limits_{-\infty}^{\infty} f(\nu-t)g(t)dt \tag{2.58}$$

Define the ratio of Lorentz to Doppler widths as:

$$y \equiv \frac{\alpha_L}{\alpha_D}\sqrt{\ln 2} \tag{2.59}$$

and the frequency scale (in units of the Doppler Line-shape half-width α_D):

$$x \equiv \frac{\nu-\nu_0}{\alpha_D}\sqrt{\ln 2} \tag{2.60}$$

The convolution of both functions is:

$$\phi_\nu(\nu) = \phi_L(\nu)\star\phi_D(\nu) = \frac{1}{\alpha_D}\sqrt{\frac{\ln 2}{\pi}}\frac{y}{\pi}\int\limits_{-\infty}^{\infty}\frac{e^{-t^2}}{(x-t)^2+y^2}dt \tag{2.61}$$

Part of the expression of the Voigt line-shape is the Voigt function $K(x,y)$. The definition of this function is:

$$K(x,y) = \frac{y}{\pi}\int\limits_{-\infty}^{\infty}\frac{e^{-t^2}}{y^2+(x-t)^2}dt \tag{2.62}$$

Then:

$$\phi_\nu(\nu) = \frac{1}{\alpha_D}\sqrt{\frac{\ln 2}{\pi}}K(x,y) \tag{2.63}$$

Using the expressions for *x* and *y* from (2.60) and (2.59), this can be rewritten in terms of the physical parameters as *[Vog]*:

$$\phi_\nu(\nu) = \frac{\alpha_L}{\alpha_D^2}\,\frac{\ln 2}{\pi^{\frac{3}{2}}} \int\limits_{-\infty}^{\infty} \frac{e^{-t^2}}{\left(\frac{\nu-\nu_0}{\alpha_D}\sqrt{\ln 2}-t\right)^2 + \left(\frac{\alpha_L}{\alpha_D}\sqrt{\ln 2}\right)^2} dt \tag{2.64}$$

Note that α_L and α_D are both **half-width at half maximum** and not FWHM's. In *[Vog]*, it is proved that:

$$\int\limits_{-\infty}^{\infty} \phi_\nu(\nu) d\nu = 1 \tag{2.65}$$

so the Voigt line-shape (eq. (2.61)) is also normalized. When we want to find the best-fit parameters of the Voigt line-shape model, we need to be able to process profiles with arbitrary area and we need a scaling factor *A*. The expression for the Voigt line-shape becomes:

$$\boxed{\phi_\nu(\nu) = A\,\frac{1}{\alpha_D}\sqrt{\frac{\ln 2}{\pi}}\,K(x,y)} \tag{2.66}$$

One can prove [*Vog*] with the substitution of:

$$\boxed{z = x + iy} \tag{2.67}$$

that the Voigt function can be expressed as the real part of a special function:

$$\boxed{K(x,y) = \Re\{\omega(z)\}} \tag{2.68}$$

$\omega(z)$ is called the complex probability function, also known as the Faddeeva function. Scipy has implemented this function under the name `scipy.special.wofz()`.

The amplitude is found at $\nu = \nu_0$. Then the relation between amplitude and area is $amp = \phi(\nu_0)$:

$$\boxed{amp = \phi(\nu_0) = \frac{A}{\alpha_D}\sqrt{\frac{\ln 2}{\pi}}K(0,y)} \tag{2.69}$$

In *[Scr]* we read that the half width at half maximum can be found with:

$$\boxed{hwhm = \frac{1}{2}\left(c_1\,\alpha_L + \sqrt{c_2\,\alpha_L^2 + 4\,\alpha_D^2}\right)} \tag{2.70}$$

with $c_1 = 1.0692$ and $c_2 = 0.86639$.

The Voigt function can be implemented using SciPy's function `wofz()`. In the next code fragments, it should be easy to find correspondence between code and boxed formulas:

```python
def voigt(x, y):
   # The Voigt function is also the real part of
   # w(z) = exp(-z^2) erfc(iz), the complex probability function,
   # which is also known as the Faddeeva function. Scipy has
   # implemented this function under the name wofz()
   z = x + 1j*y
   I = wofz(z).real
   return I

```

```

def Voigt(nu, alphaD, alphaL, nu_0, A, a=0, b=0):
   # The Voigt line shape in terms of its physical parameters
   f = numpy.sqrt(ln2)
   x = (nu-nu_0)/alphaD * f
   y = alphaL/alphaD * f
   backg = a + b*nu
   V = A*f/(alphaD*numpy.sqrt(numpy.pi)) * voigt(x, y) + backg
   return V

# Half width and amplitude
c1 = 1.0692
c2 = 0.86639
hwhm = 0.5*(c1*alphaL+numpy.sqrt(c2*alphaL**2+4*alphaD**2))
f = numpy.sqrt(ln2)
y = alphaL/alphaD * f
amp = A/alphaD*numpy.sqrt(ln2/numpy.pi)*voigt(0,y)
```

with:

- nu: x-values, usually frequencies.
- alphaD: Half width at half maximum for Doppler profile
- alphaL: Half width at half maximum for Lorentz profile
- nu_0: Central frequency
- A: Area under profile
- a, b: Background as in a + b*x

In the example below, we compare a Gaussian model with a Voigt model. We had some knowledge about the properties of the profile data so finding appropriate initial estimates is not difficult. If you need to automate the process of finding initial estimates, you can use function `gauest()` (*Gauest*) from the section about initial estimates. However, note that you need to invert the data because `gauest()` can only process peaks (positive amplitudes).

Example: kmpfit_voigt.py - The Voigt line shape

Fitting Gauss-Hermite series

If your profile deviates from a Gaussian shape (e.g. asymmetric profiles) then you can use the so called {it Gauss-Hermite} series. The series are used to derive skewness and kurtosis of your data distribution. The lowest order term of the series is a Gaussian. The higher order terms are orthogonal to this Gaussian. The higher order that we use in our fits are the parameters h_3 and h_4 measuring asymmetric and symmetric deviations of a Gaussian. The Gauss-Hermite function and its applications are described in *[Mar]*, but we use the (equivalent) formulas from *[Vog]*

$$\phi(x) = A\, e^{-\frac{1}{2}y^2} \left\{1 + \frac{h_3}{\sqrt{6}}\,(2\sqrt{2}y^3 - 3\sqrt{2}y) + \frac{h_4}{\sqrt{24}}(4y^4 - 12y^2 + 3)\right\} + Z \tag{2.71}$$

with: $y \equiv \frac{x-\mu_g}{\sigma_g}$.

Simplify this equation further:

$$\phi(x) = A\,E\,\left\{1 + h_3(c_1 y + c_3 y^3) + h_4(c_0 + c_2 y^2 + c_4 y^4)\right\} \tag{2.72}$$

or:

$$\phi(x) = A\,E\,Q \tag{2.73}$$

with $E \equiv e^{-\frac{1}{2}y^2}$ and $Q = \left\{1 + h_3(c_1 y + c_3 y^3) + h_4(c_0 + c_2 y^2 + c_4 y^4)\right\}$ and its coefficients:

$$\begin{aligned} c_0 &= \frac{1}{4}\sqrt{6} \\ c_1 &= -\sqrt{3} \\ c_2 &= -\sqrt{6} \\ c_3 &= \frac{2}{3}\sqrt{3} \\ c_4 &= \frac{1}{3}\sqrt{6} \end{aligned} \tag{2.74}$$

To find the real maximum (which is not the maximum of the Gaussian part of the expression), solve:

$$\frac{\partial \phi(x)}{\partial x} = -a\,E\,\frac{1}{c}\left[h_3(-c_1 - 3c_3 y^2) + h_4(-2c_2 y - 4c_4 y^3) + y\,Q\right] = 0 \tag{2.75}$$

We used SciPy's function `fsolve()` in the neighbourhood of 0 to find the solution of this expression.

Moments of the GH series *[Vog]*

The integrated line strength γ:

$$\boxed{\gamma_{gh} = A\,\sigma_g\,\sqrt{2\pi}(1 + \frac{1}{4}\sqrt{6}\,h_4) = \gamma_g\,(1 + \frac{1}{4}\sqrt{6}\,h_4)} \tag{2.76}$$

The mean abscissa μ_{gh}:

$$\boxed{\mu_{gh} \approx \mu_g + \sqrt{3}\,h_3\,\sigma_g} \tag{2.77}$$

The dispersion σ_{gh}:

$$\boxed{\sigma_{gh} \approx \sigma_g\,(1 + \sqrt{6}\,h_4)} \tag{2.78}$$

The Fisher coefficient of Skewness ξ_1:

A set of observations that is not symmetrically distributed is said to be skewed. If the distribution has a longer tail less than the maximum, the function has *negative skewness*. Otherwise, it has *positive skewness*.

$$\boxed{\xi_1 \approx 4\sqrt{3}\,h_3} \tag{2.79}$$

This is what we could have expected because h_3 is the parameter that measures asymmetric deviations.

The Fisher coefficient of Kurtosis ξ_2:

This parameter measures both the *peakedness* of the distribution and the heaviness of its tail:

$$\boxed{\xi_2 \approx 3 + 8\sqrt{6}\,h_4} \tag{2.80}$$

Or use the definition of excess kurtosis ξ_f:

$$\boxed{\xi_f = \xi_2 - 3 \approx 8\sqrt{6}\,h_4} \tag{2.81}$$

A negative value means that distribution is flatter then a pure Gaussian. and if it is positive then the distribution is sharper then a pure Gaussian. A Gaussian distribution has zero excess kurtosis.

It is obvious that for $h_3 = 0$ and $h_4 = 0$, all these parameters are the same as their Gaussian counterparts. A line-shape model based on the Gauss-Hermite series will resemble a pure Gaussian. Therefore it is save to set the initial guesses for the h_3 and h_4 parameters in the least-squares fit to zero because. If a fit is successful, the profile parameters γ_{gh},

μ_{gh} and σ_{gh}, skewness and kurtosis are calculated from the best fit parameters A, μ_g, σ_g, h_3 and h_4 using the formulas above. For the errors in these parameters we derived:

$$\begin{aligned}
\Delta\gamma_{gh} &= \frac{1}{\gamma_{gh}}\sqrt{\left(\frac{\Delta A}{A}\right)^2 + \left(\frac{\Delta\sigma_g}{\sigma_g}\right)^2 + \left(\frac{1}{\frac{2}{3}\sqrt{6}+h_4}\right)^2\left(\frac{\Delta h_4}{h_4}\right)^2} \\
\Delta\mu_{gh} &= \sqrt{(\Delta\mu_g)^2 + 3h_3^2(\Delta\sigma_g)^2 + 3\sigma_g^2(\Delta h_3)^2} \\
\Delta\sigma_{gh} &= \sqrt{(1+\sqrt{6}\,h_4)^2\,(\Delta\sigma_g)^2 + 6\sigma_g^2(\Delta h_4)^2} \\
\Delta\xi_1 &= 4\sqrt{3}\,\Delta h_3 \\
\Delta\xi_2 &= 8\sqrt{6}\,\Delta h_4
\end{aligned} \tag{2.82}$$

These formulas are used in the next example. It is a script that finds best-fit parameters of a Gaussian, Voigt and Gauss-Hermite model. Only the last model can quantify the asymmetry of the data. The data is derived from the GH-series and some noise is added. The Voigt line-shape has a problem with asymmetric data. It tends to find negative values for one of the half widths (α_D or α_L). To avoid this we use the *limits* option in *kmpfit*'s `parinfo` dictionary as follows:

```
>>> fitter.parinfo = [{'limits':(0,None)}, {'limits':(0,None)}, {}, {}, {}]
```

Example: kmpfit_gausshermite.py - The Gauss-Hermite series compared to Voigt and Gauss

2.3.7 Fitting data when both variables have uncertainties

Sometimes your data contains errors in both the *response* (dependent) variable y (i.e. we have values for σ_y) and in the *explanatory* (independent) variable x (i.e. we have values for σ_x). In the next sections we describe a method to use *kmpfit* for this category of least squares fit problems.

Orthogonal Distance Regression (ODR)

Assume we have a model function *f(x)* and on that curve we have a data point $(\hat{x}, \hat{y}) = (\hat{x}, f(\hat{x}))$ which has the shortest distance to a data point (x_i, y_i). The distance between those points is:

$$D_i(\hat{x}) = \sqrt{(x_i-\hat{x})^2 + (y_i - f(\hat{x}))^2} \tag{2.83}$$

or more general with weights in $\hat{x}, \hat{y}$

$$D_i(\hat{x}) = \sqrt{w_{xi}(x_i-\hat{x})^2 + w_{yi}(y_i - f(\hat{x}))^2} \tag{2.84}$$

The problem with this distance function is that it is not usable as an *Objective Function* because we don't have the model values for $\hat{x}$. But there is a condition that can be used to express $\hat{x}$ in known variables x_i and y_i Orear *[Ore]* showed that for any model *f(x)* for which

$$f(\hat{x}) = f(x_i) + (\hat{x} - x_i)f'(x_i) \tag{2.85}$$

is a good approximation, we can find an expression for a usable objective function. $D_i(\hat{x})$ has a minimum for $\frac{\partial D}{\partial \hat{x}} = 0$. Insert (2.85) in (2.84) and take the derivative to find the condition for the minimum:

$$\frac{\partial D}{\partial \hat{x}} = \frac{\partial}{\partial \hat{x}}\sqrt{w_{xi}(x_i-\hat{x})^2 + w_{yi}(y_i - [f(x_i) + (\hat{x}-x_i)f'(x_i)])^2} = 0 \tag{2.86}$$

Then one derives:

$$-2w_x(x_i-\hat{x}) - 2w_y\left(y_i - [f(x_i) - (x_i-\hat{x})f'(x_i)]\right)f'(x_i) = 0 \tag{2.87}$$

so that:

$$(x_i - \hat{x}) = \frac{-w_y\big(y_i - f(x_i)\big)f'(x_i)}{w_x + w_y f'^2(x_i)} \tag{2.88}$$

If you substitute this in (2.84), then (after a lot of re-arranging) one finds for the objective function:

$$D_i^2(\hat{x}) \approx \frac{w_{xi}w_{yi}}{w_{xi} + w_{yi}f'^2(x_i)}(y_i - f(x_i))^2 \tag{2.89}$$

If we use statistical weighting with weights $w_{xi} = 1/{\sigma_{xi}}^2$ and $w_{yi} = 1/{\sigma_{yi}}^2$, we can write this as:

$$\boxed{\chi^2 = \sum_{i=0}^{N-1} D_i^2 = \frac{\big(y_i - f(x_i)\big)^2}{\sigma_{yi}^2 + \sigma_{xi}^2 f'^2(x_i)}} \tag{2.90}$$

Effective variance

The method in the previous section can also be explained in another way: Clutton *[Clu]* shows that for a model function *f*, the effect of a small error δx_i in x_i is to change the measured value y_i by an amount $f'(x_i)\delta x_i$ and that as a result, the *effective variance* of a data point *i* is:

$$var(i) = var(y_i) + var(f'(x_i)) = \sigma_{y_i}^2 + f'^2(x_i)\sigma_{x_i}^2 \tag{2.91}$$

Best parameters for a straight line

Equation (2.90) can be used to create an objective function. We show this for a model which represents a straight line $f(x) = a + bx$. For a straight line the Taylor approximation (2.85) is exact. This can be seen as follows: With $f'(x) = b$. The relation $f(x) = f(x_i) + (x - x_i)f'(x_i)$ is equal to $f(x) = f(x_i) + (x - x_i)b = a + bx_i + bx - bx_i = a + bx$.

The objective function, chi-square, that needs to be minimized for a straight line is then:

$$\chi^2 = \sum_{i=0}^{N-1} D_i^2 = \sum_{i=0}^{N-1} \frac{(y_i - a - bx_i)^2}{\sigma_{y_i}^2 + \sigma_{x_i}^2 b^2} \tag{2.92}$$

This formula seems familiar. It resembles an ordinary least squares objective function but with 'corrected' weights in Y. A suitable residuals function for *kmpfit* is the square root of this objective function:

```
def residuals(p, data):
   a, b = p
   x, y, ex, ey = data
   w = ey*ey + b*b*ex*ex
   wi = numpy.sqrt(numpy.where(w==0.0, 0.0, 1.0/(w)))
   d = wi*(y-model(p,x))
   return d
```

Pearson's data

Another approach to find the best fit parameters for orthogonal fits of straight lines starts with the observation that best (unweighted) fitting straight lines for given data points go through the centroid of the system. This applies to regression of y on x, regression of x on y and also for the result of an orthogonal fit.

Note: Unweighted best fitting straight lines for given data points go through the centroid of the system.

If we express our straight line as $y = b + \tan(\theta)x$ and substitute the coordinates of the centroid $(\bar{x}, \bar{y})$, we get the expression for a straight line:

$$\tan(\theta)x - y + \bar{y} - \tan(\theta)\bar{x} = 0 \tag{2.93}$$

For a line $ax + by + c = 0$ we know that the distance of a data point (x_i, y_i) to this line is given by: $(ax_i + by_i + c)/\sqrt{(a^2 + b^2)}$. If we use this for (2.93) then we derive an expression for the distance *D*:

$$D_i = [\tan(\theta)x_i - y_i + \bar{y} - \tan(\theta)\bar{x}] \cos(\theta) \tag{2.94}$$

For an objective function we need to minimize:

$$\sum_{i=0}^{N-1} D_i^2 = \sum_{i=0}^{N-1} [\tan(\theta)x_i - y_i + \bar{y} - \tan(\theta)\bar{x}]^2 \cos(\theta)^2 \tag{2.95}$$

To minimize this we set the first partial derivative with respect to θ to 0 and find the condition:

$$\tan(2\theta) = \frac{\sum\limits_{i=0}^{N-1} (y_i - \bar{x})(y_i - \bar{x})}{\sum\limits_{i=0}^{N-1} (y_i - \bar{y})^2 - \sum\limits_{i=0}^{N-1} (x_i - \bar{x})^2} \tag{2.96}$$

Fitting problems like the ones we just described are not new. In 1901, Karl Pearson published an article *[Pea]* in which he discussed a problem "where the *Independent Variable* is subject to as much deviation or error as the *Dependent Variable*. He derived the same best-fit angle (2.96) in a different way (using correlation ellipsoids). Pearson writes it as:

$$\tan(2\theta) = \frac{2r_{xy}\sigma_x\sigma_y}{\sigma_x^2 - \sigma_y^2} \tag{2.97}$$

where r_{xy} is called the Pearson product-moment correlation coefficient. Using the same variables he writes for the slope b_1 of a regression of y on x and the slope b_2 for a regression of x on y:

$$b_1 = \frac{r_{xy}\sigma_y}{\sigma_x}, \quad b_2 = \frac{r_{xy}\sigma_x}{\sigma_y} \tag{2.98}$$

with:

$$r_{xy} = \frac{\sum\limits_{i=0}^{N-1} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum\limits_{i=0}^{N-1} (x_i - \bar{x})^2}\sqrt{\sum\limits_{i=0}^{N-1} (y_i - \bar{y})^2}} \tag{2.99}$$

With (2.97) and (2.98) we get the well-known relation between the slopes of the two regression lines and the correlation coefficient:

$$r_{xy}^2 = b_1 * b_2 \tag{2.100}$$

and (2.97) can be written as:

$$\boxed{\tan(2\theta) = \frac{2b_1b_2}{b_2 - b_1}} \tag{2.101}$$

On page 571 in this article he presented a table with data points. This table has been used many times in the literature to compare different methods.

```
>>> x = numpy.array([0.0, 0.9, 1.8, 2.6, 3.3, 4.4, 5.2, 6.1, 6.5, 7.4])
>>> y = numpy.array([5.9, 5.4, 4.4, 4.6, 3.5, 3.7, 2.8, 2.8, 2.4, 1.5])
```

So let's prove it works with a short program. The script in the next example calculates Pearson's best fit slope using the analytical formulas from this section. Then it shows how one can use *kmpfit* for a regression of y on x and for a regression of x on y. In the latter case, we swap the data arrays x and y in the initialization of *kmpfit*. Note that for a plot we need to transform its offset and slope in the YX plane to an offset and slope in the XY plane. If the values are (a, b) in the YX plane, then in the XY plane, the offset and slope will be $(-a/b, 1/b)$.

```
#!/usr/bin/env python

import numpy
from kapteyn import kmpfit

def model(p, x):
   # Model: y = a + numpy.tan(theta)*x
   a, theta = p
   return a + numpy.tan(theta)*x

def residuals(p, data):
   # Residuals function for data with errors in y only
   a, b = p
   x, y = data
   d = (y-model(p,x))
   return d

# Pearsons data
x = numpy.array([0.0, 0.9, 1.8, 2.6, 3.3, 4.4, 5.2, 6.1, 6.5, 7.4])
y = numpy.array([5.9, 5.4, 4.4, 4.6, 3.5, 3.7, 2.8, 2.8, 2.4, 1.5])
N = len(x)

fitobj2 = kmpfit.Fitter(residuals=residuals, data=(x, y))
fitobj2.fit(params0=[5,0])
a1, b1 = fitobj2.params[0], numpy.tan(fitobj2.params[1])
fitobj3 = kmpfit.Fitter(residuals=residuals, data=(y, x))
fitobj3.fit(params0=(0,5))
a2, b2  = fitobj3.params[0], numpy.tan(fitobj3.params[1])
tan2theta = 2*b1*b2/(b2-b1)
twotheta = numpy.arctan(tan2theta)
best_slope = numpy.tan(0.5*twotheta)
best_offs = y.mean() - best_slope*x.mean()
print "Best fit parameters: a=%.10f  b=%.10f"%(best_offs,best_slope)
```

The most remarkable fact is that Pearson applied the 'effective variance' method, formulated at a later date, to an unweighted orthogonal fit, as can be observed in the second plot in the figure. Pearson's best-fit parameters are the same as the best-fit parameters we find with the effective variance method (look in the output below). In an extended version of the program above, we added the effective variance method and added the offset and slope for the bisector of the two regression lines (y on x and x on y). The results are shown in the next figure. Note that Pearson's best-fit line is **not** the same as the bisector which has no relation to orthogonal fitting procedures.

Note: Pearson's method is an example of an orthogonal fit procedure. It cannot handle weights nor does it give you estimates of the errors on the best-fit parameters. We discussed the method because it is historically important and we wanted to prove that *kmpfit* can be used for its implementation.

Note: In the example we find best-fit values for the angle θ from which we derive the slope $b = \tan(\theta)$. The advantage of this method is that it also finds fits for data points that represent vertical lines.

Example: kmpfit_Pearsonsdata.py - Pearsons data and method (1901)

The output of this program is:

```
Analytical solution
===================
Best fit parameters: a=5.7840437745  b=-0.5455611975
Pearson's Corr. coef:  -0.976475222675
Pearson's best tan2theta, theta, slope:  -1.55350214417 -0.49942891481 -0.545561197521
b1 (Y on X), slope:  -0.539577274984 -0.539577274984
b2 (X on Y), slope -1.76713124274 -0.565888925403

======== Results kmpfit: effective variance =========
Params:                  5.78404377469 -0.545561197496
Covariance errors:       [ 0.68291482  0.11704321]
Standard errors          [ 0.18989649  0.03254593]
Chi^2 min:               0.618572759437
Reduced Chi^2:           0.0773215949296

======== Results kmpfit Y on X =========
Params:                  [5.7611851899974615, -0.4948059176648682]
Covariance errors:       [ 0.59895647  0.10313386]
Standard errors          [ 0.1894852   0.03262731]
Chi^2 min:               0.800663522236
Reduced Chi^2:           0.100082940279

======== Results kmpfit X on Y =========
Params:                  (10.358385598025167, 5.2273490890768901)
Covariance errors:       [ 0.94604747  0.05845157]
Standard errors          [ 0.54162728  0.03346446]
Chi^2 min:               2.62219628339
Reduced Chi^2:           0.327774535424

Least squares solution
======================
a1, b1 (Y on X) 5.76118519 -0.539577274869
a2, b2 (X on Y) 5.86169569507 -0.565888925412
Best fit tan2theta, Theta, slope:  -1.5535021437 -0.499428914742 -0.545561197432
Best fit parameters: a=5.7840437742  b=-0.5455611974
Bisector through centroid a, b:  5.81116055121 -0.552659830161
```

Comparisons of weighted fits methods

York *[Yor]* added weights to Pearsons data. This data set is a standard for comparisons between fit routines for weighted fits. Note that the weights are given as w_{x_i} which is equivalent to $1/\sigma^2_{x_i}$.

```
>>> x = numpy.array([0.0, 0.9, 1.8, 2.6, 3.3, 4.4, 5.2, 6.1, 6.5, 7.4])
>>> y = numpy.array([5.9, 5.4, 4.4, 4.6, 3.5, 3.7, 2.8, 2.8, 2.4, 1.5])
>>> wx = numpy.array([1000.0,1000,500,800,200,80,60,20,1.8,1.0])
>>> wy = numpy.array([1,1.8,4,8,20,20,70,70,100,500])
```

This standard set is the data we used in the next example. This program compares different methods. One of the methods is the approach of Williamson *[Wil]* using an implementation described in *[Ogr]*.

Example: kmpfit_Pearsonsdata_compare - Pearson's data with York's weights

Part of the output of this program is summarized in the next table.

Literature results:

Reference	a	b
Pearson unweighted	5.7857	-0.546
Williamson	5.47991022403	-0.48053340745
Reed	5.47991022723	-0.48053340810
Lybanon	5.47991025	-0.480533415

Practical results:

Method	a	b
kmpfit unweighted	5.76118519259	-0.53957727555
kmpfit weights in Y only	6.10010929336	-0.61081295310
kmpfit effective variance	5.47991015994	-0.48053339595
ODR	5.47991037830	-0.48053343863
Williamson	5.47991022403	-0.48053340745

>From these results we conclude that *kmpfit* with the effective variance residuals function, is very well suited to perform weighted orthogonal fits for a model that represents a straight line. If you run the program, you can observe that also the uncertainties match.

To study the effects of weights and to compare residual functions based on a combination of (2.88) and (2.84) and on the effective variance formula in (2.90) we made a small program which produces random noise for the model data and random weights for the measured data in both x an y. It also compares the results of these methods with SciPy's ODR routine. If you run the program you will observe that the three methods agree very well.

Example: kmpfit_errorsinXandYPlot - Comparing methods using random weights

Effective variance method for various models

Model with an x and 1/x factor

$$\mathbf{f([a,b],x) = ax - b/x}$$

We used data from an experiment described in Orear's article *[Ore]* to test the effective variance method. Orear starts with a model $f([a,b],x) = ax - b/x$. He tried to minimize the objective function by an iteration using (2.87) with the derivative $f'([a,b],x) = a + b/x^2$ and calls this the exact solution. He also iterates using the effective variance method as in (2.89) and finds small differences between these methods. This must be the result of an insufficient convergence criterion or numerical instability because we don't find a significant difference using these methods in a program (see example below). The corresponding residual function for the minimum distance expression is:

```python
def residuals3(p, data):
   # Minimum distance formula with expression for x_model
   a, b = p
   x, y, ex, ey = data
   wx = numpy.where(ex==0.0, 0.0, 1.0/(ex*ex))
   wy = numpy.where(ey==0.0, 0.0, 1.0/(ey*ey))
   df = a + b/(x*x)
   # Calculated the approximate values for the model
   x0 = x + (wy*(y-model(p,x))*df)/(wx+wy*df*df)
   y0 = model(p,x0)
   D = numpy.sqrt( wx*(x-x0)**2+wy*(y-y0)**2 )
   return D
```

The residual function for the effective variance is:

```
def residuals(p, data):
   # Residuals function for data with errors in both coordinates
   a, b = p
   x, y, ex, ey = data
   w = ey*ey + ex*ex*(a+b/x**2)**2
   wi = numpy.sqrt(numpy.where(w==0.0, 0.0, 1.0/(w)))
   d = wi*(y-model(p,x))
   return d
```

The conclusion, after running the example, is that *kmpfit* in combination with the effective variance method finds best-fit parameters that are better than the published best-fit parameters (because a smaller value for the minimum chi-square is obtained). The example shows that for data and model like Orear's, the effective variance, which includes uncertainties both in x and y, produces a better fit than an Ordinary Least-Squares (OLS) fit where we treat errors in x as being much smaller than the errors in y.

Example: kmpfit_Oreardata - The effective variance method with Orear's data

Model parabola

$\mathbf{f([a,b,c],x) = ax^2 + bx + c}$

Applying the effective variance method for a parabola we use the objective function:

$$\chi^2 = \sum_{i=0}^{N-1} \frac{(y_i - a - bx_i)^2}{\sigma_{y_i}^2 + \sigma_{x_i}^2 (b + 2cx)^2} \tag{2.102}$$

and we write the following residuals function for *kmpfit*:

```
def residuals(p, data):
   # Model: Y = a + b*x + c*x*x
   a, b, c = p
   x, y, ex, ey = data
   w = ey*ey + (b+2*c*x)**2*ex*ex
   wi = numpy.sqrt(numpy.where(w==0.0, 0.0, 1.0/(w)))
   d = wi*(y-model(p,x))
   return d
```

How good is our Taylor approximation here? Using $f(x) \approx f(x_i) + (x - x_i)(b + 2cx_i)$ we find that *f(x)* can be approximated by: $f(x) = a + bx + cx^2 - c(x - x_i)^2$. So this approximation works if the difference between x_i and x remains small. For *kmpfit* this implies that also the initial parameter estimates must be of reasonable quality. Using the code of residuals function above, we observed that this approach works adequately. It is interesting to compare the results of *kmpfit* with the results of Scipy's ODR routine. Often the results are comparable. That is, if we start with model parameters `(a, b, c) = (-6, 1, 0.5)` and initial estimates `beta0 = (1,1,1)`, then *kmpfit* (with smaller tolerance than the default) obtains a smaller value for chi square in 2 of 3 trials. With initial estimates `beta0 = (1.8,-0.5,0.1)` it performs worse with really wrong fits.

Note: *kmpfit* in combination with the effective variance method is more sensitive to reasonable initial estimates than Scipy's ODR.

Example: kmpfit_ODRparabola - The effective variance method for a parabola

Model with a sine function

$\mathbf{f([a,b,c],x) = a\sin(bx + c)}$

If your model is not linear in its parameters, then the effective variance method can still be applied. If your model is given for example by $f(x) = a\sin(bx + c)$, which is not linear in parameter b, then $f'(x) = ab\cos(bx + c)$ and the effective variance in relation (2.91) can be implemented as:

```python
def model(p, x):
   # Model: Y = a*sin(b*x+c)
   a,b,c = p
   return a * numpy.sin(b*x+c)

def residuals(p, data):
   # Merit function for data with errors in both coordinates
   a, b, c = p
   x, y, ex, ey = data
   w1 = ey*ey + (a*b*numpy.cos(b*x+c))**2*ex*ex
   w = numpy.sqrt(numpy.where(w1==0.0, 0.0, 1.0/(w1)))
   d = w*(y-model(p,x))
   return d
```

In the next script we implemented the listed model and residuals function. The results are compared with SciPy's ODR routine. The same conclusion applies to these results as to the results of the parabola in the previous section.

Note: *kmpfit* with effective variance can also be used for models that are not linear in their parameters.

Example: kmpfit_ODRsinus.py - Errors in both variables

2.3.8 Confidence- and prediction intervals

Experimenters often want to find the best-fit parameters *p* of their model to predict a value *f(p,x)* at given *x*. To get the predicted value $\hat{y}$ is trivial: $\hat{y} = f(p, x)$, but to estimate the error in $\hat{y}$ (σ_f) is not. Wolberg *[Wol]* starts with an expression for Δf:

$$\Delta f \cong \frac{\partial f}{\partial p_1}\Delta p_1 + \frac{\partial f}{\partial p_2}\Delta p_2 + \cdots + \frac{\partial f}{\partial p_n}\Delta p_n \tag{2.103}$$

which is a Taylor expansion of the error in y neglecting higher order terms. If one repeats the experiment many times, Wolberg finds an expression for the average error $\overline{\Delta f}^2 = \sigma_f^2$ in terms of the elements of the covariance matrix:

$$\sigma_f^2 = \sum_{j=0}^{j=n}\sum_{k=0}^{k=n} \frac{\partial f}{\partial p_j}\frac{\partial f}{\partial p_k} C_{jk} \tag{2.104}$$

which implies, as already seen in (2.103) that this error includes all variances and covariances in the covariance matrix. Note that for unit weighting or relative weighting we need to rescale the covariance matrix elements with χ_ν^2, and get:

$$\sigma_f^2 = \chi_\nu^2 \sum_{j=0}^{j=n}\sum_{k=0}^{k=n} \frac{\partial f}{\partial p_j}\frac{\partial f}{\partial p_k} C_{jk} \tag{2.105}$$

This *confidence* interval is interpreted as the region in which there is a probability of 68.3% to find the true value of *f*. To find a confidence region for another probability (e.g. 95%), we need to scale the error using Student-t statistics. If we use $100(1 - \alpha)$ percent to define the confidence interval on any fitted $\hat{y}_i$, then the scale factor is $t_{\alpha/2,\nu}$. where *t* is the upper $\alpha/2$ critical value for the t distribution with N-n degrees of freedom. All the information needed to construct a confidence interval can be found in *kpmfit*'s Fitter object:

- Degrees of freedom ν = `Fitter.dof`

- Reduced chi square: χ^2_ν = `Fitter.rchi2_min`
- Covariance matrix: C = `Fitter.covar`
- Best-fit parameters: p = `Fitter.params`

Confidence bands are often used in plots to give an impression of the quality of the predictions. To calculate confidence bands we vectorize (2.105):

$$CB = \hat{y} \pm \sigma_f \tag{2.106}$$

which is the short version of:

$$\boxed{CB = \hat{y} \pm t_{\alpha/2,\nu}\sqrt{\chi^2_\nu \sum_{j=0}^{j=n}\sum_{k=0}^{k=n}\frac{\partial f}{\partial p_j}\frac{\partial f}{\partial p_k}C_{jk}}} \tag{2.107}$$

If your model f is for example a parabola $f(x) = a + bx + cx^2$, then we have derivatives:

$$\frac{\partial f}{\partial p_0} = \frac{\partial f}{\partial a} = 1, \qquad \frac{\partial f}{\partial p_1} = \frac{\partial f}{\partial b} = x \quad \text{and} \quad \frac{\partial f}{\partial p_2} = \frac{\partial f}{\partial c} = x^2 \tag{2.108}$$

and the confidence band is calculated using:

$$CB = f(p,x) \pm t_{\alpha/2,\nu}\sqrt{\chi^2_\nu\ [(1\times 1)C_{00} + (1\times x)C_{01} + (1\times x^2)C_{02} + (x\times 1)C_{10} + \cdots\ (x^2\times x^2)C_{22}]} \tag{2.109}$$

The next code example shows a function which implements the confidence interval for a given model (variable `model` is a function or a lambda expression). The list `dfdp` is a list with derivatives evaluated at the values of `x`. The values in `x` need not to be the same values as the x coordinates of your data values. The code uses statistics module `stats.t` from SciPy to get the critical value for `t` with method `ppf` (*percent point function*). Then with the information in Fitter object `fitobj`, it creates a NumPy array with the lower values of the confidence interval (`lowerband`) and an array with the upper values of the confidence interval (`upperband`).

```python
def confidence_band(x, dfdp, alpha, fitobj, model, abswei):
   from scipy.stats import t
   # Given the confidence probability confprob = 100(1-alpha)
   # we derive for alpha: alpha = 1 - confprob
   alpha = 1.0 - confprob
   prb = 1.0 - alpha/2
   tval = t.ppf(prb, fitobj.dof)

   C = fitobj.covar
   n = len(fitobj.params)              # Number of parameters from covariance matrix
   p = fitobj.params
   N = len(x)
   if abswei:
      covscale = 1.0
   else:
      covscale = fitobj.rchi2_min
   df2 = numpy.zeros(N)
   for j in range(n):
      for k in range(n):
         df2 += dfdp[j]*dfdp[k]*C[j,k]
   df = numpy.sqrt(fitobj.rchi2_min*df2)
   y = f(p, x)
   delta = tval * df
   upperband = y + delta
   lowerband = y - delta
   return y, upperband, lowerband


def model(p, x):
   # Model: Y = a + b*x + c*x*x
   a,b,c = p
   return a + b*x + c*x*x


dfdp = [1, x, x**2]
alpha = 0.05
yhat, upperband, lowerband = confidence_band(x, dfdp, alpha, fitobj, model)
```

Confidence bands are plotted in the next program. It uses a 95% confidence probability to draw bands for a fit with weigths in y only and for a fit with errors both in x and y using the effective variance method. We used data and weights, so the weights should be treated as relative weights (`abswei=False`).

Example: kmpfit_ODRparabola_confidence - Confidence bands fit of parabola

With a small change in the confidence routine we can also derive a prediction interval. The values for a prediction band are derived from:

$$\sigma^2_{pred} = \sigma^2_f + \sigma^2_y \tag{2.110}$$

So we need the array with the data errors to derive the prediction interval. Note that this band is only smooth if we use unit weighting. Otherwise one observes a distorted band due to fluctuations in the weighting as demonstrated on the next example.

Example: kmpfit_example_partialdervs_confidence - Confidence bands fit of parabola

2.3.9 Special topics

Rejection of data with Chauvenet's criterion

With measurements one often finds one or more data points that appear isolated. If you are convinced that such data is a measurement error then of course you can throw it away or you can use a criterion based on the normal distribution using the (im)probability of large deviations. In this section we discuss a method to remove outliers where a data point is an outlier in the y direction only. The criterion we want discuss here is called Chauvenet's criterion (http://en.wikipedia.org/wiki/Chauvenet's_criterion). Suppose you have *N* measurements y_i from which we first calculate the mean and standard deviation. If a normal distribution is assumed, we can determine if the probability of a particular measurement is less than 1/2N (as proposed by the French mathematician Chauvenet). So if *P* is the probability then the criterion is:

$$P\left(\frac{y_i - \bar{y}}{\sigma}\right) < \frac{1}{2N} \tag{2.111}$$

In the next example we implemented this criterion to find outliers in a sample. We use the error function `scipy.special.erfc()` to calculate the probability *P* in the tails of the normal distribution. We implemented a clear and simple routine and a NumPy based function `chauvenet()` which is fast and efficient when we need to filter big arrays. This function returns an array of booleans. When an element in that array is *False*, we reject the corresponding data element in the data arrays:

```
def chauvenet(x, y, mean=None, stdv=None):
   #-----------------------------------------------------------
   # Input:  NumPy arrays x, y that represent measured data
   #         A single value of a mean can be entered or a
   #         sequence of means with the same length as
   #         the arrays x and y. In the latter case, the
   #         mean could be a model with best-fit parameters.
   # Output: It returns a boolean array as filter.
   #         The False values correspond to the array elements
   #         that should be excluded
   #
   # First standardize the distances to the mean value
   # d = abs(y-mean)/stdv so that this distance is in terms
   # of the standard deviation.
   # Then the  CDF of the normal distr. is given by
   # phi = 1/2+1/2*erf(d/sqrt(2))
   # Note that we want the CDF from -inf to -d and from d to +inf.
   # Note also erf(-d) = -erf(d).
   # Then the threshold probability = 1-erf(d/sqrt(2))
   # Note, the complementary error function erfc(d) = 1-erf(d)
   # So the threshold probability pt = erfc(d/sqrt(2))
   # If d becomes bigger, this probability becomes smaller.
   # If this probability (to obtain a deviation from the mean)
   # becomes smaller than 1/(2N) than we reject the data point
   # as valid. In this function we return an array with booleans
   # to set the accepted values.
   #
   # use of filter:
   # xf = x[filter]; yf = y[filter]
   # xr = x[~filter]; yr = y[~filter]
   # xf, yf are cleaned versions of x and y and with the valid entries
   # xr, yr are the rejected values from array x and y
   #-----------------------------------------------------------
   if mean is None:
      mean = y.mean()           # Mean of incoming array y
   if stdv is None:
      stdv = y.std()            # Its standard deviation
   N = len(y)                   # Length of incoming arrays
   criterion = 1.0/(2*N)        # Chauvenet's criterion
   d = abs(y-mean)/stdv         # Distance of a value to mean in stdv's
   d /= 2.0**0.5                # The left and right tail threshold values
   prob = erfc(d)               # Area normal dist.
   filter = prob >= criterion   # The 'accept' filter array with booleans
   return filter                # Use boolean array outside this function
```

In the next example we use the model with the best fit parameters a the mean and the standard deviation of the residuals as the standard deviation for all data points. Note that removing these type of outliers do not change the values of the best-fit parameters much.

Example: kmpfit_chauvenet.py - Exclude bad data with criterion of Chauvenet

Another example uses data from *[BRo]*. A weighted fit gives a value of chi-squared which is too big to accept the hypothesis that the data is consistent with the model. When we use the model and its best-fit parameters as mean and the errors on the data as standard deviation in the function `chauvenet()`, then one data point is excluded. When we redo the fit, we find a value for chi-squared that is small enough to accept the Null hypothesis that data and model are consistent.

Example: kmpfit_chauvenet2.py - Apply Chauvenet for a weighted fit

For outliers in the x direction, one need different methods.

Variance Reduction

To value a model we use a technique called Variance Reduction *[Wol]*. It can be applied to both linear and nonlinear models. Variance Reduction (VR) is defined as the percentage of the variance in the dependent variable that is explained by the model. The variance of the sample is given by:

$$\sigma_s = \frac{\sum\limits_{i=0}^{N-1} (y_i - \bar{y})^2}{N-1} \tag{2.112}$$

The variance given by the model with its best-fit parameters is:

$$\sigma_m = \frac{\sum\limits_{i=0}^{N-1} (y_i - y_{model})^2}{N-1} \tag{2.113}$$

The Variance Reduction is defined as:

$$VR = 100 * \left(1 - \frac{\sigma_m^2}{\sigma_s^2}\right) = 100 * \left(1 - \frac{\sum\limits_{i=0}^{N-1} (y_i - \bar{y})^2}{\sum\limits_{i=0}^{N-1} (y_i - y_{model})^2}\right) \tag{2.114}$$

If the quality of your model is good and your data is well behaved, then the model variance is small and the VR is close to 100%. Wrong models, or data with outliers have lower values, which even can be negative. We use VR to identify outliers in data where one (or more) points have a significant error in *x*. If we calculate the VR for samples where we exclude one data point and repeat this action for all data points, then it is possible to identify the outlier because exclusion of this outlier will improve the VR significantly. Note that for this type of outliers, one cannot use Chauvenet's criterion because the initial (bad) fit is required to exclude data points.

The VR can be calculated in a script as follows:

```
fitter.fit(params0=params0)                     # Find best-fit parameters
varmod = (y-model(fitter.params,x))**2.0        # The model variance
varmod = varmod.sum()/(N-1)
vardat = y.var()                                # Sample variance
# A vr of 100% implies that the model is perfect
# A bad model gives much lower values (sometimes negative)
vr = 100.0*(1-(varmod/vardat))
```

Below, the script that uses the VR to identify an outlier. It removes the data point that, when omitted, improves the VR the most.

Example: kmpfit_varreduct.py - Use Variance Reduction to identify outlier

In *[Wol]* an example is given of data and a fit with using a good model and a bad model. The difference between those models should be clear if we inspect the VR of both. With `kmpfit_varreduct_wol.py` we reproduced table 3.4.1 of *[Wol]* for both weighted and unweighted fits. We get the same values, with only a small deviation of the weighted fit with the straight line model (*[Wol]* gives -48.19, which is probably a typo). The data was derived from a parabolic model so we know that a parabola should be the most suitable model. From the table we learn that indeed the parabola gives the best VR. For weighted fits, the result is even more obvious because the errors on the data increase if the distance from the bottom of the parabola increases. For a weighted fit this is a recipe to get a bad value for the VR.

Model	$w_i = 1$	$w_i = 1/\sigma_i^2$
$y = a + bx$	+80.29	-48.29
$y = a + bx + cx^2$	+99.76	+99.72

Example: kmpfit_varreduct_wol.py - Use Variance Reduction to examine model

Regression through the origin

In this section we address a special case of linear regression using an analytical method. It is a regression through the origin. It is used in a practical course where students need to find the Hubble constant after they obtained a number of galaxy velocities and distances. Hubble's constant can be found if you find the slope of the best fit straight line through the data points (distance in Mpc and velocity in Km/s) and the origin (assuming velocity is zero when the distance is zero).

Hubble's first fits allowed for an offset and he found an age of the universe that was much too small. Now we know the theoretical base and the fit is reduced to a problem that is known as 'regression through the origin'.

For a model $y = a + bx$ we defined chi squared as:

$$\chi^2 = \sum_{i=0}^{N-1} \left(\frac{y_i - a - bx_i}{\sigma_i}\right)^2 \tag{2.115}$$

For regression through the origin (leaving parameter *a* out of the equations) we find for the minimum chi squared:

$$0 = \frac{\partial \chi^2}{\partial b} = -2 \sum_{i=0}^{N-1} \frac{x_i(y_i - bx_i)}{\sigma_i^2} \tag{2.116}$$

from which we derive an expression for slope *b*:

$$b = \frac{\sum \frac{x_i y_i}{\sigma_i^2}}{\sum \frac{x_i^2}{\sigma_i^2}} \tag{2.117}$$

For the standard error in *b* we follow the procedure described in section *Standard errors in weighted fits* (2.27). The error is defined as:

$$\sigma_b^2 = \sum_i \sigma_i^2 \left(\frac{\partial b}{\partial y_i}\right)^2 \tag{2.118}$$

with:

$$\frac{\partial b}{\partial y_i} = \frac{\partial \left(\frac{\sum \frac{x_i y_i}{\sigma_i^2}}{\sum \frac{x_i^2}{\sigma_i^2}}\right)}{\partial y_i} \tag{2.119}$$

where S_{xx} does not depend on y_i. With the notation $S_{xx} = \sum x_i^2/\sigma_i^2$ we write this as:

$$\frac{\partial b}{\partial y_i} = \frac{1}{S_{xx}} \frac{\partial \left(\sum \frac{x_i y_i}{\sigma_i^2}\right)}{\partial y_i} \tag{2.120}$$

Therefore:

$$\frac{\partial b}{\partial y_i} = \frac{x_i}{S_{xx}} \tag{2.121}$$

Inserting this in (2.118) gives:

$$\begin{aligned}
\sigma_b^2 &= \sum_i \sigma_i^2 \left(\frac{x_i}{S_{xx}}\right)^2 \\
&= \frac{1}{S_{xx}^2} \sum_i \sigma_i^2 \frac{x_i^2}{\sigma_i^2} \\
&= \frac{1}{S_{xx}^2} S_{xx} \\
&= \frac{1}{S_{xx}}
\end{aligned} \tag{2.122}$$

So finally:

$$\boxed{\sigma_b = \sqrt{\frac{1}{S_{xx}}}} \tag{2.123}$$

In a small program we will demonstrate that this error is the real 1 sigma error for when we exactly know what the errors on the data points are. For weights that are unit or if weights are scaled, we should scale the error on the fitted parameter with the square root of the reduced chi-squared (as described in *Reduced chi squared*).

The reduced Chi-squared for a regression through the origin is (note we have one parameter less to fit compared to a regression which is not forced to go through the origin):

$$\chi_\nu^2 = \frac{1}{N-1}\sum_{i=0}^{N-1}\frac{(y_i - bx_i)^2}{\sigma_i^2} \tag{2.124}$$

Then:

$$\sigma_b = \sqrt{\frac{\chi_\nu^2}{\sum x_i^2}} \tag{2.125}$$

This is a two pass algorithm because first you have to find slope b to get the reduced chi-squared. Note that in many references, the unweighted version of the χ_ν^2 is used to derive the error in slope b. This gives wrong results as can be seen with equal weighting. Many references give the wrong formula, so be careful. A possible implementation of the formulas above is given in the function `lingres_origin()`:

```python
def lingres_origin(xa, ya, err):
   # Apply regression through origin
   N = len(xa)
   w = numpy.where(err==0.0, 0.0, 1.0/(err*err))
   sumX2 = (w*xa*xa).sum()
   sumXY = (w*xa*ya).sum()
   sum1divX = (1/(w*xa)).sum()
   b = sumXY/sumX2
   sigma_b = 1.0/sumX2
   chi2 = (w*(ya-b*xa)**2).sum()
   red_chi2 = chi2 / (N-1)
   sigma_b_scaled = red_chi2 / sumX2
   return b, numpy.sqrt(sigma_b), numpy.sqrt(sigma_b_scaled)
```

Next we show an example of estimating the Hubble constant using data pairs (distance, velocity) found in lab experiments. We use both the analytical method described above and *kmpfit* to compare the results. We included the fast NumPy based function to filter possible outliers using Chauvenet's criterion. This criterion was discussed in the previous section. As a mean, we do not use the mean of the sample, but the model with the best fit parameters. As standard deviation we use the (artificial) errors on the data as we did in the second example of Chauvenet's criterion.

We also included a loop which gives the variance reduction when we omit one data point. The variance reduction for the unfiltered data is low which implies that the model is not the best model or that we have one or more outliers:

```
Variance reduction unfiltered data: 37.38%

      Excluded data       chi^2  red.chi^2         VR
=====================================================
(   42.00,  1294.00)      32.56       4.65      80.55
(    6.75,   462.00)     101.76      14.54      31.44
(   25.00,  2562.00)      65.93       9.42      41.20
(   33.80,  2130.00)     101.49      14.50      28.46
(    9.36,   750.00)     100.85      14.41      36.82
(   21.80,  2228.00)      75.80      10.83      44.28
(    5.58,   598.00)       99.94       14.28       35.27
(    8.52,   224.00)       99.45       14.21       26.44
(   15.10,   971.00)      101.73       14.53       38.26
========================================================
```

Based on this table we can conclude that data point (42,1294) can be regarded as an outlier. Removing this point decreases the variance of the data with respect to the model, significantly, which results in a big improvement of the variance reduction. In this case, a filter based on exclusion of data based on variance reduction, improves the fit more than a filter based on Chauvenet's criterion.

Example: kmpfit_hubblefit.py - Find Hubble constant with fit of line through origin

Fitting 2D data

Finding best-fit parameters of an ellipse

In many astronomical problems, the ellipse plays an important role. Examples are planetary orbits, binary star orbits, projections of galaxies onto the sky. etc. For an overview of ellipse properties and formulas, please visit Wolfram's page about ellipses at http://mathworld.wolfram.com/Ellipse.html Assume we got a number of measurements of the orbit of a binary system and all sky positions are converted to a rectangular grid positions (i.e. x,y coordinate pairs). If one makes a plot of these positions it is usually obvious if have to deal with an elliptical orbit. To estimate typical orbit parameters (e.g. in Kepler's laws of planetary motion) we have to estimate the best-fit ellipse parameters. These parameters are the position of the center of the ellipse, the length of the major and minor axes and the position angle (its rotation). If we want to fit ellipse parameters we have to find a suitable relation between y and x first. The equation for an unrotated ellipse with semi-major axis **a** and semi-minor axis **b** is:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{2.126}$$

Rotation of the ellipse follows the mathematical standard, i.e. an angle is positive if it is counted anti-clockwise. So if we want an expression for a rotated ellipse we use the rotation recipe:

$$\begin{aligned} x' &= x\cos(\phi) - y\sin(\phi) \\ y' &= x\sin(\phi) + y\cos(\phi) \end{aligned} \tag{2.127}$$

If the origin is not centered at position (0,0) then we need a translation also:

$$\begin{aligned} x'' &= x' + x_0 \\ y'' &= y' + y_0 \end{aligned} \tag{2.128}$$

Introduce a new function Z which depends on variables x and y.

$$Z(x, y) = \frac{x^2}{a^2} + \frac{y^2}{b^2} \tag{2.129}$$

This function is plotted in the surface plot below. The ellipse in this landscape can be found at a height 1.0. We projected the ellipse on the xy plane to prove that the two contours correspond. You can run the example and rotate the 3D plot to get an impression of the landscape. For a data position (x, y) which is exactly on the ellipse $Z(x, y) = 1$. But if not, then $Z(x, y)$ deviates from 1.0 and it is a measure for the deviations we are trying to minimize with a least squares fit. Note that the values in (x, y) represents the data on the ellipse. So in fact the should be written as (x'', y''). To calculate $Z(x, y)$, we need to calculate (x', y') first and from those coordinates the values of (x, y).

In the code example at the end of this section we need a list with positions that we want to use to make a fit. The data we used can be found in `ellipse.dat`. It is data from an artificial ellipse with origin at (5,4) semi-major axis is 10, semi-minor axis is 3. Its angle is 60 degrees. Noise was added to simulate real data.

But usually we don't know about the properties of the ellipse represented by the data so we need a routine that calculates these estimates automatically. For the ellipse there is a method based on *image moments analysis* (http://en.wikipedia.org/wiki/Image_moments) that can do the job.

$$M_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x^p y^q f(x,y)\,dx\,dy \tag{2.130}$$

The zeroth and first moments for the given set data points (positions) are given by:

$$\begin{aligned} m_{00} &= \sum_i \sum_j f_{ij} \\ m_{10} &= \sum_i \sum_j x\, f_{ij} \\ m_{01} &= \sum_i \sum_j y\, f_{ij} \end{aligned} \tag{2.131}$$

In an image the zeroth moment represents the area of an object. For our positions (x, y) it is just the number of positions. Note that our data points are just positions and not image pixels with an intensity. So the value of f is 1 for a position from the file and 0 for others (but there are no others because we don't have an image, just the values in (x, y). Therefore we need only to loop over all our positions and do the necessary summing. Then the coordinates of the centroid (center of mass) are:

$$\begin{aligned} \bar{x} &= \frac{m_{10}}{m_{00}} \\ \bar{y} &= \frac{m_{01}}{m_{00}} \end{aligned} \tag{2.132}$$

which is an estimate of the central position of the ellipse. How can we find an estimate for the other parameters? First we define the so called central moments of the sample:

$$\mu_{pq} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} (x-\bar{x})^p (y-\bar{y})^q f(x,y) dx dy \tag{2.133}$$

Now define:

$$\begin{aligned} \mu'_{20} &= \frac{\mu_{20}}{\mu_{00}} = \frac{M_{20}}{M_{00}} - \bar{x}^2 \\ \mu'_{02} &= \frac{\mu_{02}}{\mu_{00}} = \frac{M_{02}}{M_{00}} - \bar{y}^2 \\ \mu'_{11} &= \frac{\mu_{11}}{\mu_{00}} = \frac{M_{11}}{M_{00}} - \bar{x}\bar{y} \end{aligned} \tag{2.134}$$

With these definitions, one can derive the following relations:

$$\begin{aligned} \theta &= \frac{1}{2}\arctan\left(\frac{2\mu'_{11}}{\mu'_{20} - \mu'_{02}}\right) \\ \lambda_i &= \frac{\mu'_{20} + \mu'_{02}}{2} \pm \frac{\sqrt{4{\mu'}_{11}^2 + ({\mu'}_{20} - {\mu'}_{02})^2}}{2} \end{aligned} \tag{2.135}$$

θ gives us estimate for the angle and λ_i the (squared) length of the semi-major and semi-minor axes. We implemented these relations in a routine that finds initial estimates of the parameters of an ellipse based on the moments analysis above:

```python
def getestimates( x, y ):
   """
   Method described in http://en.wikipedia.org/wiki/Image_moments
   in section 'Raw moments' and 'central moments'.
   Note that we work with scalars and not with arrays. Therefore
   we use some functions from the math module because the are
   faster for scalars
   """
   m00 = len(x)
   m10 = numpy.add.reduce(x)
   m01 = numpy.add.reduce(y)
   m20 = numpy.add.reduce(x*x)
   m02 = numpy.add.reduce(y*y)
   m11 = numpy.add.reduce(x*y)

   Xav = m10/m00
   Yav = m01/m00

   mu20 = m20/m00 - Xav*Xav
   mu02 = m02/m00 - Yav*Yav
   mu11 = m11/m00 - Xav*Yav

   theta = (180.0/numpy.pi) * (0.5 * atan(-2.0*mu11/(mu02-mu20)))
   if (mu20 < mu02):                   # mu20 must be maximum
      (mu20,mu02) = (mu02,mu20)        # Swap these values
      theta += 90.0

   d1 = 0.5 * (mu20+mu02)
   d2 = 0.5 * sqrt( 4.0*mu11*mu11 + (mu20-mu02)**2.0 )
   maj = sqrt(d1+d2)
   min = sqrt(d1-d2)
   return (Xav, Yav, maj, min, theta)
```

If you study the code of the next example, you should be able to recognize the formulas we used in this section to get initial estimates and residuals. The applied method can be used for many fit problems related to 2D data.

Example: kmpfit_ellipse.py - Find best-fit parameters of ellipse model

2.3.10 Glossary

Objective Function An *Objective Function* is a function associated with an optimization problem. It determines how good a solution is. In Least Squares fit procedures, it is this function that needs to be minimized.

Independent Variable Usually the **x** in a measurement. It is also called the explanatory variable

Dependent Variable Usually the **y** in a measurement. It is also called the response variable

LLS Linear Least-Squares

NLLS Non-Linear Least Squares

Numpy NumPy is the fundamental package needed for scientific computing with Python. See also information on the Internet at: numpy.scipy.org

SE Standard error

WSSR Weighted Sum of Squared Residuals (WSSR)

2.3.11 References

See Bibliography.

2.4 Tutorial tabarray module

2.4.1 Introduction

Many applications send output of numbers to plain text (*ASCII*) files in a rectangular form. I.e. they store human readable numbers in one ore more columns in one or more rows. In one of our example figures to illustrate the use of graticules we plotted coastline data from a text file with coordinates in longitude and latitude in a `wcs` supported projection.

If you want to plot such data or you need data to do calculations, then you need a function or method to read the data into for example NumPy arrays. Package SciPy provides a function *read_array()* in module `scipy.io.array_import`. It has all necessary features to do the job but it is very slow when it needs to read big files with many numbers.

We wrote a fast version in module `tabarray` which is also part of the Kapteyn Package Its speed is comparable to a well known module that is no longer supported, called *TableIO*. The module interfaces with C and is written in *Cython*. Such interfaces improve the speed of reading the data with a factor of 5-10 compared to Python based solutions. Module `tabarray` has a simple interface and an object oriented interface. For the simple interface functions we used the same function names as *TableIO*. This will simplify the migration from *TableIO* to `tabarray`.

2.4.2 Simple interface functions

Function readColumns

A typical example of a text data file is given below. It has 3 columns and several rows of which a number of rows represent a comment. To experiment with tabarray functions and methods you can copy this data and store it as *testdata.txt* on disk.:

```
! ASCII file 'testdata.txt' 12-09-2008
!
! X     |    Y    |     err
23.4     -44.12   1.0e-3
19.32    0.211    0.332
# Next numbers are include as
-22.2    44.2     3.2
1.2e3    800      1
```

Assuming you have some knowledge of the contents and structure of the file, it is easy to read it into NumPy arrays. We use variables *x*, *y* and *err* to represent the columns. The comment characters are '#' and '!' and are included in the comment string which is the second parameter of the `tabarray.Readcolumns()` function. The file on disk is identified by its name. There is no need to open it first. Use the commands given below to read all the data from our test file.

```
>>> from kapteyn import tabarray
>>> x,y,err = tabarray.readColumns('testtable.txt','#!')
>>> print x
[   23.4 ,     19.32,    -22.2 ,   1200.  ]
```

All numbers are converted to floating point. Blank lines at the end of a file are ignored. Blank lines in the middle of a file are treated as comment lines. Suppose you want to read only the second and third column, then one needs to specify the columns. The first column has index 0.

```
>>> y,err = tabarray.readColumns('testtable.txt','#!', cols=(1,2))
>>> print err
[  1.00000000e-03   3.32000000e-01   3.20000000e+00   1.00000000e+00]
```

Note: Column and row numbers start with 0. The last row or last column is addressed with -1.

To make a selection of rows you can specify the rows parameter. Rows are given as a sequence and the first row in a file has index 0. Suppose you want to read the last two rows from the last two columns in the text file together with the first row, then we could write:

```
>>> x,y = tabarray.readColumns('testtable.txt','#!', cols=(1,2), rows=(2,3,0))
>>> print x
[  44.2    800.     -44.12]
```

To read only the last row in your data you should use *rows=(-1,)*.

If you know beforehand which lines of the data files should be read, you can set the converter to read only the lines in parameter *lines*. For a big text file (called *satview.txt*) containing longitudes and latitudes of positions in two columns, we are only interested in the first 1000 lines containing relevant data. Then the *lines* parameter saves time. So we use the following command:

```
>>> lons, lats = tabarray.readColumns('satview.txt','s', lines=(0,1000))
```

Comment lines in this *satview.txt* file do not start with a common comment character, instead it starts with the word 'segment' so our comment character becomes 's'.

Function writeColumns

One dimensional array data can also be written back to a file on disk. The function for writing data is called `tabarray.writeColumns()`. Its first argument is the name of the file. The second is a sequence with columns. With the columns 'x' and 'y' from the *testtable.txt* file in the previous section, we want to write a new file where column 'y' is the first column and column 'x' is the second. Here is the code to do this:

```
>>> x,y,err = tabarray.readColumns('testtable.txt','#!')
>>> tabarray.writeColumns('testout.txt', (y,x))
# Contents on disk is:
     -44.12       23.4
      0.211      19.32
       44.2      -22.2
        800       1200
```

The columns are one dimensional NumPy arrays. This implies that we can do some array arithmetic on the columns. We could have changed our columns to:

```
>>> tabarray.writeColumns('testout.txt', (y*y,x*y,x*x))
# Contents on disk is:
    1946.57   -1032.41     547.56
   0.044521    4.07652    373.262
    1953.64    -981.24     492.84
     640000     960000   1.44e+06
```

which makes this function very powerful.

It is common practice to start text data file with some comments. The next code shows how to write a date and the name of the author in a new file with function `tabarray.writeColumns()`. The comments parameter is a list with strings. Each string is written on a new line at the start of the text file.

```
>>> when = datetime.datetime.now().strftime("Created at: %A (%a) %d/%m/%Y")
>>> author = 'Created by: Kapteyn'
>>> tabarray.writeColumns('testout.txt', (y*y,x*y,x*x), comment=[when, author])
```

The header of the file will look similar to this:

```
# Created at: Thursday (Thu) 18/09/2008
# Created by: Kapteyn
```

2.4.3 Tabarray objects and methods

Reading data and making selections

A *tabarray* object is created with method `tabarray.tabarray()`. Again we want to read the data from file 'testtable.txt'.

```
>>> t = tabarray.tabarray('testtable.txt', '#!')
>>> print t
[[  2.34000000e+01  -4.41200000e+01   1.00000000e-03]
 [  1.93200000e+01   2.11000000e-01   3.32000000e-01]
 [ -2.22000000e+01   4.42000000e+01   3.20000000e+00]
 [  1.20000000e+03   8.00000000e+02   1.00000000e+00]]
```

Selections are made with methods `tabarray.rows()` and `tabarray.columns()`.

Warning: The *rows()* method needs to be applied before the *columns()* method because for the latter, the array *t* is transposed and its row information is changed.

With this knowledge we can combine the methods in one statement to read a selection of lines and a selection of columns into NumPy arrays.

```
>>> x,y = t.rows((2,3)).columns((1,2))
>>> print x
[  44.2  800. ]
>>> print y
[ 3.2  1. ]
```

If you want to select rows in a NumPy vector that is already filled with data from disk after applying the lines and/or rows parameters you still can extract data using NumPy indexing:

```
>>> lines = [0,1,3]
>>> print err[lines]
[ 0.001  0.332  1.    ]
```

Messy files

ASCII text readers should be flexible and robust. Examine the contents of the next ASCII data file (which we stored on disk as *messyascii.txt*):

```
! Very messy data file

23.343, 34.434, 1e-20
```

```
10, 20, xx


2 4      600
-23.23, -0.0002, -3x7
   # Some comment

40, 50.0, 70.2
```

It contains blank lines at the end and between the data and it has three different separators (spaces, comma's and tabs). Also it contains data that cannot be converted to numbers. Instead of an exception we want the converter to substitute a user given value for a string that could not be converted to a number. Assume that a user wants -999 for those bad entries, then the numbers should be read by:

```
>>> t= tabarray.tabarray('messyascii.txt','#!', sepchar=' ,\t', bad=-999)
>>> print t
[[  2.33430000e+01   3.44340000e+01   1.00000000e-20]
 [  1.00000000e+01   2.00000000e+01  -9.99000000e+02]
 [  2.00000000e+00   4.00000000e+00   6.00000000e+02]
 [ -2.32300000e+01  -2.00000000e-04  -9.99000000e+02]
 [  4.00000000e+01   5.00000000e+01   7.02000000e+01]]
>>> x,y = t.rows(range(1,4)).columns((1,2))  # Extract some rows and columns
>>> print x
[  2.00000000e+01   4.00000000e+00  -2.00000000e-04]
>>>print y   # Contains the 'bad' numbers
[-999.  600. -999.]
```

Note that we could have used function `tabarray.readColumns()` also to get the same results:

```
>>> x,y = tabarray.readColumns('messyascii.txt','#!', sepchar=' ,/t', bad=-999, rows(range(1,4)), co
```

Note: Probably more useful as a bad number indicator is the 'Not a Number' (NaN) from NumPy. Use it as in: *bad=numpy.nan* and test on these numbers with NumPy's function: *isnan()*.

2.4.4 Glossary

ASCII *American Standard Code for Information Interchange* is a character-encoding scheme based on the ordering of the English alphabet.

CHAPTER 3

Background information

3.1 Background information module celestial

3.1.1 Rotation matrices

Spherical astronomy sections in older textbooks rely heavily on the reader's knowledge of spherical trigonometry (e.g. Smart 1977). For more complicated problems than a simple rotation, this technique becomes laborious. Matrix and vector techniques come to rescue. Many of the transformations are defined in terms of rotation matrices. A rotation matrix is a matrix whose multiplication with a vector rotates the vector while preserving its length. There is a special group 3 x 3 rotation matrices R where

$$|R| = \pm 1 \; and \; R^{-1} = R^T \tag{3.1}$$

For transformations between sky systems we only use matrices with $|R| = +1$.

A coordinate rotation is a rotation about a single coordinate axis. The three coordinate rotation matrices are:

$$R_1(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{3.2}$$

$$R_2(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} \tag{3.3}$$

$$R_3(\alpha) = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Three coordinate rotations in sequence can describe any rotation. The result matrix is:

$$R_{ijk}(\phi, \theta, \psi) = R_i(\phi) R_j(\theta) R_k(\psi) \tag{3.5}$$

The angles are called Euler angles. There are 27 possible sequences of the three indices i,j,k. Not all sequences are valid rotations. The most common choices of valid combinations are (1,2,3), (3,1,3) and (3,2,3) (*[Diebel]*, 2006)

If $\vec{r}_0$ is a position vector in system 0 and $\vec{r}_1$ is the **same position** in the sky but in another sky system then, with the appropriate rotation matrix *R*, we calculate $\vec{r}_1$ in the coordinates belonging to the rotated system with:

$$\vec{r}_1 = R\vec{r}_0 \tag{3.6}$$

Note that the listed rotations represent the same position in different coordinate systems. The indices 1,2,3 correspond to the rotation axes x, y, z. In this documentation we will write R_x for R_1, R_y for R_2 and R_z for R_3:

$$\begin{bmatrix} a_{x2} \\ a_{y2} \\ a_{z2} \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} a_{x1} \\ a_{y1} \\ a_{z1} \end{bmatrix} \tag{3.7}$$

If (α, δ) is the longitude and latitude of a position in system 0, then the corresponding position vector can be written as:

$$\vec{r}_0 = \begin{bmatrix} \cos\delta_0 \cos\alpha_0 \\ \cos\delta_0 \sin\alpha_0 \\ \sin\delta_0 \end{bmatrix} \tag{3.8}$$

Note that the longitude and latitude applies to the other sky systems too, but then we use other symbols, like (λ, β), (l,b) or (sgl, sgb). From any position (x,y,z) we calculate the longitude and latitude with the expressions:

$$\tan(lon) = y/x \tag{3.9}$$

and

$$\tan(lat) = z/\sqrt{x^2 + y^2} \tag{3.10}$$

where we used the arctan2 function to solve for (lon,lat) to keep the right quadrant. Longitudes range from $0°$ to $360°$ and latitudes from $-90°$ to $90°$.

3.1.2 FK4

The impression one could get from the literature is that FK4 and FK4-NO-E are different sky systems and that there exists a matrix to rotate a position from one system to the other. But this is not true. The systems differ because positions in FK4 catalogs usually contain the elliptic terms of aberration (so they are almost mean places). Others list positions that are corrected for these E-terms (like catalogs with radio sources). Also B1950 radio interferometer data (e.g. maps from the W.S.R.T.) could be processed in a way that positions are corrected for E-terms. It is convenient to define a system that is FK4 but without the E-terms. FITS uses the name FK4-NO-E for this system. Catalog positions corrected for the E-terms are (real) mean places and are used for precession and transformations from FK4 B1950 positions to FK5 J2000 positions and galactic coordinates.

In a later section we give the original definition of galactic coordinates.

3.1.3 FK4 and the elliptic terms of aberration

Stellar aberration is caused by the motion of the earth in its orbit. This motion is represented by a circular velocity component and a component perpendicular to the major axis caused by the fact that the orbit is elliptical. This velocity component is responsible for elliptical terms of aberration (E-terms) which are less than 0.35 arcseconds (maximum is equal to the constant of aberration times the eccentricity of the earths orbit = 20".496 x 0.01673 ~= 343 mas). The terms are independent of the position of the earth and depend only on the position of the object in the sky.

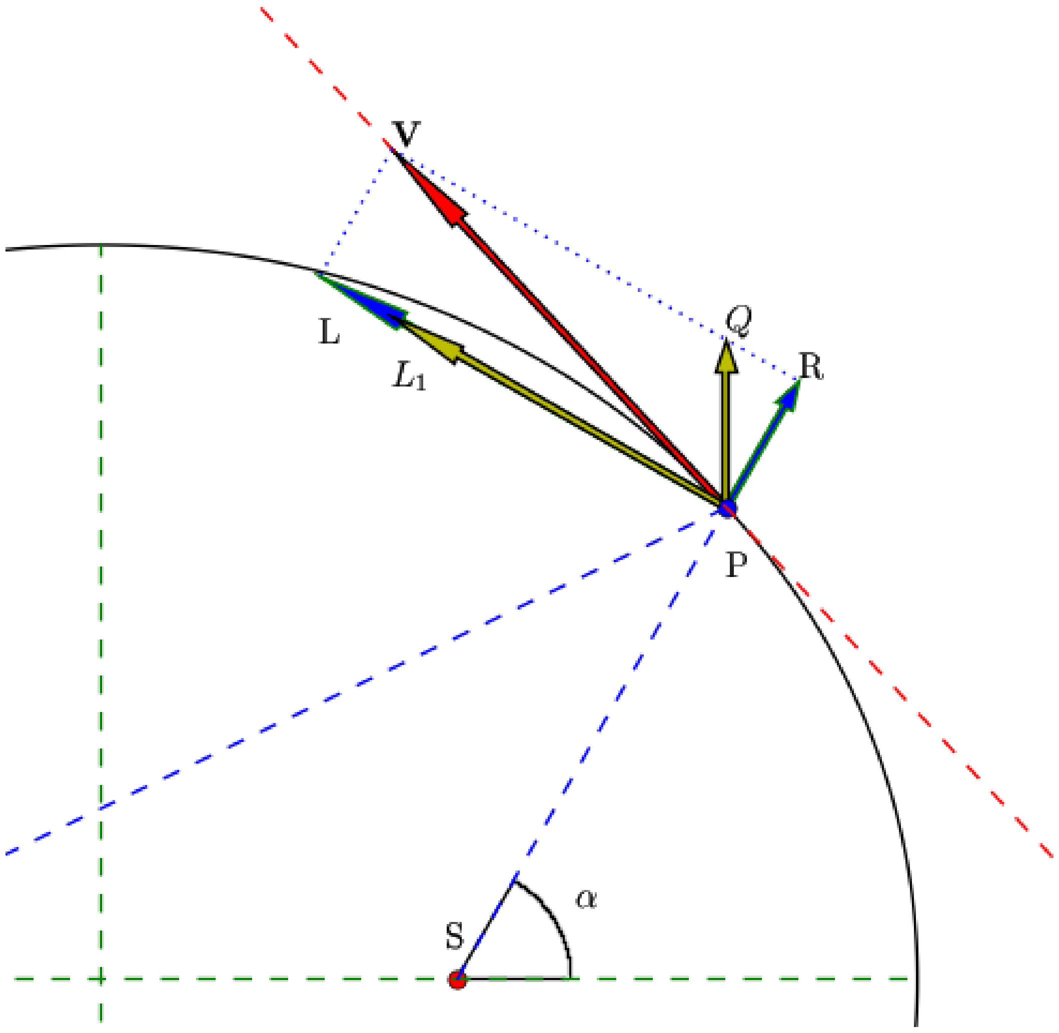


Fig.1 – Ecliptic from above showing e-terms.

Fig.1 shows the ecliptic from above. S is the Sun, in one of the focal points of the ellipse and P the position of the Earth. The plot was made with Python script `etermfig.py`.

Smart (1977) gives an excellent description of aberration and its elliptical terms. We reproduced one of his figures with a small program. Here are the steps.

- Given an elliptical orbit with semi major axis a and semi minor axis b, and center at (0,0), the positions of the focal points are (-c,0) and (c,0) with $c^2 = a^2 - b^2$
- Suppose the Sun is in focal point S and the Earth is on the ellipse in P
- The tangent in P is the normal of the bisector of the two lines from focal point to P
- **r** is the radius vector SP
- Earth has a velocity **V** along the tangent at P and:

$$V^2 = PL^2 + PR^2 = (\frac{dr}{dt})^2 + (r\frac{d\alpha}{dt})^2 \tag{3.11}$$

- So for given P and a velocity V, we can calculate the angle between the normal of SP (i.e. in the direction PL) and decompose **V** into a linear velocity perpendicular to the radius vector and a component in the direction of the radius vector
- Now we want to decompose **V** into a circular velocity component PL_1 and a velocity perpendicular to the major axis (PQ)
- $PQ = PR/\sin(\alpha)$ and $PL_1 = PL - PR/\tan(\alpha)$

Smart derives two epressions:

$$V_{PQ} = \frac{e\mu}{h} \tag{3.12}$$

$$V_{PL_1} = \frac{\mu}{h} \tag{3.13}$$

with:

$$\mu = G(M+m);\;\; h = r^2\frac{d\alpha}{dt} \tag{3.14}$$

With M is mass of the Sun, m is mass of the Earth, G is the gravitational constant and e is the eccentricity of the ellipse:

$$b^2 = a^2(1-e^2) \tag{3.15}$$

The most important observation now is that these velocities are constant! Therefore the total displacement of the position of a star due to aberration can be decomposed into a displacement due to a constant velocity at right angle to the radius vector and one due to a constant velocity perpendicular to the major axis.

If the position of a star is given by longitude λ and latitude β and the longitude (measured from the vernal equinox) is ω then the displacements due to the velocity perpendicular to the major axis are:

$$\begin{aligned}\Delta\lambda &= +e\kappa\sec(\beta)\cos(\omega-\lambda)\\ \Delta\beta &= +e\kappa\sin(\beta)\sin(\omega-\lambda)\end{aligned} \tag{3.16}$$

and κ is the constant of aberration (Smart section 108).

The constant of aberration is defined as:

$$\kappa = V_{PL_1}\frac{\csc(1'')}{c} \tag{3.17}$$

and c is the speed of light.

The value of κ is 20".496. Therefore, given the eccentricity of the Earth's orbit (0.01673), the maximum displacement in λ or β is 20".496 * 0.01673 ~= 343 mas.

Data in FK4 catalogs are 'almost' mean places because the conventional correction for annual aberration in FK4 includes only terms for circular motion and not the small E-terms. Therefore all published FK4 catalog positions are affected by elliptic aberration.

Mean places should be unaffected by aberration of any kind. Thus, for precession or transformation of FK4 positions, one should remove the E-terms first.

With a standard transformation from ecliptic coordinates to equatorial coordinates one can find expressions for the displacements in α and δ. (e.g. see ES, section 3.531, p 170):

$$\begin{aligned}\Delta\alpha = \alpha - \alpha_{cat} &= -(\Delta C\cos\alpha_{cat} + \Delta D sin\alpha_{cat}/(15\cos\delta_{cat})\\ \Delta\delta = \delta - \delta_{cat} &= -(\Delta D\cos\alpha_{cat} - \Delta C sin\alpha_{cat})sin\delta_{cat} - \Delta C\tan\epsilon\cos\delta_{cat}\end{aligned} \tag{3.18}$$

where ϵ is the obliquity of the ecliptic.

Also one could write a position vector in an equatorial system:

$$\vec{r_0} = \begin{bmatrix} \cos\delta_0 \cos\alpha_0 \\ \cos\delta_0 \sin\alpha_0 \\ \sin\delta_0 \end{bmatrix} \tag{3.19}$$

and a second vector:

$$\vec{r_1} = \begin{bmatrix} \cos(\delta_0 + \Delta\delta)\cos(\alpha_0 + \Delta\alpha) \\ \cos(\delta_0 + \Delta\delta)\sin(\alpha_0 + \Delta\alpha) \\ \sin(\delta_0 + \Delta\delta) \end{bmatrix} \tag{3.20}$$

then one can define the E-term vector as:

$$\vec{E} = \vec{r_1} - \vec{r_0} \tag{3.21}$$

If one works out this difference between two vectors, neglect terms that are very small and rearrange the equations so that we can compare them to the expressions for the displacements in α and δ, then the E-term vector is equal to:

$$\vec{E} = \begin{bmatrix} -\Delta D \\ +\Delta C \\ \Delta C\tan(\epsilon) \end{bmatrix} \tag{3.22}$$

This E-term vector can then be used to transform FK4 positions to real mean places (i.e. remove E-terms) or to convert mean places to FK4 catalog positions (i.e. add E-terms).

Module `celestial` calculates the E-term vector in the equatorial system as function of epoch. Removing and adding E-term vectors are best illustrated in a figure. In the next plot, the red circle represents the FK4 catalog system. For each unit vector in this circle one can transform a position in RA, Dec to a new position where the E-terms are removed. The new vector has its end point on the blue circle. So adding E-terms would be as simple as adding the E-term vector to the new vector. However, if one converts the new position to RA and Dec, the information about the length of the new vector will be lost. If one converts these RA and Dec back to Cartesian coordinates, and add the E-term vector, then we would not obtain the original vector that we started with. Plot and explanation demonstrate how we should deal with removing and adding E-terms:

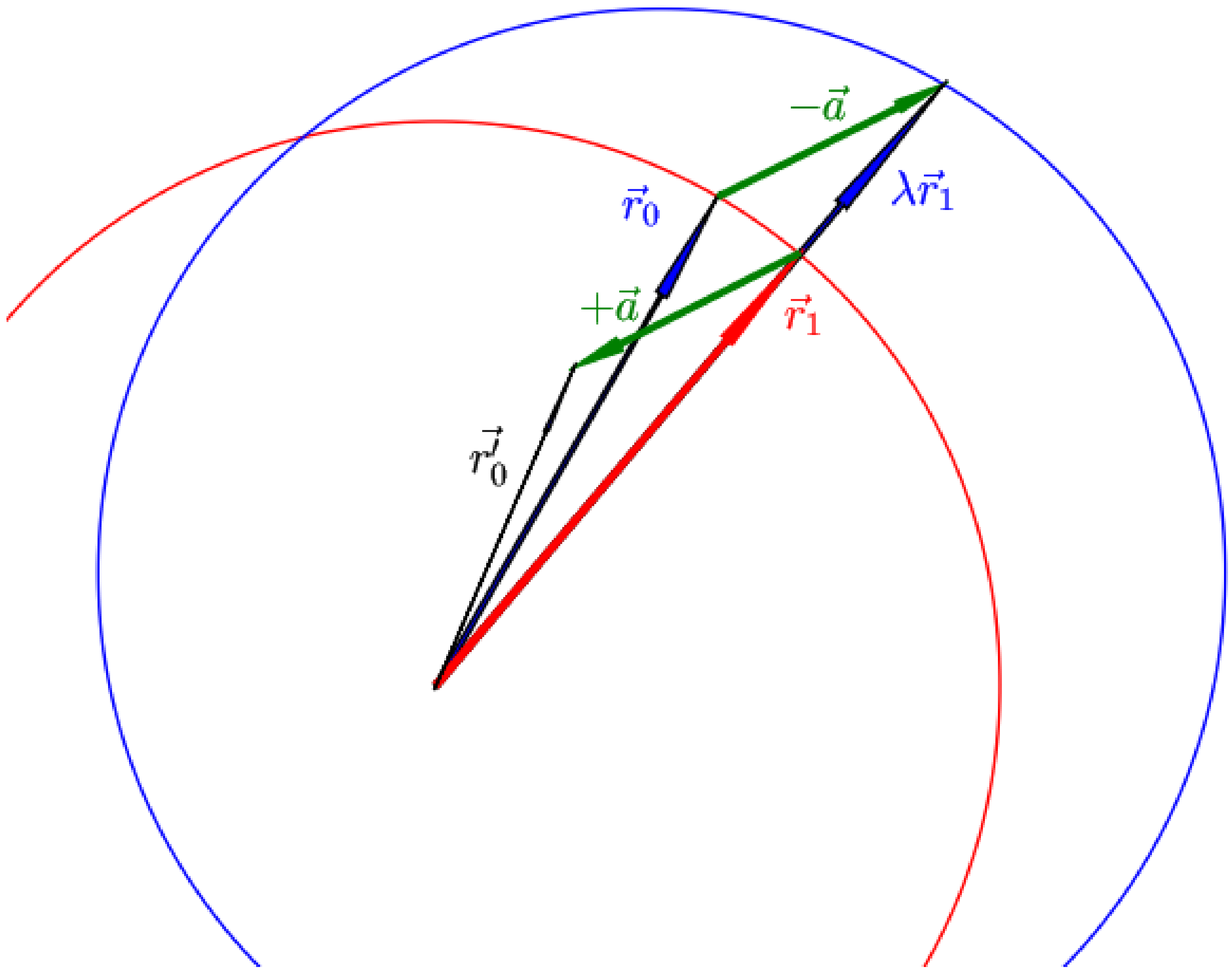


Fig.2 – E-term vectors.

In the figure one starts with a FK4 catalog position represented by vector $\vec{r}_0$. Removing the E-terms (represented by vector $\vec{a}$) results in vector $\lambda\vec{r}_1$. If vectors kept their length after converting them back to longitude and latitude then the inverse procedure would be as easy to add vector $\vec{a}$ to $\lambda\vec{r}_1$. Usually this is not the case, so for convenience we normalize $\lambda\vec{r}_1$ to get unit vector $\vec{r}_1$.

However, if we add vector $\vec{a}$ to $\vec{r}_1$ we end up with a vector $\vec{r'}_0$ which is not aligned with the original vector. To get it aligned, we have to stretch $\vec{r}_1$ again with some factor λ. We need an E-term adding procedure that applies to all unit vectors. It is straightforward to derive an expression for the wanted scaling factor λ:

Adding the E-term vector applying the conditions described above we write:

$$\lambda\vec{r}_1 + \vec{a} = \vec{r}_0 \tag{3.23}$$

And the conditions are:

$$||\vec{r}_1|| = ||\vec{r}_0|| = 1 \tag{3.24}$$

If we write this out in terms of the Cartesian coordinates x, y, z then with $\vec{r}_1 = (x_1, y_1, z_1)$, $\vec{r}_0 = (x_0, y_0, z_0)$, and $\vec{a} = a_x, a_y, a_z)$:

$$\begin{aligned}\lambda x_1 + a_x &= x_0\\ \lambda y_1 + a_y &= y_0\\ \lambda z_1 + a_z &= z_0\end{aligned} \tag{3.25}$$

And:

$$ {x_1}^2 + {y_1}^2 + {z_1}^2 = 1 \tag{3.26} $$

$$ {x_0}^2 + {y_0}^2 + {z_0}^2 = 1 \tag{3.27} $$

If we substitute the expressions for $\vec{r}_0$ (3.25) in this last equation (eq.27) then we obtain the simplified expression for λ:

$$ \lambda^2 + w\lambda + p = 0 \tag{3.28} $$

with:

$$ w = 2(a_x x_1 + a_y y_1 + a_z z_1) \tag{3.29} $$

$$ p = a_x^2 + a_y^2 + a_z^2 - 1 \tag{3.30} $$

We know that the length of the E-term vector a is much smaller than 1 so p is always less than 0. We also observe that only the positive solution for λ is the one we are searching for because a negative value represents a vector in opposite direction. Then we are left with an expression for the wanted λ:

$$ \lambda = (-w + \sqrt{w^2 - 4p})/2 \tag{3.31} $$

We started with known $\vec{r}_1$ and $\vec{a}$. With those we can calculate the wanted vector $\vec{r}_0$, which represents the catalog position.

3.1.4 Transformations between the reference systems FK4 and FK5

For conversions between FK4 and FK5 we follow the procedure of Murray *[Murray]*. Murray precesses from B1950 to J2000 using a precession matrix by Lieske (1979) and then applies the equinox correction and ends up with a transformation matrix *X(0)* and its rate of change per Julian century *X'(0)*.

If *F* is the ratio of Julian century to tropical century (1.000021359027778) and T is the time in Julian centuries from the epoch B1950, then Murray derives a transformation equation for a position and velocity in FK4:

$$ \begin{bmatrix} r \\ v \end{bmatrix} = \begin{bmatrix} X(0) + T\dot{X}(0) & TFX(0) \\ \dot{X}(0) & FX(0) \end{bmatrix} \begin{bmatrix} r_1 \\ v_1 \end{bmatrix} \tag{3.32} $$

Positions:

If the epoch of observation is *T* in Julian centuries counted from B1950 then from the previous equation we derive:

$$ r_{J2000} = X(0)(r_{B1950} + v_{B1950}FT) + T\dot{X}(0)r_{B1950} \tag{3.33} $$

Module `celestial` assumes that we have unknown or zero proper motions. We allow for fictitious proper motion in FK5, then we get the equation:

$$ r_{J2000} = r + v_{J2000}t = X(0)r_{B1950} + T\dot{X}(0)r_{B1950} \tag{3.34} $$

where *v* is the (fictitious) proper motion in FK5 and *t* is the time in Julian centuries form J2000. This is how the function `celestial.FK42FK5Matrix()` works for a given epoch of observation. In the output of the next interactive session, we show the results of varying the epoch of observation for a position R.A., Dec = (0,0):

```
>>> from kapteyn.celestial import *
>>> print sky2sky( (eq,'b1950',fk4), (eq,'j2000',fk5), 0,0)
[[ 0.640691    0.27840944]]
>>> print sky2sky( (eq,'b1950',fk4, 'J1970'), (eq,'j2000',fk5), 0,0)
[[ 0.64070422  0.27838524]]
>>> print sky2sky( (eq,'b1950',fk4, 'J1980'), (eq,'j2000',fk5), 0,0)
[[ 0.64071084  0.27837314]]
>>> print sky2sky( (eq,'b1950',fk4, 'J1990'), (eq,'j2000',fk5), 0,0)
[[ 0.64071745  0.27836105]]
```

The differences are a result of the fact that FK4 is slowly rotating with respect to the inertial FK5 system.

Velocities

The relation between velocities in the two systems is given also by the transformation equations:

$$v_{J2000} = \dot{X}(0)r_{B1950} + FX(0)v_{B1950} \tag{3.35}$$

Then:

$$v_{B1950} = F^{-1}X^{-1}(0)(v_{J2000} - \dot{X}(0)r_{B1950}) \tag{3.36}$$

Module `celestial` deals with positions from maps with objects for which we expect that the proper motion in FK5 is zero (e.g. extra-galactic sources). Then the expression for the fictitious proper motion in FK4 is:

$$v_{B1950} = -F^{-1}X^{-1}(0)\dot{X}(0)r_{B1950} \tag{3.37}$$

If we substitute this in equation (3.33) then we have the simple relation:

$$r_{J2000} = X(0)r_{B1950} \tag{3.38}$$

To summarize the possible transformations between FK4 and FK5:

Note: If you allow non zero proper motion in FK5 you should specify an epoch for the date that the mean place was correct and apply the formula:

$$r_{J2000} = X(0)r_{B1950} + T\dot{X}(0)r_{B1950} \tag{3.39}$$

If you are sure that the your position corresponds to an object with zero proper motion in FK5 then the epoch of observation is not necessary and one applies the formula:

$$r_{J2000} = X(0)r_{B1950} \tag{3.40}$$

Note that the matrix *X(0)* is not a rotation matrix because the inverse matrix is not equal to the transpose. Therefore the transformation matrix for conversion of FK5 to FK4 is the inverse of *X(0)*.

Murray's method has been described as controversial (e.g. see Soma (1990), *[Soma]*), but Poppe (2005) *[Poppe]* shows that the differences in results between the methods of Standish, Aoki and Murray are less than 5 mas.

3.1.5 Radio maps

Much of the B1950 data that users at the Kapteyn Astronomical Institute transform to FK5 J2000, is data from the Westerbork Synthesis Radio Telescope (WSRT). For this telesope we retrieved some information about the correction program that was used to transform apparent places to mean places. Apparent coordinates change during an observing run, due to:

- Refraction
- Precession
- Nutation
- **Aberration**
    1. Annual aberration
    2. Diurnal aberration
    3. Secular aberration (unknown and not significant)
    4. Planetary aberration (unknown and not significant)
- Proper motion (not significant)
- Parallax (not significant)

If X_t are the coordinates of a source at a time *t*, X_e are the coordinates at epoch e and:

- *N* is the rotation matrix describing the nutation
- *P* is the rotation matrix describing the precession
- *A* is the vector describing the annual aberration
- *D* is the vector describing the diurnal aberration

then the following relations apply:

$$X_t = N.P.X_e + A + D \tag{3.41}$$

$$X_e = P^{-1}.N^{-1}.(X_t - A - D) \tag{3.42}$$

The vector describing the correction for annual aberration is the vector

$$A = \begin{bmatrix} -D \\ +C \\ C\tan(\epsilon) \end{bmatrix} \tag{3.43}$$

C and D are the so called Besselian Day Numbers (tabulated in the *Astronomical Almanac*) that correct for annual aberration. Early interferometers like the WSRT produced images with greater resolution than obtainable in the optical at that time and in the construction of the radio maps a correction for the elliptical terms was included. So these maps are in fact FK4-NO-E (which is FITS terminology for a FK4 map where the E-terms are removed). For precession and transformations for these maps, no E-terms need to be removed.

Regretably many of FITS files with B1950 data do not include a value for the `RADESYS` keyword and one should try to find out how the coordinate system of these radio maps were constructed to be sure whether E-terms are included or not.

Calabretta (2002) writes:

FK4 coordinates are not strictly spherical since they include a contribution from the elliptic terms of aberration, the so-called E-terms which amount to a maximum of 343 milliarcsec. Strictly speaking, therefore, a map obtained from, say, a radio synthesis telescope, should be regarded as FK4-NO-E unless it has been appropriately resampled or a distortion correction provided. In common usage, however, 'CRVAL' for such maps is usually given in FK4 coordinates. In doing so, the E-terms are effectively corrected to first order only.

Contradictory to this, we understand that it depends on how a radio map is sampled whether E-terms are included or not. Also not clear is the reason why one would resample a map in FK4-NO-E. Finally, assuming that usually

CRVAL is given in FK4 coordinates seems a bit dangerous. For example for a transformation to Galactic coordinates the E-terms in the FK4 map are removed while it possibly didn't contain E-terms at all.

With a primary focus on maps with extragalactic objects we have to be sure that galaxy positions given in FK4 coordinates can reliably be converted to FK5 positions. Cotton (1999) *[Cotton]* presents a list with galaxy positions in B1950 and J2000 coordinates from the Uppsala General catalog (UGC). For the J2000 positions they used Digitized Sky Survey (DSS) images to measure accurate positions of all included UGC galaxies. The positions are accurate to the arcsecond level. For a sample of these galaxies we converted the B1950 positions and compared these to the listed J2000 positions in the article. The numbers were accurate to 10 mas, well within the positional errors given in the listing (which are > 1 arcsecond).

For VLBI data we need another kind of test for accuracy. Aoki (1986) *[Aoki2]* compares the transformation results of the B1950 position of 3C273B

$\alpha = 12^h 26^m 33.246^s$, $\delta = 2°19'42''.4238$, epoch of observation: 1978.62) to J2000 of several authors. He concludes that different authors use different methods and get different results. Aoki's method differs a few tens mas from the J2000 (VLBI radio sources based) catalog position where RA=12h29m6.6997 (no value for Dec was given). We also noticed that the highest accuracy is obtained if one uses the epoch of observation. Aoki's result differs 1.6 mas from the catalog value. The results of celestial.py differ only 0.01 mas in RA compared to Aoki's results.

Hering (1998) *[Hering]* gives a short description of a procedure in which a B1950 position of a radio source is converted to a J2000 position using the position in B1950 and J2000 of a calibrator source assuming that the angular distance between these sources is the same in both reference systems. An example of Radio star HIP 66257 was added:

```
Calibrator: 1404+286 (FK4)
            alpha(B1950) = 14h 04m 45.613s  delta(B1950) =  28d 41' 29.22''
            1404+286 (ICRF)
            alpha(J2000) = 14h 07m 00.3944s delta(J2000) =  28d 27' 14.690''


Radio star: HIP 66257 = HR 5110, Julian epoch of observation: t0 = 1982.3619
            alpha(B1950) = 13h 32m 32.145s  delta(B1950) = 37d 26' 16.18''
            Updated radio star position with respect to the calibrator given
            in the ICRF:
            alpha(J2000) = 13h 34m 45.6817s  delta(J2000) = 37d 10' 56.854''

Celestial:  FK4 to ICRS
            alpha(J2000) = 13h 34m 45.6862s  delta(J2000) = 37d 10' 56.790''
```

We assumed that the original article has an error in the value of alpha(J2000) of 2 seconds. This must be a typing mistake because the procedure described in that article is based on Aoki (1986) and when we apply this method to the data we are close to the corrected position. A difference of 2000 mas cannot be explained otherwise. The difference between `celestial` and the updated radio star position using the method of constant angular distances, is:

$(\Delta\alpha, \Delta\delta) = (68\,mas, 64\,mas)$

Hering claims a difference between the updated radio star position and that obtained by (his) formal transformation from B1950 to J2000 of:

$(\Delta\alpha\cos(\delta), \Delta\delta) = (20\,mas, 7\,mas)$

It is not straightforward to draw conclusions from these comparisons because the formal transformation is not described in detail. The results of `celestial` are close to Aoki's so if Hering's method is based on Aoki's, we expect comparable differences, which is, for unknown reasons, not the case.

3.1.6 Galactic Coordinates

According to Blaauw et al. (1959), the original definitions for the Galactic sky systems are:

- The new north galactic pole lies in the direction:

$$(\alpha,\ \delta) = (12h49m\ ,\ 27^\circ.4) = (192^\circ.25,\ 27^\circ.4) \tag{3.44}$$

  (equinox 1950.0)

- The new zero of longitude is the great semicircle originating at the new north galactic pole at the position angle theta = 123 degrees with respect to the equatorial pole for 1950.0.
- Longitude increases from 0 degrees to 360 degrees. The sense is such that, on the galactic equator increasing galactic longitude corresponds to increasing Right Ascension. Latitude increases from -90 degrees through 0 degrees to +90 degrees at the new galactic pole.

Given the RA and Dec of the galactic pole, and using the Euler angles scheme Rz(a3).Ry(a2).Rz(a1), we first rotate the spin vector of the XY plane about an angle a1 = 192.25 degrees and then rotate the spin vector in the XZ plane (i.e. around the Y axis) with an angle a2 = 90-27.4 degrees to point it in the right declination.

Now think of a circle with the galactic pole as its center. The radius is equal to the distance between this center and the equatorial pole. The zero point in longitude now is opposite to this pole We need to rotate along this circle (i.e. a rotation around the new Z-axis) in a way that the angle between the zero point and the equatorial pole is equal to 123 degrees. So first we need to compensate for the 180 degrees of the current zero longitude, opposite to the pole. Then we need to rotate about an angle 123 degrees but in a way that increasing galactic longitude corresponds to increasing Right Ascension which is opposite to the standard rotation of this circle (note that we rotated the original X axis about 192.25 degrees which flips the direction of rotation when viewed from (0,0,0). The last rotation angle therefore is a3 = 180-123 degrees. The composed rotation matrix is calculated with:

$$R = R_z(180 - 123)R_y(90 - 27.4)R_z(192.25) \tag{3.45}$$

The numbers are the same as in Slalib's 'ge50.f' and in the matrix of eq. (32) of Murray (1989) *[Murray]*. The numbers in the composed rotation matrix to convert equatorial FK4 mean places to IAU1958 galactic coordinates, calculated with `celestial` are:

```
>>> from kapteyn.celestial import *
>>> import numpy
>>> m = skymatrix((eq,'b1950',fk4), gal)[0]
>>> print numpy.array2string(numpy.array(m), precision=12)
[-0.066988739415 -0.872755765852 -0.483538914632]
[ 0.492728466075 -0.45034695802   0.744584633283]
[-0.867600811151 -0.188374601723  0.460199784784]
```

Compare this to the numbers in SLALIB's ge50.f:

```
[-0.066988739415D0,-0.872755765852D0,-0.483538914632D0]
[+0.492728466075D0,-0.450346958020D0,+0.744584633283D0]
[-0.867600811151D0,-0.188374601723D0,+0.460199784784D0]
```

And to Murray's matrix:

```
[-0.066988739 -0.872755766 -0.483538915]
[ 0.492728466 -0.450346958  0.744584633]
[-0.867600811 -0.188374602  0.460199785]
```

FK4 catalog positions are not corrected for the elliptic terms of aberration. One should remove these terms first before transforming to galactic coordinates.

Transformations from FK5 J2000 to Galactic coordinates

Galactic coordinates are defined using features in the FK4 system. If these axes could be identified with catalog objects one should first remove the E-terms. Then the rotation to FK5 results in a new system of axes that are non-orthogonal because the E-term correction depends on the position in the sky. Therefore we consider the position of the galactic pole as a FK4 position corrected for E-terms (i.e. FK4-NO-E) and apply transformations only to FK4 positions corrected for E-terms (i.e. we transform from and to the FK4-NO-E system). According to Blaauw (private communication 2008) the precision in the determination of the position of the galactic pole did not justify the effort to bother about E-terms. So if we define the position of the Galactic pole to be in FK4-NO-E coordinates, we don't change the original definition.

Using this definition of the galactic pole one can find the position of this pole in J2000 coordinates by direct transformations from FK4-NO-E to FK5 and define a rotation matrix for a transformation from FK5 to Galactic coordinates. But to preserve as accurate as possible the galactic coordinates of objects observed in the FK4 system one should first apply the transformation from FK5 to FK4-NO-E and then apply the transformation from FK4-NO-E to Galactic coordinates.

We identify the same problem with the conversion from FK4 to Ecliptic coordinates and using the same logic, we only define transformation between FK4-NO-E and the Ecliptic system.

Note: Transformations involving FK4 coordinates are defined in the FK4-NO-E system. For FK4 catalog positions, this means that one needs to remove the E-terms first before any transformation is applied.

The composed rotation matrix for *FK5 to Galactic* coordinates from `celestial` is:

```
>>> m = skymatrix((eq,'j2000',fk5), gal)[0]
[-0.054875539396 -0.873437104728 -0.48383499177 ]
[ 0.494109453628 -0.444829594298  0.7469822487  ]
[-0.867666135683 -0.198076389613  0.455983794521]
```

which is consistent with the transpose of the matrix in eq. 33 of Murray (1989) *[Murray]*.

```
[-0.054875539 -0.873437105 -0.483834992]
[ 0.494109454 -0.444829594  0.746982249]
[-0.867666136 -0.198076390  0.455983795]
```

And to SLALIB's galeq.f:

```
[-0.054875539726D0,-0.873437108010D0,-0.483834985808D0]
[+0.494109453312D0,-0.444829589425D0,+0.746982251810D0]
[-0.867666135858D0,-0.198076386122D0,+0.455983795705D0]
```

The SLALIB version also first applies the standard FK4 to FK5 transformation, for zero proper motion in FK5 and then applies the transformation from FK4 to galactic coordinates.

Galactic coordinates are given in (l,b) (also known as l^{II}, b^{II}.

3.1.7 Supergalactic coordinates

The Supergalactic equator is conceptually defined by the plane of the local (Virgo-Hydra-Centaurus) supercluster, and the origin of supergalactic longitude is at the intersection of the supergalactic and galactic planes. According to Corwin (1994) the northern supergalactic pole is at l=47 degrees.37, b=6 degrees.32 (IAU1958 galactic coordinates) and the supergalactic longitude (sgl) is zero at l=137 degrees.37.

For the rotation matrix we chose the scheme R = Rz.Ry.Rz

Then first we rotate about 47 degrees.37 along the Z-axis followed by a rotation about 90-6.32 degrees along the Y-axis to set the supergalactic pole to the right declination. The new plane intersects the old one at two positions. One of them is l=137 degrees.37, b=0 degrees (in galactic coordinates). If we want this position to be sgl=0 we have to rotate this plane along the new Z-axis about an angle of 90 degrees. So the composed rotation matrix is:

$$R = R_z(90)R_y(90 - 6.32)R_z(47.37) \tag{3.46}$$

The numbers in the matrix that converts from *galactic to supergalactic* coordinates are:

```
[ -7.357425748044e-01   6.772612964139e-01  -6.085819597056e-17]
[ -7.455377836523e-02  -8.099147130698e-02   9.939225903998e-01]
[  6.731453021092e-01   7.312711658170e-01   1.100812622248e-01]
```

Compare this to the numbers in SLALIB's galsup.f

```
[-0.735742574804D0,+0.677261296414D0,+0.000000000000D0]
[-0.074553778365D0,-0.080991471307D0,+0.993922590400D0]
[+0.673145302109D0,+0.731271165817D0,+0.110081262225D0]
```

Supergalactic coordinates are given in (sgl, sgb).

3.1.8 Ecliptic coordinates

The ecliptic coordinate system is a celestial coordinate system that uses the ecliptic for its fundamental plane. The coordinate system is suitable for objects with small deviations from the ecliptic (e.g. planets).

The latitude is measured positive towards the north. The longitude is measured eastwards and has an angle between 0 degrees and 360 degrees, the same direction as in the equatorial system. The intersection of the ecliptic and the equatorial plane at Right Ascension zero (vernal equinox) is the origin of the ecliptic longitude. In converting equatorial coordinates to ecliptic coordinates, only one angle is involved. This angle is known as the obliquity of the ecliptic. The value for the obliquity depends on epoch. In fact, the ecliptic is the rotation of the equatorial plane along the X-axis and the rotation angle is the obliquity:

$$R = R_x(\epsilon) \tag{3.47}$$

Like equatorial coordinates, ecliptic coordinates are subject to precession and a value for the equinox is required to specify positions. Ecliptic coordinates therefore are also related to the reference systems (FK4, FK5 and ICRS) known to the equatorial sky system. ICRS positions are defined without an equinox value so the corresponding ecliptic coordinates should be fixed also (to J2000). However we apply a frame bias to ICRS to get a position in the dynamical j2000 system and allow for precession of this system.

According to the IAU 1980 theory of nutation an estimation of the obliquity can be made with the expression:

$$\epsilon = 23°26'21''.448 - 46''8150T - 0''00059T^2 + 0''.001813T^3 \tag{3.48}$$

The expression is from Lieske (1977). T is the time, measured in Julian centuries of 36525 days, since 'basic' epoch J2000.

The IAU2000 expression is:

$$\epsilon = \epsilon_0 - 46''836769T - 0''0001831T^2 + 0''.0.00200340T^3 - 0.000000576T^4 - 0.0000000434T^5 \tag{3.49}$$

and ϵ_0 = 84381.406 arcseconds.

Ecliptic coordinates are given in (λ, β)

3.1.9 ICRS, Dynamical J2000 and FK5

ICRS

In 1991 a new celestial reference system was proposed by the IAU. It was adopted by the IAU General Assembly of 1997 as the *The International Celestial Reference System* (ICRS) It officially replaced the FK5 system on January 1, 1998 and is now in common use for positional astronomy. The ICRS is based on a number of extra-galactic radio sources. The system is centered on the barycenter of the Solar System. It doesn't depend on any rotating pole and its origin is close to the mean equinox at J2000. This origin is called the *Celestial Ephemeris Origin* (CEO). The realization of the reference frame is provided by a sample of suitable stars from the Hipparcos catalog. Coordinates in this frame are Right Ascension and Declination. There is no associated equinox but when dealing with proper motions one should associate an epoch of observation.

The dynamical J2000 system

The dynamical J2000 system is based on the real mean position of the equinox at J2000. We follow the inertial definition (i.e. inertial ecliptic versus rotating ecliptic) which has an offset of 93.66 mas with respect to the rotating definition. So the offsets of the right ascensions in the next sections are in correspondence with the inertial definition.

Offsets

The tilt and offset of the FK5 equator with respect to the ICRS is:

- η_0 = -19.9 mas (ICRS pole offset)
- ξ_0 = 9.1 mas (ICRS pole offset)
- $d\alpha_0$ = -22.9 (the ICRS right ascension offset)

To transform vectors from ICRS to FK5 at J2000 one uses the rotation matrix:

$$R = R_x(-\eta_0)R_y(\xi_0)R_z(d\alpha_0) \tag{3.50}$$

The rotation matrix is:

```
>>> print skymatrix(fk5,icrs)
[[  1.00000000e+00   1.11022337e-07   4.41180343e-08]
[ -1.11022333e-07   1.00000000e+00  -9.64779274e-08]
[ -4.41180450e-08   9.64779225e-08   1.00000000e+00]]
```

Observations showed that the J2000 mean pole is not at ICRS position (0,0) but at position (-0”.016617, -0”.0068192) and that the J2000 mean equinox was positioned 0”.0146 west of the ICRS meridian (IAU-SOFA 2007).

With the angles:

- η_0 = -6.8192 mas
- ξ_0 = -16.617 mas
- $d\alpha_0$ = -14.6 mas

we construct the rotation matrix:

```
>>> print skymatrix(j2000,icrs)
[[  1.00000000e+00,   7.07827948e-08,  -8.05614917e-08]
[ -7.07827974e-08,   1.00000000e+00,  -3.30604088e-08]
[  8.05614894e-08,   3.30604145e-08,   1.00000000e+00]]
```

which is similar to the rotation matrix described in eq. 8 of Hilton (2004). In this article the rotation matrix from J2000 to the ICRS is discussed. The authors follow the rotation scheme R_z R_x R_z, but we follow the scheme in Kaplan (2005) which is equivalent but is a more straightforward translation of the pole offsets and the origin.

So if we define a position (x,y,z) = (0,0,1) in the J2000 system, then we expect in the ICRS system two values that are approximately the pole offsets. Indeed this is the case as is shown in the next code fragment. Note that the offsets in x and y can be converted to angles because these angles are very small $dx \approx R.d\xi$:

```
>>> import numpy as n
>>> from kapteyn.celestial import *
>>> xyz = n.asmatrix( (0,0,1.0), 'd' ).T
>>> xyz2 = dotrans(skymatrix(j2000,icrs), xyz)
>>> print xyz2
[[ -8.05614894e-08],
[ -3.30604145e-08],
[  1.00000000e+00]]
>>> print xyz2[0,0]*(180/n.pi)*3600000
-16.6170004827
```

```
>>> print xyz2[1,0]*(180/n.pi)*3600000
-6.8191988238
```

3.1.10 Composing other transformations

With the basic transformation described above we can compose all other transformations by composing a new rotation matrix. In the next figure we show all the transformations that `celestial` supports.

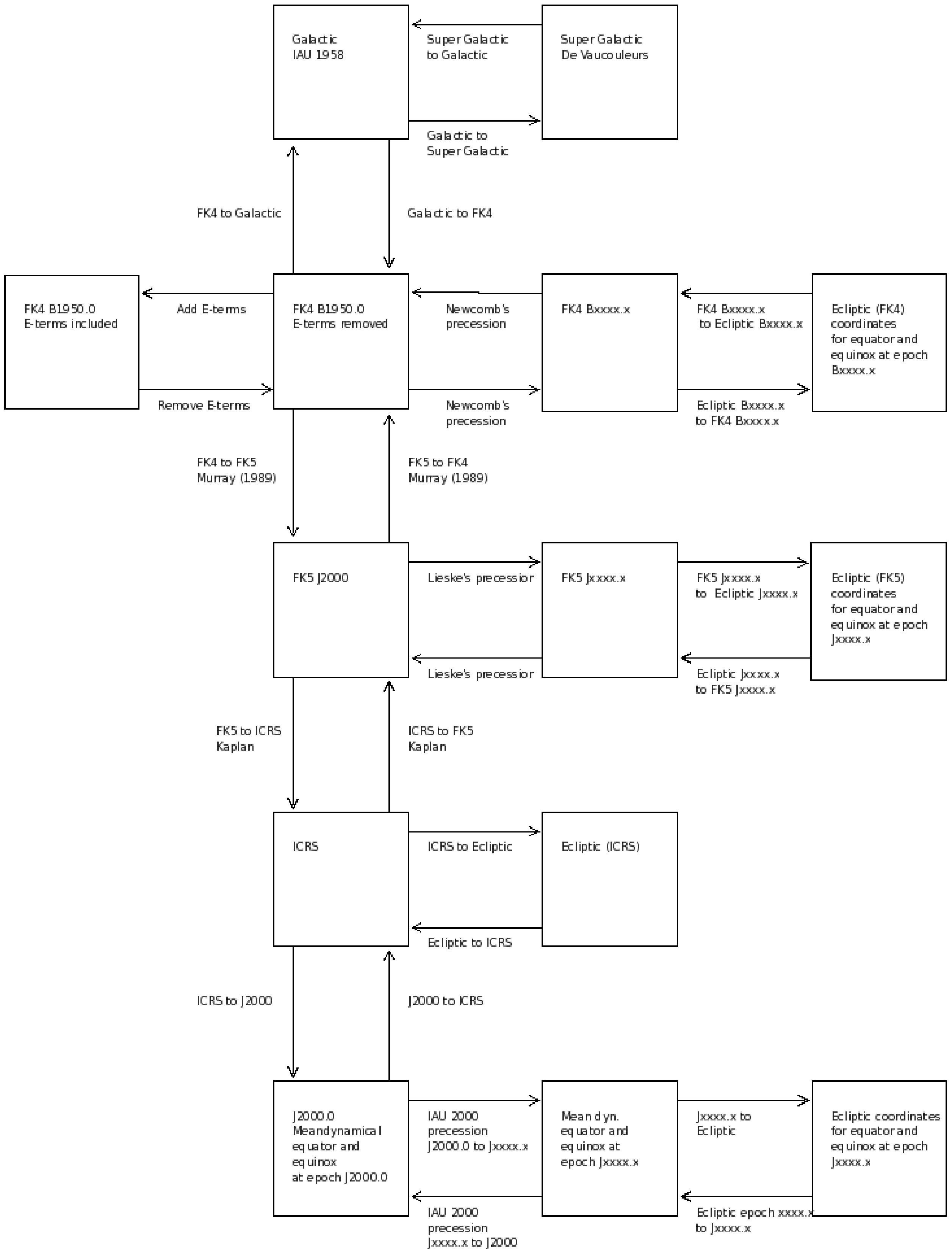

Galactic IAU 1958
Super Galactic to Galactic
Super Galactic De Vaucouleurs
Galactic to Super Galactic
FK4 to Galactic
Galactic to FK4
FK4 B1950.0 E-terms included
Add E-terms
Remove E-terms
FK4 B1950.0 E-terms removed
Newcomb's precession
Newcomb's precession
FK4 Bxxxx.x
FK4 Bxxxx.x to Ecliptic Bxxxx.x
Ecliptic Bxxxx.x to FK4 Bxxxx.x
Ecliptic (FK4) coordinates for equator and equinox at epoch Bxxxx.x
FK4 to FK5 Murray (1989)
FK5 to FK4 Murray (1989)
FK5 J2000
Lieske's precession
Lieske's precession
FK5 Jxxxx.x
FK5 Jxxxx.x to Ecliptic Jxxxx.x
Ecliptic Jxxxx.x to FK5 Jxxxx.x
Ecliptic (FK5) coordinates for equator and equinox at epoch Jxxxx.x
FK5 to ICRS Kaplan
ICRS to FK5 Kaplan
ICRS
ICRS to Ecliptic
Ecliptic to ICRS
Ecliptic (ICRS)
ICRS to J2000
J2000 to ICRS
J2000.0 Meandynamical equator and equinox at epoch J2000.0
IAU 2000 precession J2000.0 to Jxxxx.x
IAU 2000 precession Jxxxx.x to J2000
Mean dyn. equator and equinox at epoch Jxxxx.x
Jxxxx.x to Ecliptic
Ecliptic epoch xxxx.x to Jxxxx.x
Ecliptic coordinates for equator and equinox at epoch Jxxxx.x

Fig.3 – Schematic overview of all possible transformations in celestial.

Note: The figure illustrates that for each transformation from FK4 and for each transformation to FK4, the E-terms are processed. This has been motivated for transformations between FK4 and FK5. For galactic coordinates we assume that the galactic pole was given in FK4-NO-E. The difference between the position in FK4 and FK4-NO-E is much smaller than the errors in the position of the galactic pole which is the motivation to use FK4-NO-E as the starting point (which means that we use improved mean places anyhow).

3.1.11 Defaults in relation to FITS

In FITS the type of world coordinate system (celestial system) is specified in keyword `CTYPE` For equatorial systems, the reference system in FITS is given with keyword `RADESYS`

The epoch of the mean equator and equinox is given with FITS keyword `EPOCH` (deprecated) or `EQUINOX` For ecliptic and equatorial systems, some rules are set:

- Epoch is sometimes used to refer to the time of observation so if both keywords are given, `EQUINOX` takes preference
- `EQUINOX` also applies to ecliptic coordinates
- For `RADESYS` values of FK4 and FK4-NO-E any stated equinox is Besselian
- `RADESYS` also applies to ecliptic coordinates
- If for FK4 neither `EQUINOX` or `EPOCH` are given, a default of 1950 will be taken
- For `RADESYS` value of FK5 the stated equinox is Julian
- If only `EQUINOX` is given and not `RADESYS` then the reference system defaults to FK4 if `EQUINOX` < 1984 and it defaults to FK5 if `EQUINOX` > 1984
- If both `RADESYS` and `EQUINOX` are absent then `RADESYS` defaults to ICRS
- A date of observation is given in keywords `MJD-OBS` or `DATE-OBS`

3.1.12 Glossary

Most of the definitions are from the reference below or from various web sources.

Besselian to Julian epoch B = 1900.0 + (Julian date - 2415020.31352) / 365.242198781 (according to IAU).

Epoch Instant of time.

Epoch B1950 Mean orientation of the earth's equator and ecliptic at the beginning of the year 1950 (1950,01,01, 12h). It is tied to the sky by star coordinates in the FK4 catalog.

Epoch J2000 Mean orientation of the earth's equator and ecliptic at the beginning of the year 2000 (2000,01,01, 12h). It is tied to the sky by star coordinates in the FK5 catalog.

Equinox An equinox is a moment in time when the center of the Sun can be observed to be directly above the Earth's equator. At an equinox, the Sun is at one of two opposite points on the celestial sphere where the celestial equator (i.e. declination 0) and the ecliptic intersect (Vernal and autumnal points).

Equinox of the date Means that the equinox is the same as the epoch.

Ecliptic The Ecliptic is the plane of the Earth's orbit, projected onto the sky. Ecliptic coordinates are a spherical coordinate system referred to the ecliptic and expressed in terms of "Ecliptic latitude" and "Ecliptic longitude". By implication, Ecliptic coordinates are also referred to a specific "Equinox"

Equator: true equator of a date Is the plane perpendicular to direction of the celestial pole.

Equator: mean equator of a date Is deduced from the true equator of the date by a transformation given by the nutation theory.

Fiducial point A point on a scale used for reference or comparison purposes. If the plane of the ecliptic and the plane of the equator is used as lanes of reference, the equinox is used as fiducial point.

FK4 FundamentalKatalog 4. The 4th fundamental catalog. The FK4 is an equatorial coordinate system (coordinate system linked to the Earth) based on its B1950 position. The units used for time specification is the Besselian Year (Fricke & Kopff 1963). See also: Fricke, W., & Kopff, A. 1963, Fourth Fundamental Katalog (FK4), Veroeff. Astron. Rechen-Inst. Heidelb. No. 10. The FK4 system is not inertial. There is a small but significant rotation relative to distant objects. So, besides the equinox, an epoch is required to specify when the mean place was correct.

FK5 FundamentalKatalog 5. Based on J2000 positions. The units used for time specification is the Julian year.

Galactic coordinates The galactic coordinate system is a spherical reference system on the sky where the origin is close to the apparent center of the Milky Way, and the "equator" is aligned to the galactic plane.

ICRS Current astrometric observations and measurements should now be made in the International Celestial Reference System (ICRS) The best optical realization of the ICRF currently available is the Hipparcos catalogue. The Hipparcos frame is aligned to the ICRF to within about 0.5 mas For reasons of continuity and convenience, the orientation of the new ICRS frame was set up to have a close match to FK5 J2000. See for example: http://aa.usno.navy.mil/faq/docs/ICRS_doc.php

mas milliarcsecond (10^{-3} arcsec).

Obliquity (of the Ecliptic) This term refers to the angle the plane of the equator makes with the plane of the Earth's orbit.

Precession The orientation of the Earth's axis is slowly but continuously changing, tracing out a conical shape in a cycle of approximately 25,765 years This change is caused by the gravitational forces (mainly Sun and Moon).

Reference frame A reference frame consists of a set of identifiable fiducial points on the sky along with their coordinates, which serves as the practical realization of a reference system.

Reference system A reference system is the complete specification of how a celestial coordinate system is to be formed. It defines the origin and fundamental planes (or axes) of the coordinate system. It also specifies all of the constants, models, and algorithms used to transform between observable quantities and reference data that conform to the system.

3.1.13 References

3.2 Background information spectral translations

3.2.1 Introduction

This background information has been written for two reasons. First we wanted to get some understanding of the conversions between spectral quantities and second, we wanted to have some knowledge about legacy FITS headers (of which there must be a lot) where applying the conversions of WCSLIB in the context of module `wcs` without modifications will give wrong results.

Warning: One needs to be aware of the fact that WCSLIB converts between frequencies and velocities in the same reference system while in legacy FITS headers it is common to give a topocentric reference frequency and a reference velocity in a different reference system.

3.2.2 Alternate headers for a spectral line example

In “Representations of spectral coordinates in FITS” (*[Ref3]*), section 10.1 deals with an example of a VLA spectral line cube which is regularly sampled in frequency (CTYPE3=’FREQ’). The section describes how one can define alternative FITS headers to deal with different velocity definitions. We want to examine this exercise in more detail than provided in the article to illustrate how a FITS header can be modified and serve as an alternate header.

The topocentric spectral properties in the FITS header from the paper are:

```
CTYPE3= 'FREQ'
CRVAL3=  1.37835117405e9
CDELT3=  9.765625e4
CRPIX3=  32
CUNIT3= 'Hz'
RESTFRQ= 1.420405752e+9
SPECSYS='TOPOCENT'
```

Note: For a pixel coordinate N, reference pixel N_{ref} with reference world coordinate W_{ref} and a step size in world coordinates ΔW, the world coordinate W is calculated with:

$$W(N) = W_{ref} + (N - N_{ref}) \times \Delta W \tag{3.51}$$

If *CTYPE* contains a code for a non linear conversion algorithm (as in CTYPE=’VOPT-F2W’) then this relation cannot be applied.

As stated in the note above, code for a conversion algorithm is important. The statements can be verified with the following script:

```
#!/usr/bin/env python
from kapteyn import wcs

Z0 = 9120000                # Barycentric optical reference velocity
dZ0 = -2.1882651e+4         # Increment in barycentric optical velocity
N = 32                      # Pixel coordinate of reference pixel

header = {  'NAXIS'   :  1,
            'RESTWAV' :  0.211061140507,    # [m]
            'CTYPE1'  : 'VOPT',
            'CRVAL1'  :  Z0,                # [m/s]
            'CDELT1'  :  dZ0,               # [m/s]
            'CRPIX1'  :  N,
            'CUNIT1'  : 'm/s'
         }
spec = wcs.Projection(header)
print "From VOPT: Pixel, velocity wcs, velocity linear (%s)" % spec.units
pixels = range(30,35)
Vwcs = spec.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000.0, (Z0 + (p-N)*dZ0)/1000.0

header = {  'NAXIS'   :  1,
            'CNAME1'  : 'Barycentric optical velocity',
            'RESTWAV' :  0.211061140507,    # [m]
            'CTYPE1'  : 'VOPT-F2W',
            'CRVAL1'  :  Z0,                # [m/s]
            'CDELT1'  :  dZ0,               # [m/s]
            'CRPIX1'  :  N,
             'CUNIT1'   : 'm/s'
          }
spec = wcs.Projection(header)
print "From VOPT-F2W: Pixel, velocity wcs, velocity linear (%s)" % spec.units
pixels = range(30,35)
Vwcs = spec.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000.0, (Z0 + (p-N)*dZ0)/1000.0

# Output:
#
# From VOPT: Pixel, velocity wcs, velocity linear (m/s)
# Conversion is linear; no differences
# 30 9163.765302 9163.765302
# 31 9141.882651 9141.882651
# 32 9120.0 9120.0
# 33 9098.117349 9098.117349
# 34 9076.234698 9076.234698
# From VOPT-F2W: Pixel, velocity wcs, velocity linear (m/s)
# Conversion is not linear
# 30 9163.77150335 9163.765302
# 31 9141.88420123 9141.882651
# 32 9120.0 9120.0
# 33 9098.11889901 9098.117349
# 34 9076.24089759 9076.234698
```

Relation optical velocity and barycentric/lsrk reference frequency

Let's start to find the alternate header information for the header from article in *[Ref3]* . The extra information about the velocity there is that we have an optical barycentric velocity of 9120 km/s (as required by an observer) stored as an alternate FITS keyword CRVAL3Z.:

```
CTYPE3Z= 'VOPT-F2W'
CRVAL3Z=  9.120e+6       / [m/s]
```

The relation between frequency and optical velocity requires a rest frequency (RESTFRQ=). The relation is:

$$Z = c\left(\frac{\lambda-\lambda_0}{\lambda_0}\right) = c\left(\frac{\nu_0-\nu}{\nu}\right) \tag{3.52}$$

We adopted variable Z for velocities following the optical definition. The header tells us that equal steps in pixel coordinates are equal steps in frequency and the formula above shows that these steps in terms of optical velocity depends on the frequency in a non-linear way. Therefore we set the conversion algorithm to **F2W** which indicates that there is a non linear conversion from frequency to wavelength (optical velocities are associated with wavelength, see *[Ref3]* .). Note that we can use wildcards for the non linear conversion algorithm, so *CTYPE3Z='VOPT-???'* is also allowed in our programs.

We can rewrite equation 1 into:

$$\nu = \frac{\nu_0}{(1+Z/c)} \tag{3.53}$$

If we enter the numbers we get a **barycentric** HI reference frequency:

$$\nu_b = \frac{1.420405752\times10^9}{(1+9120000/299792458.0)} = 1378471216.43\,Hz \tag{3.54}$$

and we have part of a new alternate header:

```
CTYPE3F= 'FREQ'
CRVAL3F= 1.37847121643e+9 / [Hz]
```

So given an optical velocity in a reference system (in our case the barycentric system), we can calculate which barycentric frequency we can use as a reference frequency. For a conversion between a barycentric frequency and a barycentric velocity we also need to know what the barycentric frequency increment is.

Barycentric/lsrk frequency increments

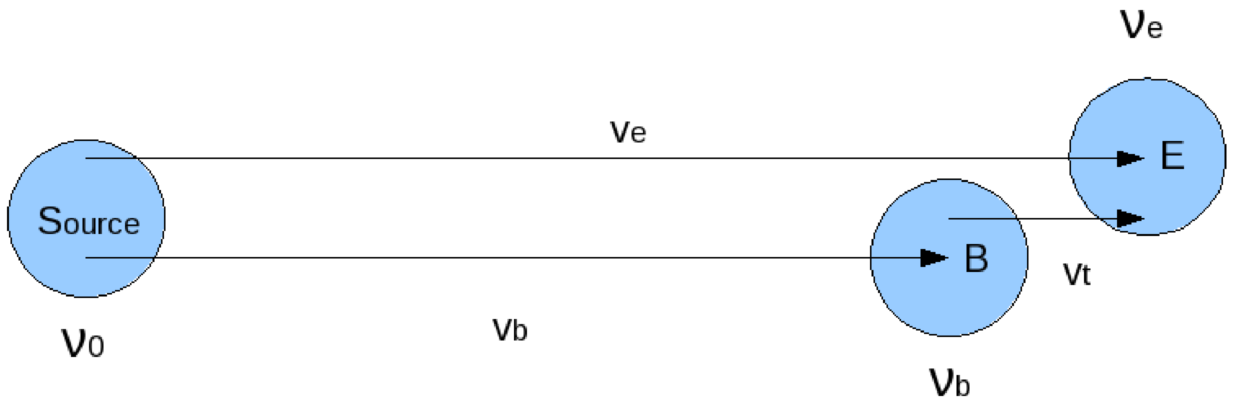


fig.1 Overview of velocities and frequencies of barycenter (B) and Earth (E) w.r.t. source. The arrows represent velocities. The object and the Earth are moving. The longest arrow represents the (relativistic) addition of two velocities

Let's use index *b* for variables bound to the barycentric system and *e* for the topocentric system. This frequency, ν_b =1.37847121643 GHz is greater than the reference frequency ν_e at the observatory (FITS keyword *CRVAL3*= 1.37835117405 GHz).

The difference between frequencies in the topocentric and barycentric system is caused by the difference between the velocities of reference frames B and E at the time of observation.

This velocity is a *true* velocity. It is called the *topocentric correction*.

Let's try to find an expression for this topocentric correction in terms of frequencies. The relation between a true velocity and a shift in frequency is given by the formula

$$\nu = \nu_0\sqrt{\frac{1-v/c}{1+v/c}} = \nu_0\sqrt{\frac{c-v}{c+v}} = \nu_0\frac{c-v}{\sqrt{c^2-v^2}} \tag{3.55}$$

If we want to express the apparent radial velocity in terms of frequencies, then this can be written as:

$$v = c\,\frac{\nu_0^2-\nu^2}{\nu_0^2+\nu^2} \tag{3.56}$$

For the apparent radial velocities v_b and v_e we have:

$$v_b = c\,\frac{\nu_0^2-\nu_b^2}{\nu_0^2+\nu_b^2} = 299792458.0\,\frac{1420405752.0^2-1378471216.43^2}{1420405752.0^2+1378471216.43^2} = 8981342.29811\ m/s \tag{3.57}$$

and:

$$v_e = c\,\frac{\nu_0^2-\nu_e^2}{\nu_0^2+\nu_e^2} = 299792458.0\,\frac{1420405752.0^2-1378351174.05^2}{1420405752.0^2+1378351174.05^2} = 9007426.97201\ m/s \tag{3.58}$$

The relativistic addition of velocities in fig. 1. requires:

$$v_e = \frac{v_b + v_t}{1 + \frac{v_b v_t}{c^2}} \tag{3.59}$$

which gives the topocentric correction as:

$$v_t = \frac{v_e - v_b}{1 - \frac{v_b v_e}{c^2}} \tag{3.60}$$

With the numbers inserted we find:

$$v_t = \frac{9007426.97201 - 8981342.29811}{1 - \frac{8981342.29811 \times 9007426.97201}{299792458.0^2}} = 26108.1743997\ m/s \tag{3.61}$$

If the FITS header has keywords with the position of the source, the time of observation and the location of the observatory then one can calculate the topocentric correction by hand. This information was needed at the observatory to set a frequency for a given barycentric velocity. However many FITS files do not have enough information to calculate the topocentric correction. Also it is not needed if one knows the shifted frequencies ν_e and ν_b , then we can calculate the topocentric velocity without calculating the apparent radial velocities. This can be shown if we insert the expressions for velocities v_e and v_b in the expression for v_t . Then after some rearranging one finds:

$$v_t = c\,\frac{\nu_b^2 - \nu_e^2}{\nu_b^2 + \nu_e^2} \tag{3.62}$$

and with the numbers:

$$v_t = 299792458.0\,\frac{1378471216.43^2 - 1378351174.05^2}{1378471216.43^2 + 1378351174.05^2} = 26108.1743998\ m/s \tag{3.63}$$

which is consistent with (3.61).

```
VELOSYSZ=26108    / [m/s]
```

With a given topocentric correction and the reference frequency in the barycenter we can reconstruct the reference frequency at the observatory with (3.62) written as:

$$\nu_e = \nu_b \sqrt{\frac{c - v_t}{c + v_t}} \tag{3.64}$$

Note: 1) It is important to realize that the reference frequency at E is smaller than the reference frequency at B because w.r.t. the source E moves faster than B. So if there is a change in the velocity of the source, the frequencies in B and E will change, but the topocentric correction keeps the same value and therefore the relation between the frequencies ν_e and ν_b remains the same (eq. (3.64)).

Note: 2) If we forget about the source and we have an *event on E* with a certain frequency then an *observer* in barycenter *B* will observe a *lower* frequency. This is because on the line that connects the source and B, the observatory at E moves away from B which decreases the remote frequency.

So if we change a frequency on E by tuning the receiver at the observatory at frequency $\nu_e + \Delta\nu_e$, then the observer at B would observe a smaller frequency $\nu_b + \Delta\nu_b$. The amount of the decrease is related to the topocentric correction as follows:

$$\nu_b + \Delta\nu_b = (\nu_e + \Delta\nu_e)\sqrt{\frac{c - v_t}{c + v_t}} \tag{3.65}$$

and therefore we can write for the frequency bandwidth in B:

$$\Delta\nu_b = \Delta\nu_e \sqrt{\frac{c - v_t}{c + v_t}} \tag{3.66}$$

At first it seems that this contradicts eq. (3.64) (where the indices seem to be swapped), but this is not true because we changed the frame of the observer from Earth to the barycenter. The event was in E and it is observed in B.

$$\Delta\nu_b = 97656.25\,\frac{\sqrt{299792458.0 - 26108.1743998}}{\sqrt{299792458.0 + 26108.1743998}} = 97647.745732\ Hz \tag{3.67}$$

The increment in frequency therefore becomes 97.64775 kHz:

```
CDELT3F=  9.764775e+4 / [Hz]
```

So if we change CRVAL1 and CDELT1 in our demonstration script to the barycentric values, we get the barycentric optical convention velocities for the pixels. As a check we listed the script and the value for pixel 32 which is exactly 9120 (km/s):

```
#!/usr/bin/env python
from kapteyn import wcs
header  = { 'NAXIS'  : 1,
            'CTYPE1' : 'FREQ',
            'CRVAL1' : 1378471216.4292786,
            'CRPIX1' : 32,
            'CUNIT1' : 'Hz',
            'CDELT1' : 97647.745732,
            'RESTFRQ': 1.420405752e+9
         }
spec = wcs.Projection(header).spectra('VOPT-F2W')
pixels = range(30,35)
Vwcs = spec.toworld1d(pixels)
print "Pixel, velocity (%s)" % spec.units
for p,v in zip(pixels, Vwcs):
   print p, v/1000.0

print "Pixel at velocity 9120 km/s: ", spec.topixel1d(9120000)
# Output
# Pixel, velocity (m/s)
# 30 9163.77150423
# 31 9141.88420167
# 32 9120.0
# 33 9098.11889856
# 34 9076.2408967
# Pixel at velocity 9120 km/s:  32.0
```

Note: A closure test is added with method *topixel1d()*

Note: In the previous two sections we started with a topocentric frequency and a topocentric frequency increment and derived values for a barycentric frequency and a barycentric frequency increment. These values can be used to set an alternate header (barycentric frequency system 'F') for which we can convert between frequency and optical velocity. For GIPSY legacy headers these steps are used to convert between topocentric frequencies and velocities in another reference system, See *A recipe for modification of Nmap/GIPSY FITS data*

Increment in barycentric/lsrk optical velocity

The optical velocity was given by:

$$Z = c\,\bigl(\frac{\nu_0-\nu}{\nu}\bigr) = c\,\bigl(\frac{\nu_0}{\nu}-1\bigr) \tag{3.68}$$

Its derivative is:

$$\frac{dZ}{d\nu} = \frac{-c\nu_0}{\nu^2} \tag{3.69}$$

But for ν we have the expression:

$$\nu = \frac{\nu_0}{(1+\frac{Z}{c})} \tag{3.70}$$

so we end up with:

$$dZ = \frac{-c}{\nu_0}\,\bigl(1+\frac{Z}{c}\bigr)^2\,d\nu \tag{3.71}$$

With $d\nu = \Delta\nu_b$ and the given barycentric velocity Z_b = 9120000 m/s, this gives an increment in optical velocity of:

$$dZ_b = \frac{-299792458.0}{1420405752.0}\,\bigl(1+\frac{9120000.0}{299792458.0}\bigr)^2\;97647.745732 = -21882.651\;m/s \tag{3.72}$$

With these values we explained some other alternate header keywords in the basic spectral-line example:

```
CDELT3Z= -2.1882651e+4  / [m/s]
SPECSYSZ= 'BARYCENT'    / Velocities w.r.t. barycenter
SSYSOBSZ= 'TOPOCENT'    / Observation was made from the 'TOPOCENT' frame
```

Barycentric/lsrk radio velocity

For radio velocities one needs to apply the definition:

$$V_{radio} = V = c\,\bigl(\frac{\nu_0-\nu}{\nu_0}\bigr) \tag{3.73}$$

and for the shifted frequency we derive from this equation:

$$\nu = \nu_0\,\bigl(1-\frac{V}{c}\bigr) \tag{3.74}$$

and the spectral translation code becomes: *proj.spectra('VRAD')*

In the next code example we demonstrate for a barycentric radio velocity V = 8850.750904 km/s how to calculate the barycentric velocities at arbitrary pixels. This velocity is derived from the optical example in a way that shifted frequency and topocentric correction are the same. One can use the formula

$$\frac{V_b}{Z_b} = \frac{\nu_b}{\nu_0} \tag{3.75}$$

to find the value of $V_b = 1.37847121643 * 9120/1.420405752 = 8850.750904$ km/s (with the frequencies in GHz and the velocity in km/s). In a next section we will derive this value in another way; see (3.76) and (3.77)

```python
#!/usr/bin/env python
from kapteyn import wcs
import numpy as n

c = 299792458.0       # Speed of light (m/s)
f = 1.37835117405e9   # Topocentric reference frequency (Hz)
df = 9.765625e4       # Topocentric frequency increment (Hz)
f0 = 1.420405752e+9   # Rest frequency (Hz)
V = 8850750.904       # Barycentric radio velocity (m/s)

fb = f0*(1-V/c)
print "Barycentric freq.: ", fb
v = c * ((fb*fb-f*f)/(fb*fb+f*f))
print "VELOSYSR= Topocentric correction:", v, "m/s"
dfb = df*(c-v)/n.sqrt(c*c-v*v)
print "CDELT3F= Delta in frequency in the barycentric frame eq.4): ", dfb

header = { 'NAXIS'  : 1,
           'CTYPE1' : 'FREQ',
           'CRVAL1' : fb,
           'CRPIX1' : 32,
           'CUNIT1' : 'Hz',
           'CDELT1' : dfb,
           'RESTFRQ': 1.420405752e+9
      }
line = wcs.Projection(header).spectra('VRAD')
pixels =  range(30,35)
Vwcs = line.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000

# Output:
# Barycentric freq.:  1378471216.43
# VELOSYSR= Topocentric correction: 26108.1745986 m/s
# CDELT3F= Delta in frequency in the barycentric frame eq.4):  97647.745732
#
# Output Radio velocities (km/s)
# 30 8891.97019316
# 31 8871.36054858
# 32 8850.750904
# 33 8830.14125942
# 34 8809.53161484
```

Frequency to Radio velocity

From the definition of radio velocity:

$$V = c\,\bigl(\frac{\nu_0-\nu}{\nu_0}\bigr) \tag{3.76}$$

we can find a radio velocity that corresponds to the value of the optical velocity. This (barycentric) optical velocity (9120 Km/s) caused a shift of the rest frequency. The new frequency became $\nu_b = 1.37847122\times 10^9 Hz$. If we insert this number in the equation above we find:

$$V_b = c\,\bigl(\frac{1420405752.0-1378471216.43}{1420405752.0}\bigr) = 8850750.90419\,m/s \tag{3.77}$$

The formula for a direct conversion from optical to radio velocity can be derived by inserting the formula for the frequency shift corresponding to optical velocity, into the expression for the radio velocity:

$$V = c\,(1 - \frac{1}{1+\frac{Z}{c}}) \tag{3.78}$$

With eq. (3.76) it is easy to find the increment of the velocity if the increment in frequency at the reference frequency is given:

$$dV = \frac{-c}{\nu_0}\,d\nu \tag{3.79}$$

Note that this increment in frequency is the **increment in the barycentric system**!

Inserting the numbers with $d\nu = \Delta\nu_b$ we find:

$$dV_b = \frac{-299792458.0}{1420405752.0} \times 97647.7457312 = -20609.644582\; m/s \tag{3.80}$$

This gives us another two values for the alternate header keywords:

```
CTYPE3R= 'VRAD'
CRVAL3R= 8.85075090419e+6   / [m/s]
CDELT3R= -2.0609645e+4      / [m/s]
```

Note that *CTYPE3R= 'VRAD'* indicates that the conversion between frequency and radio velocity is linear.

The next script shows how we can use these new header values to get a list of radio velocities as function of pixel. We commented out the rest frequency. Its value is not necessary because we can rewrite the formulas for the velocity in terms of ν/ν_0 and $\Delta\nu/\nu_0$

```
#!/usr/bin/env python
from kapteyn import wcs
header = { 'NAXIS'  : 1,
           'CTYPE1' : 'VRAD',
           'CRVAL1' : 8850750.904193053,
           'CRPIX1' : 32,
           'CUNIT1' : 'm/s',
           'CDELT1' : -20609.644582145629,
#          'RESTFRQ': 1.420405752e+9
         }
line = wcs.Projection(header)
pixels =  range(30,35)
Vwcs = line.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000
#
# Output barycentric radio velocity in km/s:
# 30 8891.97019336
# 31 8871.36054878
# 32 8850.75090419
# 33 8830.14125961
# 34 8809.53161503
```

Alternatively use the spectral translation method *spectra()* with the values of the barycentric frequency and frequency increment as follows to get (exactly) the same output:

```
#!/usr/bin/env python
from kapteyn import wcs
header = { 'NAXIS'  : 1,
           'CTYPE1' : 'FREQ',
            'CRVAL1' : 1378471216.4292786,
            'CRPIX1' : 32,
            'CUNIT1' : 'Hz',
            'CDELT1' : 97647.745732,
            'RESTFRQ': 1.420405752e+9
          }
line = wcs.Projection(header).spectra('VRAD')
pixels =  range(30,35)
Vwcs = line.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000
#
# Output barycentric radio velocity in km/s:
# 30 8891.97019336
# 31 8871.36054878
# 32 8850.75090419
# 33 8830.14125961
# 34 8809.53161503
```

Frequency to Apparent radial velocity

As written before, the relation between a true velocity and a shifted frequency is:

$$v = c\,\frac{\nu_0^2-\nu^2}{\nu_0^2+\nu^2} \tag{3.81}$$

Observed from the barycenter the source has an apparent radial velocity:

$$v_b = 299792458.0\,\frac{1420405752.0^2-1378471216.42927^2}{1420405752.0^2+1378471216.42927^2} = 8981342.29811\ m/s \tag{3.82}$$

```
CTYPE3V= 'VELO-F2V'
CRVAL3V= 8.98134229811e+6 / [m/s]
```

Note that *CTYPE3V= 'VELO-F2V'* indicates that we derived these velocities from a system in which the frequency is linear with the pixel value.

For the increment of the apparent radial velocity we need to find the derivative of eq. (3.56)

$$\frac{dv}{d\nu} = c(\nu_0^2-\nu^2)\frac{d}{d\nu}\left(\nu_0^2+\nu^2\right)^{-1} + c\left(\nu_0^2+\nu^2\right)^{-1}\frac{d}{d\nu}\left(\nu_0^2-\nu^2\right) \tag{3.83}$$

This works out as:

$$dv = \frac{-4c\nu\nu_0^2}{\left(\nu_0^2+\nu^2\right)^2}\,d\nu \tag{3.84}$$

and with the appropriate numbers inserted for $d\nu = \Delta\nu_b$

and $\nu = \nu_b$:

$$dv_b = \frac{-4\times 299792458.0\times 1378471216.4292786\times 1420405752.0^2}{\left(1420405752.0^2+1378471216.4292786^2\right)^2}\,97647.745732 = -21217.55136 \tag{3.85}$$

which reveals the value of another keyword from the header in the article's example:

```
CDELT3V= -2.1217551e+4  / [m/s]
```

Sometimes you might encounter an alternative formula that doesn't list the frequency. It uses eq. (3.55) to express the frequency in terms of the apparent radial velocity and the rest frequency.

$$\nu = \nu_0 \sqrt{\frac{1 - v/c}{1 + v/c}} \tag{3.86}$$

If you insert this into:

$$dv = \frac{-4c\nu\nu_0^2}{(\nu_0^2 + \nu^2)^2}\, d\nu \tag{3.87}$$

then after some rearrangements you end up with the expression:

$$dv = \frac{-c}{\nu_0} \sqrt{(1 - \frac{v}{c})}\, (1 + \frac{v}{c})^{\frac{3}{2}}\, d\nu \tag{3.88}$$

If you insert v = 8981342.29811 (m/s) in this expression you will get exactly the same apparent radial velocity increment (-2.1217551e+4 m/s).

We found an apparent radial velocity and calculated the increment for this radial velocity. With a short script and a minimal header we demonstrate how to use WCSLIB to get an apparent radial velocity for an arbitrary pixel:

```python
#!/usr/bin/env python
from kapteyn import wcs

header = { 'NAXIS'  : 1,
           'CTYPE1' : 'VELO-F2V',
           'CRVAL1' : 8981342.2981121931,
           'CRPIX1' : 32,
           'CUNIT1' : 'm/s',
           'CDELT1' : -21217.5513673598,
           'RESTFRQ': 1.420405752e+9
         }
line = wcs.Projection(header)
pixels = range(30,35)
Vwcs = line.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000
# Output:
# 30 9023.78022672
# 31 9002.56055595
# 32 8981.34229811
# 33 8960.12545322
# 34 8938.9100213
```

How can this work? From eq. (3.86) and eq. (3.87) it is obvious that WCSLIB can calculate the reference frequency from the reference apparent radial velocity. For this reference frequency and the increment in apparent radial velocity it can calculate the increment in frequency at this reference frequency. Then we have all the information to use eq. (3.86) to calculate radial velocities for different frequencies (i.e. different pixels). Note that the step in frequency is linear and the step in radial velocity is **not** (which explains the extension 'F2V' in the CTYPE keyword).

Next script and header is an alternative to get exactly the same results. The header lists the barycentric frequency and frequency increment. We need a spectral translation with method *spectra()* to tell WCSLIB to calculate apparent radial velocities:

```
#!/usr/bin/env python
from kapteyn import wcs
header  = { 'NAXIS'  : 1,
            'CTYPE1' : 'FREQ',
            'CRVAL1' : 1378471216.4292786,
            'CRPIX1' : 32,
            'CUNIT1' : 'Hz',
            'CDELT1' : 97647.745732,
            'RESTFRQ': 1.420405752e+9
         }
line = wcs.Projection(header).spectra('VELO-F2V')
pixels = range(30,35)
Vwcs = line.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000
# Output:
# 30 9023.78022672
# 31 9002.56055595
# 32 8981.34229811
# 33 8960.12545322
# 34 8938.9100213
```

Frequency to Wavelength

The rest wavelength is given by the relation:

$$\lambda_0 = \frac{c}{\nu_0} \tag{3.89}$$

Inserting the right numbers we find:

$$\lambda_0 = \frac{299792458.0}{1420405752.0} = 0.211061140507\ m \tag{3.90}$$

For the barycentric wavelength we need to insert the barycentric frequency.

$$\lambda = \frac{299792458.0}{1378471216.43} = 0.217481841062\ m \tag{3.91}$$

The increment in wavelength as function of the increment in (barycentric) frequency is:

$$d\lambda = \frac{-c}{\nu^2} d\nu \tag{3.92}$$

With the right numbers:

$$d\lambda = \frac{-299792458.0}{1378471216.43^2}\ 97647.745732 = -1.54059158176 \times 10^{-5}\ m \tag{3.93}$$

This gives us the alternate header keywords:

```
RESTWAVZ= 0.211061140507  / [m]
```

```
CTYPE3W= 'WAVE-F2W'
CRVAL3W=  0.217481841062  / [m]
CDELT3W= -1.5405916e-05   / [m]
CUNIT3W=  'm'
RESTWAVW= 0.211061140507  / [m]
```

Note that CTYPE indicates that there is a non linear conversion from frequency to wavelength.

From the standard definition of optical velocity:

$$Z = c\,\frac{\lambda - \lambda_0}{\lambda_0} \tag{3.94}$$

it follows that the increment in optical velocity as function of increment of wavelength is given by:

$$dZ = \frac{c}{\lambda_0}\,d\lambda \tag{3.95}$$

Then with the numbers we find:

$$dZ_b = \frac{299792458.0}{0.211061140507} \times -1.54059158176 \times 10^{-5} = -21882.6514422\;m/s \tag{3.96}$$

which is the increment in optical velocity earlier given for CDELT3Z.

This is one of the possible conversions between wavelength and velocity. Others are listed in scs.pdf table 3 of E.W. Greisen et al. page 750.

Conclusions

- Note that the inertial system is set by a (FITS) header using a special keyword (e.g. VELREF=) or it is coded in the CTYPEn keyword. It doesn't change anything in the calculations above. Conversions between inertial reference systems is not possible because headers do (usually) not contain the relevant information to calculate the topocentric correction w.r.t. that system (one needs time of observation, position of observatory and position of the observed source).
- From a header with CTYPEn='FREQ' we can derive optical, radio and apparent radial velocities with method *spectra()*:
    - *proj = wcs.Projection(header).spectra('VOPT-F2W')*
    - *proj = wcs.Projection(header).spectra('VRAD')*
    - *proj = wcs.Projection(header).spectra('VELO-F2V')*

  This applies also to alternate axis descriptions. So if CTYPE1='VRAD' one can derive one of the other velocity definitions by adding the *spectra()* method with the appropriate argument.

  Here is an example:

```
#!/usr/bin/env python
from kapteyn import wcs
wcs.debug = True
header = { 'NAXIS'  : 1,
           'CTYPE1' : 'VRAD',
           'CRVAL1' : 8850750.904193053,
           'CRPIX1' : 32,
           'CUNIT1' : 'm/s',
           'CDELT1' : -20609.644582145629,
           'RESTFRQ': 1.420405752e+9
           }
line = wcs.Projection(header).spectra('VOPT-F2W')
pixels = range(30,35)
Vwcs = line.toworld1d(pixels)
for p,v in zip(pixels, Vwcs):
   print p, v/1000
# Output:
```

```
# Velocities in km/s converted from 'VRAD' to 'VOPT-F2W'
# 30 9163.77150423
# 31 9141.88420167
# 32 9120.0
# 33 9098.11889856
# 34 9076.2408967
```

Note that the rest frequency is required now.

Note also that we added statement *wcs.debug = True* to get some debug information from WCSLIB.

- Axis types 'FREQ-HEL' and 'FREQ-LSR' (AIPS definitions) are recognized by WCSLIB and are treated as 'FREQ'. No conversions are done. Internally the keyword *SPECSYS=* gets a value.

The complete alternate axis descriptions

In this section we summarize the alternate axis descriptions and we add a small script that proves that these descriptions are consistent:

```
CNAME=  'Topocentric Frequency. Basic header'
CTYPE3= 'FREQ'
CRVAL3=  1.37835117405e9
CDELT3=  9.765625e4
CRPIX3=  32
CUNIT3= 'Hz'
RESTFRQ= 1.420405752e+9
SPECSYS='TOPOCENT'

CNAME3Z= 'Barycentric optical velocity'
RESTWAVZ= 0.211061140507   / [m]
CTYPE3Z= 'VOPT-F2W'
CRVAL3Z=  9.120e+6         / [m/s]
CDELT3Z= -2.1882651e+4     / [m/s]
CRPIX3Z=  32
CUNIT3Z= 'm/s'
SPECSYSZ='BARYCENT'        / Velocities w.r.t. barycenter
SSYSOBSZ='TOPOCENT'        / Observation was made from the 'TOPOCENT' frame
VELOSYSZ= 26108            / [m/s]

CNAME3F= 'Barycentric frequency'
CTYPE3F= 'FREQ'
CRVAL3F=  1.37847121643e+9 / [Hz]
CDELT3F=  9.764775e+4      / [Hz]
CRPIX3F=  32
CUNIT3F= 'Hz'
RESTFRQF= 1.420405752e+9
SPECSYSF='BARYCENT'
SSYSOBSF='TOPOCENT'
VELOSYSF= 26108            / [m/s]

CNAME3R= 'Barycentric radio velocity'
CTYPE3R= 'VRAD'
CRVAL3R=  8.85075090419e+6 / [m/s]
CDELT3R= -2.0609645e+4     / [m/s]
CRPIX3R=  32
CUNIT3R= 'm/s'
RESTFRQR= 1.420405752e+9
SPECSYSR='BARYCENT'
```

```
SSYSOBSR='TOPOCENT'
VELOSYSR= 26108            / [m/s]

CNAME3V= 'Barycentric apparent radial velocity'
RESTFRQV= 1.420405752e+9   / [Hz]
CTYPE3V= 'VELO-F2V'
CRVAL3V=  8.98134229811e+6 / [m/s]
CDELT3V= -2.1217551e+4     / [m/s]
CRPIX3V=  32
CUNIT3V= 'm/s'
SPECSYSV='BARYCENT'
SSYSOBSV='TOPOCENT'
VELOSYSV= 26108            / [m/s]

CNAME3W= 'Barycentric wavelength'
CTYPE3W= 'WAVE-F2W'
CRVAL3W=  0.217481841062   / [m]
CDELT3W= -1.5405916e-05    / [m]
CRPIX3W=  32
CUNIT3W= 'm'
RESTWAVW=  0.211061140507  / [m]
SPECSYSW='BARYCENT'
SSYSOBSW='TOPOCENT'
VELOSYSW= 26108            / [m/s]
```

To check the validity and completeness of these alternate axis descriptions, we wrote a small script that loops over all the mnemonic letter codes in a header that is composed from the header fragments above. We only changed axisnumber 3 to 1. The output is the same within the boundaries of the given precision of the numbers. To change the axis description in a header we use the *alter* parameter when we create the projection object.

Parameter *alter* is an optional letter from 'A' through 'Z', indicating an alternative WCS axis description:

```python
#!/usr/bin/env python
from kapteyn import wcs
header = {  'NAXIS'    :  1,
            'CTYPE1'   : 'FREQ',
            'CRVAL1'   :  1378471216.4292786,
            'CRPIX1'   :  32,
            'CUNIT1'   : 'Hz',
            'CDELT1'   :  97647.745732,
            'RESTFRQ'  :  1.420405752e+9,
            'CNAME1Z'  : 'Barycentric optical velocity',
            'RESTWAVZ' :  0.211061140507,    # [m]
            'CTYPE1Z'  : 'VOPT-F2W',
            'CRVAL1Z'  :  9.120e+6,          # [m/s]
            'CDELT1Z'  : -2.1882651e+4,      # [m/s]
            'CRPIX1Z'  :  32,
            'CUNIT1Z'  : 'm/s',
            'SPECSYSZ' : 'BARYCENT',         # Velocities w.r.t. barycenter,
            'SSYSOBSZ' : 'TOPOCENT',         # Observation was made from the 'TOPOCENT' frame,
            'VELOSYSZ' :  26108,             # [m/s]
            'CNAME1F'  : 'Barycentric frequency',
            'CTYPE1F'  : 'FREQ',
            'CRVAL1F'  :  1.37847121643e+9, # [Hz]
            'CDELT1F'  :  9.764775e+4,       # [Hz]
            'CRPIX1F'  :  32,
            'CUNIT1F'  : 'Hz',
            'RESTFRQF' :  1.420405752e+9,
            'SPECSYSF' : 'BARYCENT',
            'SSYSOBSF' : 'TOPOCENT',
            'VELOSYSF' :  26108,             # [m/s]
            'CNAME1W'  : 'Barycentric wavelength',
            'CTYPE1W'  : 'WAVE-F2W',
            'CRVAL1W'  :  0.217481841062,    # [m]
            'CDELT1W'  : -1.5405916e-05,     # [m]
            'CRPIX1W'  :  32,
            'CUNIT1W'  : 'm',
            'RESTWAVW' :  0.211061140507,    # [m]
            'SPECSYSW' : 'BARYCENT',
            'SSYSOBSW' : 'TOPOCENT',
            'VELOSYSW' :  26108,             # [m/s]
            'CNAME1R'  : 'Barycentric radio velocity',
            'CTYPE1R'  : 'VRAD',
            'CRVAL1R'  :  8.85075090419e+6, # [m/s]
            'CDELT1R'  : -2.0609645e+4,      # [m/s]
            'CRPIX1R'  :  32,
            'CUNIT1R'  : 'm/s',
            'RESTFRQR' :  1.420405752e+9,
            'SPECSYSR' : 'BARYCENT',
            'SSYSOBSR' : 'TOPOCENT',
            'VELOSYSR' :  26108,             # [m/s]
            'CNAME1V'  : 'Barycentric apparent radial velocity',
            'CTYPE1V'  : 'VELO-F2V',
            'CRVAL1V'  :  8.98134229811e+6, # [m/s]
            'CDELT1V'  : -2.1217551e+4,      # [m/s]
            'CRPIX1V'  :  32,
            'CUNIT1V'  : 'm/s',
            'RESTFRQV' :  1.420405752e+9,    # [Hz]
            'SPECSYSV' : 'BARYCENT',
            'SSYSOBSV' : 'TOPOCENT',
            'VELOSYSV' :  26108              # [m/s]
         }

# Loop over all the alternative headers
for alt in ['F', 'Z', 'W', 'R', 'V']:
   spec = wcs.Projection(header, alter=alt).spectra('VOPT-F2W')
   pixels = range(30,35)
   Vwcs = spec.toworld1d(pixels)
   cname = header['CNAME1'+alt]            # Just a header text
   print "VOPT-F2W from %s" % (cname,)
   print "Pixel, velocity (%s)" % spec.units
   for p,v in zip(pixels, Vwcs):
      print p, v/1000.0
# Output
# VOPT-F2W from Barycentric frequency
# Pixel, velocity (m/s)
# 30 9163.77150598
# 31 9141.88420246
# 32 9119.99999984
# 33 9098.11889745
# 34 9076.24089463
# VOPT-F2W from Barycentric optical velocity
# Pixel, velocity (m/s)
# 30 9163.77150335
# 31 9141.88420123
# 32 9120.0
# 33 9098.11889901
# 34 9076.24089759
# VOPT-F2W from Barycentric wavelength
# Pixel, velocity (m/s)
# 30 9163.77150495
# 31 9141.88420213
# 32 9120.0000002
# 33 9098.1188985
# 34 9076.24089638
# VOPT-F2W from Barycentric radio velocity
# Pixel, velocity (m/s)
# 30 9163.77150512
# 31 9141.88420211
# 32 9120.0
# 33 9098.11889812
# 34 9076.24089581
# VOPT-F2W from Barycentric apparent radial velocity
# Pixel, velocity (m/s)
# 30 9163.77150347
# 31 9141.88420129
# 32 9120.0
# 33 9098.11889894
# 34 9076.24089746
```

3.2.3 Alternative conversions

Conversion between radio and optical velocity

In the next two sections we give some formula's that could be handy if you want to verify numbers. They are not used in WCSLIB.

With the definitions for radio and optical velocity it is easy to derive:

$$\frac{V}{Z} = \frac{\nu}{\nu_0} \tag{3.97}$$

This can be verified with:

- Z = 9120000.00000 m/s
- V = 8850750.90419 m/s
- ν_0 = 1420405752.00 Hz
- ν_b = 1378471216.43 Hz

Both ratios are equal to 1.030421045482.

Conversion between apparent radial velocity and optical/radio velocity

It is possible to find a relation between the true velocity and the optical velocity using eq. (3.53) and eq. (3.57). The apparent radial velocity can be written as:

$$\frac{v}{c} = \frac{\frac{\nu_0^2}{\nu^2} - 1}{\frac{\nu_0^2}{\nu^2} + 1} \tag{3.98}$$

The frequency shift for an optical velocity is:

$$\frac{\nu_0}{\nu} = \big(1 + \frac{Z}{c}\big) \tag{3.99}$$

Then:

$$\frac{v}{c} = \frac{(1+Z/c)^2-1}{(1+Z/c)^2+1} = \frac{Z^2+2cZ}{Z^2+2cZ+2c^2} \tag{3.100}$$

This equation is used in AIPS memo 27 *[Aipsmemo]* to relate an optical velocity to an apparent radial velocity. If we insert Z_b = 9120000 (m/s) then we find v_b = 8981342.29811 (m/s) as expected (eq. (3.57), (3.82))

For radio velocities we find in a similar way:

$$\frac{\nu_0}{\nu} = \frac{1}{(1-\frac{V}{c})} \tag{3.101}$$

which gives the relation between apparent radial velocity and radio velocity:

$$\frac{v}{c} = \frac{2cV-V^2}{V^2-2cV+2c^2} \tag{3.102}$$

If we substitute the calculated barycentric radio velocity V_b = 8850750.90419 (m/s) then one finds again: v_b = 8981342.29811 (m/s) (see also (eq. (3.57), (3.82)) Note that the last formula is equation 4 in AIPS memo 27 *[Aipsmemo]* Non-Linear Coordinate Systems in AIPS. However that formula lacks a minus sign in the nominator and therefore does not give a correct result.

3.2.4 Legacy headers

A recipe for modification of Nmap/GIPSY FITS data

For FITS headers produced by Nmap/GIPSY we don't have an increment in velocity available so we cannot use them as input for WCSLIB (otherwise we would treat them like the FELO axis recognized by AIPS). The Python interface to WCSLIB applies a conversion for these headers before they are processed by WCSLIB. From the previous steps we can summarize how the data in the Nmap/GIPSY FITS header is changed:

- The extension in CTYPEn is '-OHEL', '-OLSR', '-RHEL' or '-RLSR'
- The velocity is retrieved from FITS keyword VELR= (always in m/s) or DRVALn= (in units of DUNITn)
- Convert reference frequency to a frequency in Hz.
- Calculate the reference frequency in the barycentric system using eq. (3.53) if the velocity is optical and eq. (3.74) if the velocity is a radio velocity.
- Calculate the topocentric velocity using eq. (3.62)
- Convert frequency increment to an increment in Hz
- Calculate the increment in frequency in the selected reference system (HEL, LSR) using eq. (3.66).
- Change CRVALn and CDELTn to the barycentric values
- Change CTYPEn to 'FREQ'
- Create a projection object with spectral translation, e.g. **proj.spectra('VOPT-F2W')**

In the following script we show:

- the (invisible) conversion to the heliocentric system
- how to get the same output by applying the appropriate formulas
- the approximation that GIPSY uses

```
from kapteyn import wcs
from math import sqrt

header = { 'NAXIS'   : 1,
           'CTYPE1'  : 'FREQ-OHEL',
           'CRVAL1'  : 1.415418199417E+09,
           'CRPIX1'  : 32,
           'CUNIT1'  : 'HZ',
           'CDELT1'  : -7.812500000000E+04,
           'VELR'    : 1.050000000000E+06,
           'RESTFRQ': 0.14204057520000E+10
         }

f = crval = header['CRVAL1']
df = cdelt = header['CDELT1']
crpix = header['CRPIX1']
velr = header['VELR']
f0 =  header['RESTFRQ']
c = wcs.c    # Speed of light

print "VELR is the reference velocity given in the velocity frame"
print "coded in CTYPE (e.g. HEL, LSR)"
print "The velocity is either an optical or a radio velocity. This"
print "is also coded in CTYPE (e.g. 'O', 'R')"

proj = wcs.Projection(header)
spec = proj.spectra(ctype='VOPT-F2W')
pixrange = range(crpix-3, crpix+3)
V = spec.toworld1d(pixrange)
print "\n VOPT-F2W with spectral translation:"
for p, v in zip(pixrange, V):
  print "%4d %15f" % (p, v/1000)

print "\n VOPT calculated:"
fb = f0/(1.0+velr/c)
Vtopo = c * ((fb*fb-f*f)/(fb*fb+f*f))
dfb = df*(c-Vtopo)/sqrt(c*c-Vtopo*Vtopo)
for p in pixrange:
   f2 = fb + (p-crpix)*dfb
   Z = c * (f0/f2-1.0)
   print "%4d %15f" % (p, Z/1000.0)

print "\nOptical with native GIPSY formula, which is an approximation:"
fR = crval
dfR = cdelt
for p in pixrange:
   Zs = velr + c*f0*(1/(fR+(p-crpix)*dfR)-1/fR)
   print "%4d %15f" % (p, Zs/1000.0)
```

Output:

```
VELR is the reference velocity given in the velocity frame
coded in CTYPE (e.g. HEL, LSR)
The velocity is either an optical or a radio velocity. This
is also coded in CTYPE (e.g. 'O', 'R')

VOPT-F2W with spectral translation:
29     1000.194731
```

```
30      1016.794655
31      1033.396411
32      1050.000000
33      1066.605422
34      1083.212677

VOPT calculated:
29      1000.194731
30      1016.794655
31      1033.396411
32      1050.000000
33      1066.605422
34      1083.212677

VOPT with native GIPSY formula, which is an approximation:
29      1000.191559
30      1016.792540
31      1033.395354
32      1050.000000
33      1066.606480
34      1083.214793
```

The Python interface allows for an easy implementation for these special exceptions. Here is a script that uses this facility. The conversion here is triggered by the CTYPE extension **OHEL**. So as long this is unique to GIPSY spectral axes, you are safe to use it. Note that we converted the frequencies to optical, radio and apparent radial velocities. This is added value to the existing GIPSY implementation where these conversions are not possible. These WCSLIB conversions are explained in previous sections:

```python
#!/usr/bin/env python
from kapteyn import wcs
header = { 'NAXIS'  : 1,
           'CTYPE1' : 'FREQ-OHEL',
           'CRVAL1' : 1.37835117405e9,
           'CRPIX1' : 32,
           'CUNIT1' : 'Hz',
           'CDELT1' : 9.765625e4,
           'RESTFRQ': 1.420405752e+9,
           'DRVAL1' : 9120000.0,
#          'VELR'   : 9120000.0
           'DUNIT1' : 'm/s'
         }
proj = wcs.Projection(header)
pixels = range(30,35)

voptical = proj.spectra('VOPT-F2W')
Vwcs = voptical.toworld1d(pixels)
print "\nPixel, optical velocity (%s)" % voptical.units
for p,v in zip(pixels, Vwcs):
   print p, v/1000.0

vradio = proj.spectra('VRAD')
Vwcs = vradio.toworld1d(pixels)
print "\nPixel, radio velocity (%s)" % vradio.units
for p,v in zip(pixels, Vwcs):
   print p, v/1000.0

vradial = proj.spectra('VELO-F2V')
Vwcs = vradial.toworld1d(pixels)
```

```
print "\nPixel, apparent radial velocity (%s)" % vradial.units
for p,v in zip(pixels, Vwcs):
   print p, v/1000.0

# Output:
# Pixel, optical velocity (m/s)
# 30 9163.77150423
# 31 9141.88420167
# 32 9120.0
# 33 9098.11889856
# 34 9076.2408967
#
# Pixel, radio velocity (m/s)
# 30 8891.97019336
# 31 8871.36054878
# 32 8850.75090419
# 33 8830.14125961
# 34 8809.53161503
#
# Pixel, apparent radial velocity (m/s)
# 30 9023.78022672
# 31 9002.56055595
# 32 8981.34229811
# 33 8960.12545322
# 34 8938.9100213
```

Note: Note that changing DRVAL1 to VELR gives the same output. Both are recognized as keywords that store a velocity. The value in VELR should always be in m/s. Note also how we created different sub-projections (one for each type of velocity) from the same main projection. All these objects can coexist.

AIPS axis type FELO

Next script and output shows that with the optical reference velocity and the corresponding increment in velocity (CDELT3Z), we can get velocities without spectral translation. WCSLIB recognizes the axis type 'FELO' which is regularly gridded in frequency but expressed in velocity units in the optical convention. It is therefore not a surprise that the output is the same as the list with optical velocities derived from the spectral translation 'VOPT-F2W'.

We can prove this if we calculate the barycentric reference frequency and its increment. If *Zr* is the optical reference velocity then we find the barycentric reference frequency with:

$$\nu_r = \frac{\nu_0}{(1 + \frac{Z_r}{c})} \tag{3.103}$$

and from

$$dZ = \frac{-c}{\nu_0} \left(1 + \frac{Z_r}{c}\right)^2 d\nu \tag{3.104}$$

we derive:

$$d\nu = \frac{-\nu_0}{\left(1 + \frac{Z_r}{c}\right)^2} dZ \tag{3.105}$$

which we rewrite in:

$$d\nu = \frac{-\nu_0 c}{(c + Z_r)^2} dZ \tag{3.106}$$

So if we have a barycentric reference velocity and a barycentric velocity increment, then according to the formulas above it is easy to retrieve the values for the barycentric reference frequency and the barycentric frequency increment. The script below proves that indeed with these values the optical velocities are derived from a linear frequency axis and not from a linear velocity axis (see the last option in this script):

```
#!/usr/bin/env python
from kapteyn import wcs
from numpy import arange

header = { 'NAXIS'  : 1,
           'CTYPE1' : 'FELO-HEL',
           'CRVAL1' : 9120,
           'CRPIX1' : 32,
           'CUNIT1' : 'km/s',
           'CDELT1' : -21.882651442,
           'RESTFRQ': 1.420405752e+9
         }
crpix = header['CRPIX1']
pixrange = arange(crpix-2, crpix+3)
proj = wcs.Projection(header)
Z = proj.toworld1d(pixrange)
print "Pixel, velocity (km/s) with native header with FELO-HEL"
for p,v in zip(pixrange, Z):
   print p, v/1000.0

# Calculate the barycentric reference frequency and the frequency increment
f0 = header['RESTFRQ']
Zr = header['CRVAL1'] * 1000.0 # m/s
dZ = header['CDELT1'] * 1000.0 # m/s
c = wcs.c
fr = f0 / (1 + Zr/c)
print "\nCalculated a reference frequency: ", fr
df = -f0* dZ *c / ((c+Zr)*(c+Zr))
print "Calculated a frequency increment: ", df
Z = Zr + c*f0*(1/(fr+(pixrange-crpix)*df)-1/fr)
print "Pixel, velocity (km/s) with barycentric reference frequency and increment:"
for p,z in zip(pixrange, Z):
   print p, z/1000.0

# FELO-HEL is equivalent to VOPT-F2W
header['CTYPE1'] = 'VOPT-F2W'
proj = wcs.Projection(header)
Z = proj.toworld1d(pixrange)
print "\nPixel, velocity (km/s) with spectral translation VOPT-F2W"
for p,v in zip(pixrange, Z):
   print p, v/1000.0

# Now as a linear axis. Note that thoe output of toworld is in km/s
# and not in standard units (m/s) as for the recognized axis types
header['CTYPE1'] = 'FELO'
proj = wcs.Projection(header)
Z = proj.toworld1d(pixrange)
print "\nPixel, velocity (km/s) with CUNIT='FELO', which is unrecognized "
print "and therefore linear. This deviates from the previous output."
print "The second velocity is calculated manually."
for p,v in zip(pixrange, Z):
   print p, v, (Zr+(p-crpix)*dZ)/1000.0
```

Output:

```
Pixel, velocity (km/s) with native header with FELO-HEL
30 9163.77150423
31 9141.88420167
32 9120.0
33 9098.11889857
34 9076.24089671

Calculated a reference frequency:  1378471216.43
Calculated a frequency increment:  97647.7457311
Pixel, velocity (km/s) with barycentric reference frequency and increment:
30 9163.77150423
31 9141.88420167
32 9120.0
33 9098.11889857
34 9076.24089671

Pixel, velocity (km/s) with spectral translation VOPT-F2W
30 9163.77150423
31 9141.88420167
32 9120.0
33 9098.11889857
34 9076.24089671

Pixel, velocity (km/s) with CUNIT='FELO', which is unrecognized
and therefore linear. This deviates from the previous output.
The second velocity is calculated manually.
30 9163.76530288 9163.76530288
31 9141.88265144 9141.88265144
32 9120.0 9120.0
33 9098.11734856 9098.11734856
34 9076.23469712 9076.23469712
```

So in this script we demonstrated the use of a special velocity axis type which originates from a classic AIPS data FITS file. It is called ‘FELO’. WCSLIB (and not our Python interface) recognizes this type as an **optical velocity** and performs the necessary internal conversions as we can see in the source code:

```
if (strcmp(wcs->ctype[i], "FELO") == 0) {
   strcpy(wcs->ctype[i], "VOPT-F2W");
```

The source code also reveals that the extensions in CUNITn are translated into values for FITS keyword *SPECSYS*:

```
if (strcmp(scode, "-LSR") == 0) {
   strcpy(wcs->specsys, "LSRK");
} else if (strcmp(scode, "-HEL") == 0) {
   strcpy(wcs->specsys, "BARYCENT");
} else if (strcmp(scode, "-OBS") == 0) {
   strcpy(wcs->specsys, "TOPOCENT");
```

Conclusions

- The extension HEL or LSR after FELO in *CTYPE1* is not used in the calculations. But when you omit a valid extension the axis will be treated as a linear axis.
- In the example above one can replace *FELO-HEL* in *CTYPE1* by FITS standard *VOPT-F2W* showing that for WCSLIB *FELO-HEL* is in fact the same as *VOPT-F2W*.

AIPS axis type VELO

In this section we want to address the question what WCSLIB does if it encounters an AIPS VELO-XXX axis as in *CTYPE1='VELO-HEL'* or *'VELO-LSR'*. From the AIPS documentation we learn that VELO is regularly gridded in velocity (m/s) in the optical convention, unless overridden by use of the *VELREF* keyword. VELREF is an integer. From the documentation of WCSLIB we learn that for Classic Aips:

1. LSR kinematic, originally described simply as "LSR" without distinction between the kinematic and dynamic definitions.
2. Barycentric, originally described as "HEL" meaning heliocentric.
3. Topocentric, originally described as "OBS" meaning geocentric but widely interpreted as topocentric.

And for AIPS++ extensions to VELREF which are also recognized:

4. LSR dynamic.
5. Geocentric.
6. Source rest frame.
7. Galactocentric.

Note: From the WCSLIB documentation:

For an AIPS 'VELO' axis, a radio convention velocity is denoted by adding 256 to VELREF, otherwise an optical velocity is indicated (not applicable to 'FELO' axes). Unrecognized values of VELREF are simply ignored. VELREF takes precedence over CTYPEia in defining the Doppler frame.

Note: Only WCSLIB (versions >= 4.5.1) do recognize keyword *VELREF*.

We show the use of *VELREF* with the following script:

```python
#!/usr/bin/env python
from kapteyn import wcs
from math import sqrt

V0 = -.24300000000000E+06             # Radio vel in m/s
dV = 5000.0                           # Delta in m/s
f0 = 0.14204057583700e+10
c = wcs.c                             # Speed of light 299792458.0 m/s
crpix = 32
pixels = range(30,35)

header = { 'NAXIS'  : 1,
           'CTYPE1' : 'VELO-HEL',
           'VELREF' : 258,
           'CRVAL1' : V0,
           'CRPIX1' : crpix,
           'CUNIT1' : 'm/s',
           'CDELT1' : dV,
           'RESTFRQ': f0
        }

print "The velocity increment is constant and equal to %f (km/s): "\
      % (dV/1000.0)
```

```
proj = wcs.Projection(header)
print "Allowed spectral translations", proj.altspec
p2 = proj.spectra('VOPT-???')

print "\nT1. Radio velocity directly from header and optical velocity"
print "from spectral translation. VELO is a radio velocity here because"
print "VELREF > 256"

V = proj.toworld1d(pixels)
Z = p2.toworld1d(pixels)
print "Pixel Vradio in (km/s) and Voptical (km/s)"
for p,v,z in zip(pixels, V, Z):
  print "%4d %15f %15f" % (p, v/1000, z/1000)

print "\nT2. Now insert CTYPE1='VRAD' in the header and convert to VOPT-F2W"
print "with a spectral translation (Z1) and with a calculation (Z2)"
print "This should give the same results as in table T1."
header['CTYPE1'] = 'VRAD'
proj = wcs.Projection(header)
p2 = proj.spectra('VOPT-F2W')
Z0 = proj.toworld1d(pixels)
Z1 = p2.toworld1d(pixels)
print "\nWith CTYPE='RAD' and spec.trans 'VOPT-F2W': Pixel , Vrad, Z1 (km/s), Z2 (km/s)"
for p,z0,z1 in zip(pixels, Z0, Z1):
  V = V0 + (p-crpix)*dV
  nu_r = f0* (1-V/c)
  Z2 = c*((f0-nu_r)/nu_r)
  print p, z0/1000, z1/1000, Z2/1000

print "\nT3. We set CTYPE1 to VELO-HEL and VELREF to 2 (Helio) and "
print "derive optical and radio velocities from it. Compare these with"
print "the relativistic velocity in Table T4."
header['CTYPE1'] = 'VELO-HEL'
header['VELREF'] = 2
proj = wcs.Projection(header)
print "Allowed spectral translations for VELO as optical velocity", proj.altspec
p2 = proj.spectra('VRAD-???')
V = proj.toworld1d(pixels)
Z = p2.toworld1d(pixels)
print "Pixel Voptical in (km/s) and Vradio (km/s)"
for p,v,z in zip(pixels, V, Z):
  print "%4d %15f %15f" % (p, v/1000, z/1000)

print "\nT4. Next a list with optical velocities calculated from relativistic"
print "velocity with constant increment."
print "If these values are different from the previous optical velocity then "
print "obviously the velocities derived from the header are not relativistic"
print "as in pre 4.5.1 versions of WCSLIB."
v0 = V0
for i in pixels:
  v1 = v0 + (i-crpix)*dV
  beta = v1/c
  frac = (1-beta)/(1+beta)
  f = f0 * sqrt(frac)
  Z = c* (f0-f)/f
  print "%4d %15f" % (i ,Z/1000.0)
```

Output:

```
 The velocity increment is constant and equal to 5.000000 (km/s):
 Allowed spectral translations [('FREQ', 'Hz'), ('ENER', 'J'), ('WAVN', '/m'),
 ('VOPT-F2W', 'm/s'), ('VRAD', 'm/s'), ('VELO-F2V', 'm/s'), ('WAVE-F2W', 'm'),
 ('ZOPT-F2W', ''), ('AWAV-F2A', 'm'), ('BETA-F2V', '')]

 T1. Radio velocity directly from header and optical velocity
 from spectral translation. VELO is a radio velocity here because
 VELREF > 256
 Pixel Vradio in (km/s) and Voptical (km/s)
 30     -253.000000     -252.786669
 31     -248.000000     -247.795014
 32     -243.000000     -242.803193
 33     -238.000000     -237.811206
 34     -233.000000     -232.819052

 T2. Now insert CTYPE1='VRAD' in the header and convert to VOPT-F2W
 with a spectral translation (Z1) and with a calculation (Z2)
 This should give the same results as in table T1.

 With CTYPE='RAD' and spec.trans 'VOPT-F2W': Pixel , Vrad, Z1 (km/s), Z2 (km/s)
 30 -253.0 -252.786668992 -252.786668992
 31 -248.0 -247.795014311 -247.795014311
 32 -243.0 -242.803193261 -242.803193261
 33 -238.0 -237.811205834 -237.811205834
 34 -233.0 -232.819052022 -232.819052022

 T3. We set CTYPE1 to VELO-HEL and VELREF to 2 (Helio) and
 derive optical and radio velocities from it. Compare these with
 the relativistic velocity in Table T4.
 Allowed spectral translations for VELO as optical velocity [('FREQ-W2F', 'Hz'),
('ENER-W2F', 'J'), ('WAVN-W2F', '/m'), ('VOPT', 'm/s'), ('VRAD-W2F', 'm/s'),
('VELO-W2V', 'm/s'), ('WAVE', 'm'), ('ZOPT', ''), ('AWAV-W2A', 'm'),
('BETA-W2V', '')]
 Pixel Voptical in (km/s) and Vradio (km/s)
 30     -253.000000     -253.213691
 31     -248.000000     -248.205325
 32     -243.000000     -243.197126
 33     -238.000000     -238.189094
 34     -233.000000     -233.181229

 T4. Next a list with optical velocities calculated from relativistic
 velocity with constant increment.
 If these values are different from the previous optical velocity then
 obviously the velocities derived from the header are not relativistic
 as in pre 4.5.1 versions of WCSLIB.
 30     -252.893335
 31     -247.897507
 32     -242.901597
 33     -237.905603
 34     -232.909526
```

We used eq. (3.55) to calculate a frequency for a given apparent radial velocity. This frequency is used in eq. (3.52) to calculate the optical velocity. The script proves:

- Axis VELO-HEL is processed as an optical velocity and if keyword *VELREF* is present and its value is greater than 256, then VELO-HEL is processed as a radio velocity. In versions of WCSLIB < 4.5.1, the VELO-XXX axis was processed as VELO i.e. a relativistic velocity.

Note: From the WCSLIB API documentation:

AIPS-convention celestial projection types, NCP and GLS, and spectral types, '{FREQ,FELO,VELO}-{OBS,HEL,LSR}' as in 'FREQ-LSR', 'FELO-HEL', etc., set in CTYPEia are translated on-the-fly by wcsset() but without modifying the relevant ctype[], pv[] or specsys members of the wcsprm struct. That is, only the information extracted from ctype[] is translated when wcsset() fills in wcsprm::cel (celprm struct) or wcsprm::spc (spcprm struct).

On the other hand, these routines do change the values of wcsprm::ctype[], wcsprm::pv[], wcsprm::specsys and other wcsprm struct members as appropriate to produce the same result as if the FITS header itself had been translated.

Definitions and formulas from AIPS and GIPSY

AIPS

A radio velocity is defined by:

$$V = c(\frac{\nu_0 - \nu^{'}}{\nu_0}) \tag{3.107}$$

where ν is the Doppler shifted rest frequency, given by:

$$\nu' = \nu_0 \sqrt{(\frac{c-v}{c+v})} \tag{3.108}$$

Equivalent to the relativistic addition of apparent radial velocities we can derive a relation for radio velocities if the velocities in given in different reference systems.

The addition of apparent radial velocities is given in AIPS memo 27 *[Aipsmemo]* Non-Linear Coordinate Systems in AIPS (Eric W. Greisen, NRAO) Greisen, is

$$v = \frac{v_s + v_{obs}}{1 + \frac{v_s v_{obs}}{c^2}} \tag{3.109}$$

To stay close to our previous examples and definitions we set v_s which is the apparent radial velocity of an object w.r.t. an inertial system, to be equal to v_b (our inertial system in this case is barycentric).

The other velocity, v_{obs} is equal to the topocentric correction: v_t and the result $v = v_e$, the apparent radial velocity of the object as we would observe it on earth.

Then we get the familiar formula (eq. (3.59)):

$$v_e = \frac{v_b + v_t}{1 + \frac{v_b v_t}{c^2}} \tag{3.110}$$

With the relation between V and v and the relativistic addition of velocities we find that the radio velocities in different systems are related according to the equation:

$$V_e = V_b + V_t - V_b V_t / c \tag{3.111}$$

(see also AIPS memo 27 *[Aipsmemo]*). The barycentric radio velocity was calculated in a previous section. Its value was V_b = 8850750.90404 m/s. With the topocentric reference frequency 1378351174.05 Hz we find V_e = 8876087.18567 m/s. We know from fig. 1 that the topocentric correction is positive. To calculate the corresponding radio velocity V_t we use:

$$V_t = c(\frac{\nu_b - \nu_e}{\nu_b}) = 299792458.0 \times \frac{(1378471216.43 - 1378351174.05)}{1378471216.43} = 26107.03781\ m/s \tag{3.112}$$

With these values for V_b and V_t you can verify that the expression for V_e is valid.

$$V_e = 8850750.90404 + 26107.03781 - \frac{8850750.90404 \times 26107.03781}{299792458.0} = 8876087.18567\ m/s \tag{3.113}$$

which is the value of V_e that we found before using the topocentric reference frequency, so we can have confidence in the relation for radio velocities as found in the AIPS memo *[Aipsmemo]* .

But this radio velocity V_e (w.r.t. observer on Earth) for a pixel N is also given by the relation:

$$V_e(N) = -\frac{c}{\nu_0}(\nu_e(N) - \nu_0) = -\frac{c}{\nu_0}(\nu_e + \delta_\nu(N - N_\nu) - \nu_0) \tag{3.114}$$

It is important to emphasize the meaning of the variables:

- ν_e = topocentric reference frequency).
- δ_ν = the increment in frequency per pixel in the topocentric system
- N_ν = the frequency reference pixel
- N = the pixel

If we use the previous formulas we can also write:

$$V_e(N_V) = V_b' + V_t - V_b' V_t/c \tag{3.115}$$

$$V_e(N_V) = -\frac{c}{\nu_0}(\nu_e + \delta_\nu(N_V - N_\nu) - \nu_0) \tag{3.116}$$

The velocity $V_b^{'}$ is the barycentric reference velocity at velocity reference pixel N_V.

From these relations we observe:

$$V_b(N) = \frac{V_e(N) - V_t}{1 - \frac{V_t}{c}} \tag{3.117}$$

and from eq. (3.115) with $V_b^{'} = V_b(N_V)$:

$$V_t = \frac{V_e(N_V) - V_b(N_V)}{1 - \frac{V_b(N_V)}{c}} \tag{3.118}$$

Using also the equations with the frequencies, we can derive the following expression for $V_b(N)$:

$$V_b(N) = V_b(N_V) - \frac{\delta_\nu\big(c - V_b(N_V)\big)(N - N_V)}{\nu_e + \delta_\nu(N_V - N_\nu)} \tag{3.119}$$

or in an alternative notation:

$$V_b(N) = V_b(N_V) + \delta_V(N - N_V) \tag{3.120}$$

Note that in AIPS memo 27 *[Aipsmemo]* the variable V_R is used for $V_b(N_V)$ and V_R and N_V are stored in AIPS headers as alternative reference information (if frequency is in the main axis description).

The difference between the velocity and frequency reference pixel can be expressed in terms of the radio velocities $V_b(N_V)$ and $V_b(N_\nu)$. It follows from eq. (3.119)) that for $N = N_\nu$ and a little rearranging:

$$N_V - N_\nu = \frac{\nu_e\,\big[V_b(N_\nu) - V_b(N_V)\big]}{\delta_\nu\,\big[c - V_b(N_\nu)\big]} \tag{3.121}$$

We conclude that either one calculates (barycentric) radio velocities using the reference frequency and the frequency increment from the header, or one calculates these velocities using a reference velocity and a velocity increment from the header.

Note that we assumed that the frequency increment in the barycentric system is the same as in the the system of the observer, which is not correct. However the differences are small (less than 0.01% for 100 pixels from the reference pixel for typical observations as in our examples).

For optical velocities Greisen derives:

$$Z_e = Z_b + Z_t + Z_b Z_t / c \tag{3.122}$$

and:

$$Z_b(N) = Z_b(N_V) - \frac{\delta_\nu \left(c + Z_b(N_V)\right)(N - N_Z)}{\nu_e + \delta_\nu (N - N_\nu)} \tag{3.123}$$

The difference between the velocity and frequency reference pixels in terms of optical velocity is:

$$N_Z - N_\nu = \frac{\nu_e \left[Z_b(N_\nu) - Z_b(N_Z)\right]}{\delta_\nu \left[c + Z_b(N_\nu)\right]} \tag{3.124}$$

Next script demonstrates how we reconstruct the topocentric optical velocity and the reference pixel for that velocity as it is used in the AIPS formula. Then we compare the output of the WCSLIB method and the AIPS formula:

```
#!/usr/bin/env python
from kapteyn import wcs
import numpy

c   = 299792458.0             # m/s From literature
f0  = 1.42040575200e+9        # Rest frequency HI (Hz)
fR  = 1.37835117405e+9        # Topocentric reference frequency (Hz)
dfR = 9.765625e+4             # Increment in topocentric frequency (Hz)
fb  = 1.3784712164292786e+9 # Barycentric reference frequency (Hz)
dfb = 97647.745732            # Increment in barycentric frequency (Hz)
Zb  = 9120.0e+3               # Barycentric optical velocity in m/s
Nf  = 32                      # Reference pixel for frequency

header = { 'NAXIS'   : 1,
           'CTYPE1' : 'FREQ',
           'CRVAL1' : fb,
           'CRPIX1' : Nf,
           'CUNIT1' : 'Hz',
           'CDELT1' : dfR,
           'RESTFRQ': f0
         }
line = wcs.Projection(header).spectra('VOPT-F2W')
pixels = numpy.array(range(30,35))
Vwcs = line.toworld1d(pixels) / 1000
print """Optical velocities from WCSLIB with spectral
translation and with barycentric ref. freq. (km/s):"""
for p,v in zip(pixels, Vwcs):
   print p, v

# Select an arbitrary velocity reference pixel
Nz = 44.0
# then calculate corresponding velocity
Zb2 = (fR*Zb-dfR*c*(Nz-Nf))/(fR+dfR*(Nz-Nf))
print "Zb(Nz) =", Zb2
```

```
dN = fR*(Zb-Zb2)/(dfR*(c+Zb2))
Nz = dN + Nf
print "Closure test for selected reference pixel: Nz=", Nz

print "\nOptical velocities using AIPS formula (km/s):"
Zs = Zb2 - dfR*(c+Zb2)*(pixels-Nz)/(fR+dfR*(pixels-Nf))
Zs /= 1000
for p,z in zip(pixels, Zs):
   print p, z

fx = fR + dfR*(Nz-Nf)
dZ = -dfR*(c+Zb2) / fx
print "Velocity increment: ", dZ

header = { 'NAXIS'  : 1,
           'CTYPE1' : 'VOPT-F2W',
           'CRVAL1' : Zb2,
           'CRPIX1' : Nz,
           'CUNIT1' : 'm/s',
           'CDELT1' : dZ,
           'RESTFRQ': f0
         }
line2 = wcs.Projection(header)
Vwcs = line2.toworld1d(pixels) / 1000
print """\nOptical velocities from WCSLIB without spectral
translation with barycentric Z (km/s):"""
for p,v in zip(pixels, Vwcs):
   print p, v
# Output:
# Optical velocities from WCSLIB with spectral
# translation and with barycentric ref. freq. (km/s):
# 30 9163.77531689
# 31 9141.88610773
# 32 9120.0
# 33 9098.11699305
# 34 9076.23708621
# Zb(Nz) = 8857585.54671
# Closure test for selected reference pixel: Nz= 44.0
#
# Optical velocities using AIPS formula (km/s):
# 30 9163.77912988
# 31 9141.88801395
# 32 9120.0
# 33 9098.11508736
# 34 9076.23327538
# Velocity increment:  -21849.2948239
#
# Optical velocities from WCSLIB without spectral
# translation with barycentric Z (km/s):
# 30 9163.77912988
# 31 9141.88801395
# 32 9120.0
# 33 9098.11508736
# 34 9076.23327538
```

Note that we used the topocentric frequency increment in the WCSLIB call for a better comparison with the AIPS formula. The output of velocities with the AIPS formula is exactly the same as WCSLIB with optical velocities using the velocity increment calculated with the AIPS method (as to be expected). And these velocities are very close to the velocities calculates with WCSLIB using the barycentric frequency that corresponds to the given optical velocity. The differences can be explained by the fact that the different methods are used to calculate a velocity increment.

What did we prove with this script? We selected an arbitrary pixel as reference pixel for the velocity. This velocity has a relation with the initial optical velocity (9120 km/s) through the difference in reference pixels. We calculated that velocity and showed that the AIPS formula generates results that are almost equal to WCSLIB with the barycentric reference frequency. If we use the AIPS formulas to calculate a velocity increment, we can use the values in WCSLIB if we set CTYPE to 'VOPT-F2W'. This generates exactly the same results as with the AIPS formula for velocities. So in frequency mode WCSLIB calculates topocentric frequencies (and *topocentric* velocities if we use a spectral translation method) and in velocity mode it calculates barycentric velocities. AIPS axis type FELO can be used as input for WCSLIB without modification.

Conclusions

- In AIPS the reference pixel for the reference velocity differs from the frequency reference pixel. There is a relation between this reference velocity and the barycentric velocity and these reference pixels. To us it is not clear what this reference velocity represents and why it is not changed to a velocity at the same reference pixel as the frequency.
- In the AIPS approach it is assumed that the increment in frequency is the same in different reference systems. This assumption is not correct, but the deviations are usually very small.

GIPSY

The formulas used in GIPSY to convert frequencies to velocities are described in section: spectral coordinates in the GIPSY programmers guide. There is a formula for optical velocities and one for radio velocities. Both formulas are derived from the standard formulas for velocities but the result is split into a reference velocity and a part that is a non linear function of the increment in frequency.

Optical For optical velocities we use symbol *Z*. The conversion from frequencies to **optical** velocities is not linear. One can try to approximate a constant step in velocity, and to apply the standard linear transformation $Z(N) = Z_r + (N - crpix) \times dZ$, but this approximation can deviate significantly in certain circumstances. Therefore most reduction and analysis packages provide functionality to calculate velocities also for the non-linear cases. Like Classic AIPS, GIPSY provides a system for these transformations (e.g. function `velpro.c`), but it turns out that these transformations are also approximations because where a barycentric or lsrk frequency should be used, GIPSY uses values from the FITS header and for FITS files made by Newstar/Nmap for data observed before 2006-07-03, these frequencies are topocentric. In this section we show how GIPSY transforms frequencies to optical velocities. Also we derive formulas for a linear transformation (i.e. for a constant velocity increment) which can be used if one wants to compose a modified header for a linear transformation $Z(N) = Z_r + (N - crpix) \times dZ$

Given a barycentric (or lsrk) frequency one calculates an optical velocity *Z* in that system with:

$$Z = -c(\frac{\nu_b - \nu_0}{\nu_b}) \tag{3.125}$$

Assume for channel N:

$$\nu(N) = \nu_{br} + (N - N_{ref})\delta_{\nu_b} = \nu_{br} + \mathbf{n}\delta_{\nu_b} \tag{3.126}$$

For $(N - N_{ref})$ we wrote $\mathbf{n}$. The frequencies are related to the barycentric (or lrsk) reference system. N_{ref} is the reference pixel (*CRPIX*) given in a FITS header, ν_{br} is the reference frequency in this barycentric system and δ_{ν_b} is the barycentric frequency increment.

Inserting (3.126) into (3.125) gives:

$$Z(N) = -c(\frac{\nu_{br} + \mathbf{n}\delta_{\nu_\mathbf{b}} - \nu_\mathbf{0}}{\nu_{br} + \mathbf{n}\delta_{\nu_\mathbf{b}}}) = -c(\frac{\nu_{br} - \nu_0}{\nu_{br}}) + \mathbf{n}dZ = Z_r + \mathbf{n}dZ \tag{3.127}$$

Z_r is the given reference velocity in the barycentric/lsrk reference system. Solve this equation for $\mathbf{n}dZ$ to get an expression for the increment:

$$\mathbf{n}dZ = \mathbf{n}\,\frac{-c\nu_0\delta_{\nu_b}}{(\nu_{br}+\mathbf{n}\delta_{\nu_b})\nu_{br}} = c\nu_0\big(\frac{1}{(\nu_{br}+\mathbf{n}\delta_{\nu_b})} - \frac{1}{\nu_{br}}\big) \tag{3.128}$$

The formula to calculate optical velocities then becomes:

$$Z(N) = Z_r + c\nu_0\big(\frac{1}{(\nu_{br}+\mathbf{n}\delta_{\nu_b})} - \frac{1}{\nu_{br}}\big) \tag{3.129}$$

with:

- $Z(N)$ is the barycentric optical velocity for pixel N
- ν_{br} is the barycentric reference frequency
- δ_{ν_b} is the increment in barycentric frequency

This is the formula that GIPSY uses to calculate optical velocities. However, GIPSY uses the topocentric reference frequency and the topocentric frequency increment.

If we want to express the optical velocity at pixel N as a function of the reference velocity and a **constant** velocity increment as in $Z(N) = Z_r + \mathbf{n}dZ$, then we need to find an expression for *dZ* which does not depend on *n*. Rewrite *ndZ* into:

$$\mathbf{n}dZ = \mathbf{n}\frac{-c\nu_0\delta_{\nu_b}}{(\nu_{br}+n\delta_{\nu_b})\nu_{br}} \tag{3.130}$$

Then, with the observation that $\mathbf{n}\delta_{\nu_b} << \nu_{br}$:

$$\mathbf{n}dZ \approx \mathbf{n}\frac{-c\nu_0\delta_{\nu_b}}{{\nu_{br}}^2} \tag{3.131}$$

and thereby:

$$dZ \approx \frac{-c\nu_0\delta_{\nu_b}}{{\nu_{br}}^2} \tag{3.132}$$

This is the formula that is documented in the programmers manual to get a value for GIPSY's keyword *DDELT* (one of the alternative keywords from the list DRVAL, DDELT, DRPIX, DUNIT which describe an alternative coordinate system with a higher priority than the system described by the corresponding keywords that start with 'C'). However the formula is never used in GIPSY to explicitly set the value of DDELT. Only when DDELT is given in a header, it is used as an increment.

So the formula to calculate optical velocities, without the use of the rest frequency, is:

$$Z(N) = Z_r + \mathbf{n}\frac{-c\nu_0\delta_{\nu_b}}{{\nu_{br}}^2} \tag{3.133}$$

In the formulas above we included the rest frequency. But it is not necessary to know its value because we can express this rest frequency in terms of optical velocity:

$$Z = -c(\frac{\nu_b - \nu_0}{\nu_b}) \rightarrow \nu_0 = \nu_{br}\big(1+\frac{Z_r}{c}\big) \tag{3.134}$$

Then:

$$Z(N) = Z_r + c\nu_{br}\big(1+\frac{Z_r}{c}\big)\big(\frac{1}{(\nu_{br}+\mathbf{n}\delta_{\nu_b})} - \frac{1}{\nu_{br}}\big) \tag{3.135}$$

from which we derive in a straightforward way:

$$Z(N) = \frac{Z_r\nu_{br} - c\mathbf{n}\delta_{\nu_b}}{\nu_{br}+\mathbf{n}\delta_{\nu_b}} \tag{3.136}$$

The formula above is the method used by GIPSY's function velpro.c to get velocities if the rest frequency is unknown.

And again, if we want to express the optical velocity at pixel N as a function of the reference velocity and a **constant** velocity increment as in $Z(N) = Z_r + \mathbf{n}dZ$ then we need to find an expression for dZ which does not depend on n. Note that $\mathbf{n}\delta\nu_b << \nu_{br}$, then

$$Z(N) \approx \frac{Z_r\nu_{br} - \mathbf{n}c\delta_{\nu_b}}{\nu_{br}} = Z_r + \mathbf{n}(-c\frac{\delta_{\nu_b}}{\nu_{br}}) \tag{3.137}$$

Next script implements these formulas and show the deviations. The first three columns show the correct result.

```
from kapteyn import wcs
from math import sqrt
from numpy import arange

header_gds = {
             'NAXIS'  : 1,
             'NAXIS1' : 127,
             'CTYPE1' : 'FREQ-OHEL',
             'CRVAL1' : 1418921567.851000,
             'CRPIX1' : 63.993952051196288,
             'CUNIT1' : 'HZ',
             'CDELT1' : -9765.625,
             'VELR'   : 304000.0,
             'RESTFRQ': 1420405752.0,
           }

f0 = header_gds['RESTFRQ']
Zr = header_gds['VELR']
fr = header_gds['CRVAL1']
df = header_gds['CDELT1']
crpix = header_gds['CRPIX1']
c = wcs.c                                    # Speed of light
p = pixrange = arange(crpix-2, crpix+3)      # Range of pixels for which we
                                             # want world coordinates
# Calculate the barycentric equivalents
fb = f0/(1.0+Zr/c)
Vtopo = c * ((fb*fb-fr*fr)/(fb*fb+fr*fr))
dfb = df*(c-Vtopo)/sqrt(c*c-Vtopo*Vtopo)
print "Topocentric correction (km/s):", Vtopo/1000
print "Barycentric frequency and increment (Hz):", fb, dfb

# VOPT-F2W from spectral translation, assumed to give the correct velocities
proj = wcs.Projection(header_gds)
spec = proj.spectra(ctype='VOPT-F2W')
Z1 = spec.toworld1d(pixrange)

# Non linear: Optical with GIPSY formula with barycentric
# values (excact).
Z2 = Zr + c*f0*(1/(fb+(p-crpix)*dfb)-1/fb)

# Non Linear: Optical with GIPSY formula without rest frequency and
# with barycentric values (exact).
Z3 = (Zr*fb - (p-crpix)*c*dfb) / (fb+(p-crpix)*dfb)

# Non Linear: Optical with GIPSY formula using topocentric,
# values (approximation).
Z4 = Zr + c*f0*(1/(fr+(p-crpix)*df)-1/fr)
```

```
# Linear: Optical with GIPSY formula with barycentric values
# and dZ approximation for linear transformation
# Rest frequency is part of formula.
dZ = -c*f0*dfb/fb/fb
Z5 = Zr + (p-crpix) * dZ

# Linear: Optical with GIPSY formula with barycentric values
# and dZ approximation for linear transformation
# Rest frequency is not used.
dZ = -c *dfb/fb
Z6 = Zr + (p-crpix) * dZ

print "\n%10s %14s %14s %14s %14s %14s %14s" % ('pix', 'WCSLIB',
'GIP+bary', 'GIP+bary-f0', 'GIP+topo', 'Linear+f0', 'Linear-f0')
for pixel, z1,z2,z3,z4,z5, z6 in zip(pixrange, Z1, Z2, Z3, Z4, Z5, Z6):
   print "%10.4f %14f %14f %14f %14f %14f %14f" % (pixel, z1/1000, z2/1000,
         z3/1000, z4/1000, z5/1000, z6/1000)
```

Output:

```
Topocentric correction (km/s): 9.57140206387
Barycentric frequency and increment (Hz): 1418966870.14 -9765.3132202

     pix         WCSLIB       GIP+bary    GIP+bary-f0
 61.9940     299.869536     299.869536     299.869536
 62.9940     301.934754     301.934754     301.934754
 63.9940     304.000000     304.000000     304.000000
 64.9940     306.065274     306.065274     306.065274
 65.9940     308.130577     308.130577     308.130577

   GIP+topo      Linear+f0      Linear-f0
 299.869141     299.869479     299.873664
 301.934556     301.934740     301.936832
 304.000000     304.000000     304.000000
 306.065472     306.065260     306.063168
 308.130973     308.130521     308.126336
```

The columns in the output are:

1. *pix*: The (non integer) pixel value at which a velocity is calculated.
2. *WCSLIB*: The optical velocity (km/s) as calculated by WCSLIB. The extension in CTYPE is recognized and the frequencies are replaced by their barycentric counterparts according to the recipe in *A recipe for modification of Nmap/GIPSY FITS data*.
3. *GIP+bary*: The optical velocity (km/s) calculated with GIPSY formula in eq. (3.129) using barycentric reference frequency and barycentric frequency increment.
4. *GIP+bary-f0*: The optical velocity (km/s) calculated with GIPSY formula without the rest frequency as in eq. (3.136) using barycentric reference frequency and barycentric frequency increment.
5. *GIP+topo*: The optical velocity (km/s) calculated with GIPSY formula in eq. (3.129) using topocentric/geocentric reference frequency and frequency increment.
6. *Linear+f0*: The optical velocity (km/s) calculated with GIPSY formula in eq. (3.133) using a rest frequency.
7. *Linear-f0*: The optical velocity (km/s) calculated with GIPSY formula in eq. (3.137) without a rest frequency.

If you do some experiments with the values in this script, you will observe that the GIPSY formula with topocentric instead of the barycentric/lsrk values is not a bad approximation although it is sensitive to the channel number (p). The linear approximations are worse and should be avoided if high precision is required.

What remains is the question how good GIPSY's approximation is. With (3.129) we write:

$$Z_{\nu_b}(N) - Z_{\nu_t}(N) = c\nu_0\Big(\frac{1}{\nu_{br} + \mathbf{n}\delta_{\nu_b}} - \frac{1}{\nu_{br}} - \big(\frac{1}{\nu_{tr} + \mathbf{n}\delta_{\nu_t}} - \frac{1}{\nu_{tr}}\big)\Big) \tag{3.138}$$

With the parameters:

- $Z_{\nu_t}(N)$ the optical velocity at pixel *N* using topocentric values
- ν_{tr} the topocentric frequency at the reference pixel
- δ_{ν_t} the topocentric frequency increment

Rewrite this in:

$$Z_{\nu_b}(N) - Z_{\nu_t}(N) = -\mathbf{n}c\nu_0\big(\frac{\delta_{\nu_b}}{\nu_{br}(\nu_{br} + \mathbf{n}\delta_{\nu_b})} - \frac{\delta_{\nu_t}}{\nu_{tr}(\nu_{tr} + \mathbf{n}\delta_{\nu_t})}\big) \tag{3.139}$$

Note that $\mathbf{n}\delta\nu_b << \nu_{br}$ and $\mathbf{n}\delta\nu_t << \nu_{tr}$. Then write the difference in increment as function of *N* as:

$$Z_{\nu_b}(N) - Z_{\nu_t}(N) \approx -\mathbf{n}c\nu_0\big(\frac{\delta_{\nu_b}}{\nu_{br}^2} - \frac{\delta_{\nu_t}}{\nu_{tr}^2}\big) \tag{3.140}$$

This expression explains the different values in the output of our previous script and it shows that the differences depend on **n**.

Use (3.64) to write:

$$\nu_{tr} = \nu_{br}\sqrt{\frac{c - v_{tc}}{c + v_{tc}}} \tag{3.141}$$

and from (3.66)

$$\delta_{\nu_b} = \delta_{\nu_t}\sqrt{\frac{c - v_{tc}}{c + v_{tc}}} \tag{3.142}$$

Define $q = \sqrt{\frac{c-v_{tc}}{c+v_{tc}}}$ then $\nu_{br} = q/\nu_{tr}$ and $\delta_{\nu_b} = q * \delta_{\nu_t}$

Insert this in (3.140) to obtain:

$$Z_{\nu_b}(N) - Z_{\nu_t}(N) \approx -\mathbf{n}c\nu_0\frac{\delta_{\nu_t}}{\nu_{tr}^2}(q^3 - 1) \tag{3.143}$$

The topocentric correction v_{tc} has a range between -30 Km/s and 30 Km/s. For $v_{tc} = 30000$ *m/s* this corresponds to a maximum of q: $q = 0.99989993577786473$. With this maximum for *q* we find for (3.143) approximately 0.62 m/s

Note that the difference is a function of **n**, so after 64 channels the deviation is almost 40 m/s. In our example, the channel separation is approximately 2 km/s and the deviations are therefore small (2%).

For the example at the start of this chapter, the reference velocity was 9120 km/s. The channel separation (*CDELT3Z*) is approximately 20 km/s. For the listed topocentric frequency and the calculated barycentric frequency we find with (3.143) an error of approximately 6.6 m/s. After 64 channels the deviation is approximately 420 m/s (2%).

With (3.132) we get an relative error:

$$\frac{Z_{\nu_b}(N) - Z_{\nu_t}(N)}{dZ} = \mathbf{n}(q^3 - 1)\frac{\delta_{\nu_t}}{\nu_{tr}^2}\frac{\nu_{br}^2}{\delta_{\nu_t}} \approx \mathbf{n}(q^3 - 1) \tag{3.144}$$

With the maximum value of *q* we find a maximum percentage of 0.03% for 1 channel. After 64 channels the deviation is almost 2%. After 512 channels it is more than 15%.

Conclusions

- The GIPSY formulas assume constant frequency increments in the system of the reference system. When these are topocentric, there are small deviations from the result with WCSLIB which assume the frequencies in the same reference system as the given velocity.
- The formula that GIPSY routines use to calculate optical velocities is an approximation. The deviations are small but depend on the pixel i.e. $(N - N_\nu)$. This approximation is not necessary because when the optical velocity in the barycenter is given, then one can calculate the barycentric reference frequency (see eq. (3.53)) and use that frequency in the GIPSY formula to get the exact result.
- The deviation is more sensitive to the topocentric correction (velocity between observatory on earth and barycenter/lsrk) than the reference frequency and the frequency increment. Also there is a maximum value for the topocentric velocity which results in a maximum deviation of 0.03% for one channel.

For the data in the previous script, we used the code below (which should be added to the previous script) to calculate the percentages:

```
q = sqrt((c-Vtopo)/(c+Vtopo))
delta = -c*f0*df/fr/fr * (q*q*q-1)
d = (p-crpix) * delta

# Now change the topocentric correction to its maximum.
Vtopo = 30000.0
qmax = sqrt((c-Vtopo)/(c+Vtopo))
deltamax = -c*f0*df/fr/fr * (qmax*qmax*qmax-1)
dmax = (p-crpix) * deltamax
perc = abs(100*deltamax/dZ)

print "dZ, deltamax:", dZ, deltamax
print "Percentage deviation for 1 channel: ", perc
print "Approximate percentage: ", abs(100 * (qmax*qmax*qmax-1))
print "Percentage deviation for 64 channel: ", 64*perc
print "Approximate percentage: ", abs(100 * 64*(qmax*qmax*qmax-1))
print "Percentage deviation for 64 channel: ", 512*perc
print "Approximate percentage: ", abs(100 * 512*(qmax*qmax*qmax-1))

print "\nThe approximate difference and the real difference"
print "between topocentric nd barycentric increments"
for pixel, d1,d2,d3 in zip(pixrange, d, Z2-Z4, dmax):
   print "%10.4f %14f %14f %14f" % (pixel, d1/1000, d2/1000, d3/1000)
```

Output:

```
dZ, deltamax: -21236.6115174 6.57007047211
Percentage deviation for 1 channel:  0.0309374707295
Approximate percentage:  0.0300162628862
Percentage deviation for 64 channel:  1.97999812669
Approximate percentage:  1.92104082472
Percentage deviation for 64 channel:  15.8399850135
Approximate percentage:  15.3683265977

The approximate difference and the real difference
between topocentric and barycentric increments and
the maximum deviation as function of the pixel:
61.9940      -0.011436      -0.011438      -0.013140
62.9940      -0.005718      -0.005719      -0.006570
63.9940       0.000000       0.000000       0.000000
64.9940       0.005718       0.005717       0.006570
65.9940       0.011436       0.011433       0.013140


```

```
    61.9940       -0.011436       -0.011438       -0.013140
    62.9940       -0.005718       -0.005719       -0.006570
    63.9940        0.000000        0.000000        0.000000
    64.9940        0.005718        0.005717        0.006570
    65.9940        0.011436        0.011433        0.013140
```

Radio Given a frequency, a radio velocity is calculated with the formula:

$$V = -c(\frac{\nu' - \nu_0}{\nu_0}) \tag{3.145}$$

Assume for channel N:

$$\nu(N) = \nu_{br} + (N - N_{ref})\delta_{\nu_b} = \nu_{br} + \mathbf{n}\delta_{\nu_b} \tag{3.146}$$

For $(N - N_{ref})$ we wrote $\mathbf{n}$. The frequencies are related to the barycentric (or lrsk) reference system. N_{ref} is the reference pixel (*CRPIX*) given in a FITS header, ν_{br} is the reference frequency in this barycentric system and δ_{ν_b} is the barycentric frequency increment.

Inserting (3.145) into (3.146) gives:

$$V_b(N) = -c(\frac{\nu_{br} + \mathbf{n}\delta_{\nu_b} - \nu_0}{\nu_0}) = V_r + \mathbf{n}\frac{-c\delta_{\nu_b}}{\nu_0} \tag{3.147}$$

with:

- $V_b(N)$ is the barycentric radio velocity for pixel N using barycentric frequency increments
- ν_{br} is the barycentric reference frequency
- δ_{ν_b} is the increment in barycentric frequency

This increment in radio velocity was also derived in eq. (3.79). The increment in radio velocity is a linear function of the increment in frequency. The frequencies in the FITS and GIPSY headers for pre July, 2006 WSRT/Nmap FITS files are the topocentric frequencies.

We show the difference between the velocities derived from the barycentric/lsrk values and the velocities derived from the topocentric values.

```python
from kapteyn import wcs
from math import sqrt
from numpy import arange

header_gds = {
           'NAXIS'  : 1,
           'NAXIS1' : 127,
           'CTYPE1' : 'FREQ-RHEL',
           'CRVAL1' : 1418921567.851000,
           'CRPIX1' : 63.993952051196288,
           'CUNIT1' : 'HZ',
           'CDELT1' : -9765.625,
           'VELR'   : 304000.0,
           'RESTFRQ': 1420405752.0,
          }

f0 = header_gds['RESTFRQ']
Vr = header_gds['VELR']
fr = header_gds['CRVAL1']
df = header_gds['CDELT1']
crpix = header_gds['CRPIX1']
```

```
c = wcs.c                                    # Speed of light
p = pixrange = arange(crpix-2, crpix+3)      # Range of pixels for which we
                                             # want world coordinates
# Calculate the barycentric equivalents
fb = f0*(1.0-Vr/c)
Vtopo = c * ((fb*fb-fr*fr)/(fb*fb+fr*fr))
dfb = df*(c-Vtopo)/sqrt(c*c-Vtopo*Vtopo)
print "Topocentric correction (km/s):", Vtopo/1000
print "Barycentric frequency and increment (Hz):", fb, dfb

# VRAD from spectral translation, assumed to give the correct velocities
proj = wcs.Projection(header_gds)
spec = proj.spectra(ctype='VRAD')
V1 = spec.toworld1d(pixrange)

# Radio velocities with GIPSY formula with barycentric
# values (excact).
V2 = Vr - c*(p-crpix)*dfb/f0

# Radio velocities with GIPSY formula without rest frequency and
# with barycentric values (exact).
V3 = Vr + (p-crpix)*dfb*(Vr-c)/fb

# Radio velocities with GIPSY formula using topocentric,
# values (approximation).
V4 = Vr - c*(p-crpix)*df/f0

# Check the differences
# d = -c*(p-crpix)*(df-dfb)/f0
# print (V4-V1)/1000, d/1000

print "\n%10s %14s %14s %14s %14s" % ('pix', 'WCSLIB',
'GIP+bary', 'GIP+bary-f0', 'GIP+topo')
for pixel, v1,v2,v3,v4 in zip(pixrange, V1, V2, V3, V4):
   print "%10.4f %14f %14f %14f %14f" % (pixel, v1/1000, v2/1000,
         v3/1000, v4/1000)
```

Output:

```
Topocentric correction (km/s): 9.26313531147
Barycentric frequency and increment (Hz): 1418965411.07 -9765.32326156

       pix         WCSLIB       GIP+bary    GIP+bary-f0       GIP+topo
   61.9940     299.877839     299.877839     299.877839     299.877712
   62.9940     301.938920     301.938920     301.938920     301.938856
   63.9940     304.000000     304.000000     304.000000     304.000000
   64.9940     306.061080     306.061080     306.061080     306.061144
   65.9940     308.122161     308.122161     308.122161     308.122288
```

The second, third and fourth column represent V_b and the last column is V_t. The difference between the exact and approximate velocities as function of **n** is given by:

$$V_t(N) - V_b(N) = -\mathbf{n}\frac{c}{\nu_0}(\delta_{\nu_t} - \delta_{\nu_b}) \tag{3.148}$$

With the parameters:

- $V_t(N)$ the barycentric radio velocity at pixel N using topocentric frequency increments
- δ_{ν_t} the topocentric frequency increment

The topocentric correction v_{tc} has a range between -30 Km/s and 30 Km/s. Rewrite (3.66) into:

$$\frac{\delta_{\nu_b}}{\delta_{\nu_t}} = \sqrt{\frac{c - v_{tc}}{c + v_{tc}}} \tag{3.149}$$

For $v_{tc} = 30000$ *m/s* this corresponds to a maximum $q = \delta_{\nu_b}/\delta_{\nu_t} = 0.99989993577786473$ which is equivalent to:

$$\frac{c}{\nu_0}(1-q)\delta_{\nu_t} \approx 0.2\, m/s \tag{3.150}$$

Note that the difference is a function of **n**, so after 64 channels the deviation is more than 12 m/s. In our example, the channel separation is approximately 2 km/s and the deviations are therefore small.

Header items in a (legacy) WSRT FITS file

Program *nmap* (part of NEWSTAR which is a package developed to process WSRT and ATCA data) is/was used to create FITS files with WSRT line data. We investigated the meaning or interpretation of the various FITS header items. The program generates it own descriptors related to velocities and frequencies. For example:

- VEL: Velocity (m/s)
- **VELC: Velocity code**
    - 0=continuum,
    - 1=heliocentric radio
    - 2=LSR radio
    - 3=heliocentric optical
    - 4=LSR optical
- VELR: Velocity at reference frequency (FRQC)
- INST: Instrument code (0=WSRT, 1=ATCA)
- FRQ0: Rest frequency for line (MHz)
- FRQV: Real frequency for line (MHz)
- FRQC: Centre frequency for line (MHz)

One of functions in *nmap* is called *nmawfh.for*. It writes a FITS header using the values in the *nmap* descriptors.

The value of *CRVAL3* is set to *FRQV* if the velocity code is one of combinations of optical and radio velocity with heliocentric or local standard of rest reference systems (i.e. RHEL, RLSR, OHEL, OLSR).

The value of *CRPIX3* is equal to *FRQV* -lowest frequency divided by the channel separation. 'lowest frequency' is the frequency of the input channel with the lowest frequency.

- The value for FITS keyword VEL= is equal to *nmap* descriptor VEL, the centre velocity in m/s
- The value for FITS keyword VELR= is equal to *nmap* descriptor VELR, the Reference velocity
- The value for FITS keyword FREQR= is equal to *nmap* descriptor FRQC, the Reference frequency (Hz)
- The value for FITS keyword FREQ0= is equal to *nmap* descriptor FRQ0, the Rest frequency (Hz)

```
VEL             !CENTRE VELOCITY (M/S)
VELCODE         !VELOCITY CODE
VELR            !REFERENCE VELOCITY (M/S)
FREQR           !REFERENCE FREQUENCY (HERTZ)
FREQ0           !REST FREQUENCY (HERTZ)
```

3.2.5 WCSLIB in a GIPSY task

GIPSY (Groningen Image Processing SYstem) is one of the oldest image processing and data analysis systems. Python can be used to create GIPSY tasks. The Kapteyn Package is integrated in GIPSY. Here we give a small example how to use both.

Assuming you have a data set with three axes and the last axis is the spectral axis, the next script is a very small GIPSY program that asks the user for the name of this set and then calculates the optical velocities for a number of pixels in the neighborhood of the reference pixel (CRPIX3).

GIPSY data have a descriptor which contains FITS header items (e.g. CRVAL1=) and GIPSY specific keywords but not only attached to the set but also to subsets (slices) of the data. Not only planes or lines can have their own header but even pixels can. The script below reads its information from top level (which hosts the global description of the data cube itself):

```
#!/usr/bin/env python
from gipsy import *
from kapteyn import wcs

init()

while True:
   try:
      set = Set(usertext('INSET=', 'Input set'))
      break
   except:
      reject('INSET=', 'Cannot open set')

proj = wcs.Projection(set).sub((3,))
s = "Ref. freq at that pixel: %f Hz" % (set['CRVAL3'],)
anyout(s)
s = "Velocity: %f m/s" % (set['DRVAL3'],)
anyout(s)

crpix = set['CRPIX3']

proj2 = proj.spectra('VOPT-F2W')
for i in range(-2,+3):
   world = proj2.toworld((crpix+i,))[0]/1000.0    #  to world coordinates
   anyout(str(world)+' km/s')

finis()
```

This little GIPSY task simulates the functionality of GIPSY task *COORDS* which lists world coordinates for data slices. The two most important differences between this task and *COORDS* are:

- With WCSLIB it is simple to change the output velocity to radio or apparent radial by changing the spectral translation.
- The Python interface to WCSLIB prepares the GIPSY header information to give correct barycentric or lsrk velocities (i.e. it also converts the frequency increment to the barycentric or lsrk system).

Read more about GIPSY tasks written in Python in Python recipes for GIPSY

References

CHAPTER 4

Module reference

4.1 Module Celestial

This document describes functions from the Python module *celestial* (celestial.py) which provides a programmer with a basic set of routines to transform a world coordinate in a given sky system into a world coordinate of another system assuming zero proper motion, parallax, and recessional velocity.

The most important function builds a matrix for conversions of positions between sky systems, celestial reference systems and epochs of the equinox. This function is called `skymatrix()` and it can be used in the following contexts:

- Implicit, in module *wcs*, using the *Transformation* class as in:

```
world_eq = (192.25, 27.4)    # FK4 coordinates of galactic pole
tran = wcs.Transformation("equatorial fk4_no_e B1950.0", "galactic")
print tran(world_eq)
```

- As stand alone utility in scripts or in an interactive Python session. Usually one uses function `sky2sky()` to transform longitudes and latitudes:

```
M = celestial.sky2sky( (celestial.eq, celestial.fk5), celestial.gal,
                       (0,0,1.0), (10,20,20) )
```

- Hidden in the *topixel()* and *toworld()* methods in module *wcs*. There the sky system is read from a (FITS) header and the sky system for which we want the transformed coordinates is set with attribute *skyout* of the projection object.

See also:

Tutorial material:

- Background information module celestial which contains many examples with source code.

4.1.1 Sky definitions

A sky definition can consist of a *sky system*, a *reference system*, an *equinox* and an *epoch of observation*. It is either a string or it is a tuple with one or more elements. It can also be a single element. The elements in a tuple representing a sky- or reference system are symbols from the table below. For a string, the parts of the string representing a sky- or reference system are minimal matched against the strings in the table below. The match is case insensitive.

Sky systems

Symbol	String	Description
eq, *equatorial*	EQUATORIAL	Equatorial coordinates (u03B1, u03B4), See also next table with reference systems
ecl, *ecliptic*	ECLIPTIC	Ecliptic coordinates (u03BB, u03B2) referred to the ecliptic and mean equinox
gal, *galactic*	GALACTIC	Galactic coordinates (lII, bII)
sgal, *supergalactic*	SUPERGALAC-TIC	De Vaucouleurs Supergalactic coordinates (sgl, sgb)

Reference systems

Symbol	String	Description
fk4	FK4	Mean place pre-IAU 1976 system. FK4 is the old barycentric (i.e. w.r.t. the common center of mass) equatorial coordinate system, which should be qualified by an Equinox value. For accurate work FK4 coordinate systems should also be qualified by an Epoch value. This is the *epoch of observation*.
fk4_no_e	FK4_NO_E, FK4-NO-E	The old FK4 (barycentric) equatorial system but without the *E-terms of aberration*. This coordinate system should also be qualified by both an Equinox and an Epoch value.
fk5	FK5	Mean place post IAU 1976 system. Also a barycentric equatorial coordinate system. This should be qualified by an Equinox value (only).
icrs	ICRS	The International Celestial Reference System, for optical data realized through the Hipparcos catalog. By definition, ICRS is not an equatorial system, but it is very close to the FK5 (J2000) system. No Equinox value is required.
j2000, *dynj2000*	DYNJ2000	This is an equatorial coordinate system based on the mean dynamical equator and equinox at epoch J2000. The dynamical equator and equinox differ slightly compared to the equator and equinox of FK5 at J2000 and the ICRS system. This system need not be qualified by an Equinox value

Note: Reference systems are stored in FITS headers under keyword *RADESYS=*.

Note: Standard in FITS: RADESYS defaults to IRCS unless EQUINOX is given alone, in which case it defaults to FK4 prior to 1984 and FK5 after 1984.

EQUINOX defaults to 2000 unless RADESYS is FK4, in which case it defaults to 1950.

Note: In routines dealing with sky definitions tne names are minimal matched against a list with full names.

Epochs for the equinox and epoch of observation

An epoch can be set in various ways. The options are distinguished by a prefix. Only the ‘B’ and ‘J’ epochs can be negative.

Prefix	Epoch
B	Besselian epoch. Example: `'B 1950'`,`'b1950'`,`'B1983.5'`,`'-B1100'`
J	Julian epoch. Example: `'j2000.7'`,`'J 2000'`,`'-j100.0'`
JD	Julian date. This number of days (with decimals) that have elapsed since the initial epoch defined as noon Universal Time (UT) Monday, January 1, 4713 BC in the proleptic Julian calendar Example: `'JD2450123.7'`
MJD	The Modified Julian Day (MJD) is the number of days that have elapsed since midnight at the beginning of Wednesday November 17, 1858. In terms of the Julian day: MJD = JD - 2400000.5 Example: `'mJD 24034'`,`'MJD50123.2'`
RJD	The Reduced Julian Day (RJD): Julian date counted from nearly the same day as the MJD, but lacks the additional offset of 12 hours that MJD has. It therefore starts from the previous noon UT or TT, on Tuesday November 16, 1858. It is defined as: RJD = JD - 2400000 Example: `'rJD50123.2'`,`'Rjd 23433'`
F	Various FITS formats: • DD/MM/YY Old FITS format. Example: `'F29/11/57'` • YYYY-MM-DD FITS format. Example: `'F2000-01-01'` • YYYY-MM-DDTHH:MM:SS FITS format with date and time. Example: `'F2002-04-04T09:42:42.1'`

Epoch of observation.

Reference system FK4 is not an inertial system. It is slowly rotating and positions are further away from the true mean places if the date of observation is greater than B1950. FK5 is an inertial system. If we convert coordinates from FK4 to FK5, the accuracy of the FK5 position can be improved if we know the date of the observation. So in all transformations where a conversion between FK4 and FK5 is involved, an epoch of observation can be part of the sky definition. Note that this also involves a conversion between galactic coordinates and equatorial, FK5 coordinates because that conversion is done in steps and one step involves FK4.

To be able to distinguish an equinox from an epoch of observation, an epoch of observation is followed by an underscore character and some arbitrary characters to indicate that it is a special epoch (e.q. “B1960_OBS”). Only the underscore is obligatory.

Note: If a sky definition is entered as a string, there cannot be a space between the prefix and the epoch, because a space is a separator for the parser in `celestial.skyparser()`.

Note: An *epoch of observation* is either the second epoch in your input or or the epoch string has a suffix ‘_’ which may be followed by arbitrary characters (e.g. “B1963.5_OBS”).

Input Examples

Input string	Description	Remarks
“eq”	Equatorial, ICRS	ICRS because no reference system and no equinox is given.
“Eclip”	Ecliptic, ICRS	Ecliptic coordinates
“ecl fk5”	Ecliptic, FK5	Ecliptic coordinates with a non default reference system
“GALACtic”	Galactic II	Minimal match is case insensitive
“s”	Supergalactic	Shortest string to identify system.
“fk4”	Equatorial, FK4	Only a reference system is entered. Sky system is assumed to be equatorial
“B1960”	Equatorial, FK4	Only an equinox is given. This is a date before 1984 so FK4 is assumed. Therefore the sky system is equatorial
“EQ, fk4_no_e, B1960”	Equatorial, FK4 no e-terms	Sky system, reference system, and an equinox
“EQ, fk4-no-e, B1960”	Equatorial, FK4 no e-terms	Same as above but underscores replaced by hyphens.
“fk4,J1983.5_OBS”	Equatorial, FK4 + epobs	FK4 with an epoch of observation. Note that only the underscore is important.
“J1983.5_OBS”	Equatorial, FK4 + epobs	Only a date of observation. Then reference system FK4 is assumed.
“EQ,fk4,B1960, B1983.5_O”	Equatorial, FK4 + epobs	A complete description of an equatorial system.
“B1983.5_O fk4 B1960,eq”	Equatorial, FK4 + epobs	The same as above, showing that the order of the elements are unimportant.

Code examples

To show that one can use both the tuple and the string representation of a system, we use both for the same system and compare a transformed position. The result should be 0 for both coordinates.

```
>>> world_eq = numpy.array([192.25, 27.4])      # FK4 coordinates of galactic pole
>>> tran1 = wcs.Transformation("equatorial fk4_no_e B1950.0", "galactic")
>>> tran2 = wcs.Transformation((wcs.equatorial, wcs.fk4_no_e, 'B1950.0'), wcs.galactic)
>>> print tran1(world_eq)-tran2(world_eq)
[ 0.  0.]
```

4.1.2 Module level data

skyrefsystems An object from class *skyrefset* which is a container with a list with systems and two dictionaries with systems.

```
>>> for s in skyrefsystems.skyrefs_list:
>>>    print s.fullname, s.description, s.idnum
```

For programmers who need to access the id’s of the sky and reference systems: External modules can set their own variables. Here are some examples how one can do this.

Example with copy of celestial’s variables:

- `eq = celestial.eq`
- `ec = celestial.ecl`
- `ga = celestial.gal` etc.

Example with minimal match:

- `eq = celestial.skyrefsystems.minmatch2skyref('EQUA')[0].idnum`
- `ec = celestial.skyrefsystems.minmatch2skyref('ecli')[0].idnum`

Read this as: get the object for which a minimal match is found. Item [0] is the object (the other is the number of times a match is found). The 'idnum' is the integer for which we can identify a system.

Or use the equivalent with method *`skyrefset.minmatch2id()`*:

- `eq = celestial.skyrefsystems.minmatch2id('EQUA')`
- `ec = celestial.skyrefsystems.minmatch2id('ecli')`

Example with full name (case sensitive!):

- `eq = celestial.skyrefsystems.fullname2id('EQUATORIAL')`
- `ec = celestial.skyrefsystems.fullname2id('ECLIPTIC')`

4.1.3 Classes

class `kapteyn.celestial.`**`skyrefsys`** (*fullname*, *idnum*, *description*, *refsystem*)

Class creates an object that describes a sky- or reference system. This module initializes a set of systems. They are accessible through methods in class `celestial.skyrefset`

Parameters

- **`fullname`** (*`String`*) – Complete name to identify the system, e.g. *"EQUATORIAL"*
- **`idnum`** (*`Integer`*) – A unique integer to identify the system
- **`description`** (*`String`*) – A short description of the system
- **`refsystem`** (*`Boolean`*) – Is this system a reference system?

Attributes:

`fullname`
A string to identify a system, e.g. "EQUATORIAL".

`idnum`
A unique integer to identify the system.

`description`
A string to describe the system.

`refsystem`
If *True* then this system is a reference system. Else it is a sky system.

class `kapteyn.celestial.`**`skyrefset`**

A container with sky- and reference system objects from class `celestial.skyrefsys`. It is used to initialize variables that can be used as identifiers for sky- or reference systems. Applications can use its methods to retrieve information given an integer identifier or (part of) a string.

For example when we want a list with all the supported systems then type:

```
>>> for s in skyrefsystems.skyrefs_list:
>>>    print s.fullname, s.description, s.idnum
```

`append` (*skyrefsys*)

Parameters **`skyrefsys`** (Instance of class *`skyrefsys`*) – Append this system to the list with supported systems

Returns A unique integer id which can be used to identify a system.

`minmatch2skyref`(*s*)
Return the relevant skyrefsys object with the number of times it is matched or return None if nothing was found.

Parameters **`s`** (*`String`*) – Part of the string name of a system

Returns Instance of class *`skyrefsys`* and the number of times that the input string gives a match.

`minmatch2id`(*s*)
From the found skyrefsys object corresponding to string *s*, return the idnum attribute. Case insensitive minimal match is used to find the sky- or reference system. Return None if there was no match or more than one match.

Parameters **`s`** (*`String`*) – Part of the string name of a system

Returns Instance of class *`skyrefsys`* or None if there was not a match or more than one match.

`fullname2id`(*fullname*)
This is the fastest method to get an integer id from a string which represents a sky system or a reference system. Note that the routine is case sensitive because it uses the full names as keys in a dictionary. The parameter *fullname* therefore must be in in capitals!

Parameters **`fullname`** (*`String`*) – The full descriptive name of a system e.g. "EQUATORIAL"

Returns Integer id of the found system or *None* if nothing was found.

`id2skyref`(*idnum*)
Given an integer id of a system, return the corresponding system as an instance of class *`skyrefsys`*. Usually the calling environment will deal with the attributes of this object, for instance to write a short description of the system.

Parameters **`idnum`** (*`Integer`*) – Integer id of a system

Returns Instance of class *`skyrefsys`* or None if there was not a corresponding system.

`id2fullname`(*idnum*)
Given an integer id of a system, return the full name of the corresponding system.

Parameters **`idnum`** (*`Integer`*) – Integer id of a system

Returns Full name (e.g. "EQUATORIAL") of the corresponding system or an empty string if nothing was found.

`id2description`(*idnum*)
Given an integer id of a system, return the description of the corresponding system.

Parameters **`idnum`** (*`Integer`*) – Integer id of a system

Returns A short description of the corresponding system or an empty string if nothing was found.

Attributes:

`skyrefs_list`
The list with systems

`skyrefs_id`
A dictionary with the systems and with id's as keys

skyrefs_fullname
A dictionary with the systems and with full names as keys

Examples Next short script shows how to get a list with sky systems and how to use methods of this class to get data for a system if an (integer) id is found:

```
from kapteyn.celestial import skyrefsystems

for s in skyrefsystems.skyrefs_list:
   print s.fullname, s.description, s.idnum
   i = s.idnum
   print "Full name using id2fullname:", skyrefsystems.id2fullname(i)
   print "Description using id2description:", skyrefsystems.id2description(i)
   print "id of %s with minimal match: %d" % \
         (s.fullname[:3], skyrefsystems.minmatch2skyref(s.fullname[:3])[0].idnum)
   print "id of %s with minimal match, alternative: %d" % \
         (s.fullname[:3], skyrefsystems.minmatch2id(s.fullname[:3]))
   print "id of %s with full name: %d" % \
         (s.fullname[:3], skyrefsystems.fullname2id(s.fullname))
```

4.1.4 Core Functions

`kapteyn.celestial.`**`skyparser`**`(`*skyin*`)`

Parse a string, tuple or single integer that represents a sky definition. A sky definition can consist of a *sky system*, a *reference system*, an *equinox* and an *epoch of observation*. See also the description at *Sky definitions*. The elements in the string are separated by a comma or a space. The order of the elements is not important. The string is converted to a tuple by `celestial.parseskydefs()`.

The parser is used in function `celestial.skymatrix()` and `celestial.sky2sky()`. External applications can use this function to check whether user input is valid.

Definitions in strings are usually used to define output sky definitions in prompts or on command lines. Applications can use integer id's for the sky- and reference systems. These integer id's are global constants See also *Sky systems* and *Reference systems*.

The sky system and reference system strings are minimal matched (case INsensitive) with the strings in the table in the documentation at *Sky systems* and *Reference systems*.

For the epoch syntax read the documentation at *Epochs for the equinox and epoch of observation*. Note that an epoch of observation is either a second epoch in the string (the first is always the equinox) or the epoch string has a suffix '_' which may be follwed by arbitrary characters.

Parameters **`skyin`** (*`String, tuple or integer`*) – Represents a sky definition. See examples.

Returns A tuple with the 'coded' system where strings for sky- and reference systems are replaced by integer id's. Missing values are filled in with defaults.

If an error occurred then an exception will be raised.

Raises

`ValueError` From `celestial.parseskydefs()`:

- *Empty string!*
- *Too many items for sky definition!*
- *... is ambiguous sky or reference system!*

- *... is not a valid epoch or sky/ref system!*

From this function:

- *Sky definition is not a string nor a tuple!*
- *Too many elements in sky definition (max. 4)!*
- *Two sky systems given!*
- *Two reference systems given!*
- *Invalid number for sky- or reference system!*
- *Cannot determine the sky system!*
- *Input contains an element that is not an integer or a string!*

Examples

```
>>> print celestial.skyparser("B1983.5_O fk4 B1960,eq")
(0, 4, 1960.0, 1983.5)
```

```
>>> print celestial.skyparser("su")
(3, None, None, None)
```

```
>>> print celestial.skyparser("supergal")
(3, None, None, None)
```

Notes This is the parser for a sky definition. In this definition one can specify the sky system, the reference system, an equinox and an epoch of observation if the reference system is fk4. The order of these elements is not important.

The rules for the defaults are:

- What if the sky system is not defined? If there is a reference system then we assume it is equatorial (could have been ecliptic).
- If there no sky system and no reference system but there is an equinox, assume sky system is equatorial (could have been ecliptic).
- If there no sky system and no reference system and no equinox but there is an epoch of observation, assume sky system is equatorial.
- Assume we have a sky system. What if there is no reference system? Standard in FITS: RADESYS (i.e our reference system) defaults to IRCS unless EQUINOX is given alone, in which case it defaults to FK4 prior to 1984 and FK5 after 1984.
- Assume we have a sky system and a reference system and the sky system was ecliptic or equatorial. What if we don't have an equinox? Standard in FITS: EQUINOX defaults to 2000 unless RADESYS is FK4, in which case it defaults to 1950.
- We have one item to address and that is the epoch of observation. This epoch of observation only applies to the reference systems FK4 and FK4_NO_E. In 'Representations of celestial coordinates in FITS' (Calabretta & Greisen) we read that all reference systems are allowed for both equatorial- and ecliptic coordinates, except FK4-NO-E, which is only allowed for equatorial coordinates. If FK4-NO-E is given in combination with an ecliptic sky system then silently FK4 is assumed.

`kapteyn.celestial.`**`skymatrix`**`(`*skyin*, *skyout*`)`

Create a transformation matrix to be used to transform a position from one sky system to another (including epoch transformations). For a description of the sky definitions see *Sky definitions*.

Parameters

- **skyin** (*Integer or tuple with one to four elements*) – One of the supported sky systems or a tuple for equatorial systems which are identified with an equinox an reference system. This is the sky system from which you want to transform to another sky system (*skyout*).
- **skyout** – The destination sky system

Returns Three elements:

- The transformation matrix *M* for the transformation of positions in (x,y,z) as in *XYZskyout = M * XYZskyin*
- followed by 'None' or a tuple with the e-term vector belonging input epoch.
- followed by *None* or a tuple with the e-term vector belonging to the output epoch.

See also notes below.

Notes The reference systems FK4 and FK4_NO_E are special. We consider FK4 as a catalog position where the **e-terms** are included. So besides a transformation matrix, this function should also return a flag for the addition or removal of e-terms. This flag is either *None* or the e-term vector which depends on the epoch.

The structure of the output then is as follows: `M, (A1,A2,A3), (A4,A5,A6)` where:

- *M*: The 3x3 transformation matrix
- *(A1,A2,A3)* or *None*: for adding or removing e-terms in the input sky system using this e-term vector *(A1,A2,A3)*.
- *(A4,A5,A6)* or *None*: for adding or removing e-terms in the output sky system using this e-term vector *(A4,A5,A6)*.

This function is the main function of this module. It calls function *skyparser()* for the parsing of the input and *rotmatrix()* to get the rotation matrix. There utility function *sky2sky()* transforms a sequence of longitudes and latitudes from one sky system to another. It is a valuable tool for experiments in an interactive Python session.

Examples Some examples of transformations between sky systems using either strings or tuples. We advise to use strings which is more safe then using variables from celestial (which can be accidentally replaced by other values). Note that for transformations where FK4 is involved, the matrix is followed by a vector with e-terms.

```
>>> from kapteyn import celestial
>>> print skymatrix(celestial.gal,(celestial.eq,"j2000",celestial.fk5))
(matrix([[-0.05487554,  0.49410945, -0.86766614],
         [-0.8734371 , -0.44482959, -0.19807639],
         [-0.48383499,  0.74698225,  0.45598379]]),
      None,
      None)
```

```
>>> print skymatrix(celestial.fk4, celestial.fk5)
(matrix([[  9.99925679e-01,  -1.11814832e-02,  -4.85900382e-03],
         [  1.11814832e-02,   9.99937485e-01,  -2.71625947e-05],
         [  4.85900377e-03,  -2.71702937e-05,   9.99988195e-01]]),
      (-1.6255503575995309e-06,
         -3.1918587795578522e-07,
         -1.3842701121066153e-07), None)
```

```
>>> print skymatrix("eq,B1950.0,fk4_no_e","eq,B1950.0,fk4")
(matrix([[ 1.,  0.,  0.],
         [ 0.,  1.,  0.],
         [ 0.,  0.,  1.]]),
      None,
      (-1.6255503575995309e-06,
         -3.1918587795578522e-07,
         -1.3842701121066153e-07))
```

```
>>> print skymatrix("eq b1950 fk4 j1983.5", "eq J2000 fk5")
(matrix([[  9.99925679e-01,  -1.11818698e-02,  -4.85829658e-03],
         [  1.11818699e-02,   9.99937481e-01,  -2.71546879e-05],
         [  4.85829648e-03,  -2.71721706e-05,   9.99988198e-01]]),
      (-1.6255503575995309e-06,
         -3.1918587795578522e-07,
         -1.3842701121066153e-07),
      None)
```

```
>>> print skymatrix("eq J2000 fk4 F1984-1-1T0:30", "eq J2000 fk5")
(matrix([[  1.00000000e+00,  -5.45185721e-06,  -3.39404820e-07],
         [  5.45185723e-06,   1.00000000e+00,   2.24950276e-08],
         [  3.39404701e-07,  -2.24971595e-08,   1.00000000e+00]]),
      (-1.6181121582090453e-06,
         -3.4112123324131958e-07,
         -1.4789407828956555e-07),
      None)
```

See *Epochs for the equinox and epoch of observation* for the possible epoch formats.

`kapteyn.celestial.`**`sky2sky`**(*skyin*, *skyout*, *lons*, *lats*)
Utility function to facilitate command line use of skymatrix.

Parameters

- **skyin** (See function *skymatrix()*) – The input sky definition
- **skyout** (See function *skymatrix()*) – The output sky definition
- **lons** (*Floating point number(s), scalar, list or tuple*) – Input longitude(s)
- **lats** (*Floating point number(s), scalar, list or tuple*) – Input latitude(s)

Returns Matrix. One position per row. See example below how to extract rows, columns and elements from this matrix.

Example Interactive Python session:

```
>>> from kapteyn import celestial
>>> M = celestial.sky2sky( (celestial.eq, celestial.fk5), celestial.gal,
                          (0,0,1.0), (10,20,20) )
>>> M
matrix([[ 102.6262244 ,  -50.83256452],
        [ 106.78021643,  -41.25289649],
        [ 107.9914125 ,  -41.49143448]])
>>> M[2,0]
107.99141249678289
>>> M[0]         # Extract first transformed long, lat
```

```
             matrix([[ 102.6262244 ,  -50.83256452]])
             >>> M[:,1]        # Extract second column with latitudes
             matrix([[-50.83256452],
                     [-41.25289649],
                     [-41.49143448]])
```

Notes This function illustrates the core use of module *celestial*. First it converts the input of world coordinates into a matrix. This matrix is converted to spatial positions (X,Y,Z) with function *longlat2xyz()*. The function *dotrans()* transforms these positions (X,Y,Z) to positions (X2,Y2,Z2) in the output sky system. Then the function *xyz2longlat()* converts these positions into longitudes and latitudes and finally a matrix with these values is returned:

```
             lonlat = n.array( [(lons,lats)] )
             xyz = longlat2xyz(lonlat)
             xyz2 = dotrans(skymatrix(skyin, skyout), xyz)
             newlonlats = xyz2longlat(xyz2)
             return newlonlats
```

`kapteyn.celestial.`**`epochs`**(*spec*)

Flexible epoch parser. The functions in this module have different input parameters (Julian epoch, Besselian epochs, Julian dates) because the algorithms came from different sources. What we needed was a routine that could convert a string which represents a date in various formats, to values for a Julian epoch, Besselian epochs and a Julian date. This function returns these value for any valid input date.

For the epoch syntax read the documentation at *Epochs for the equinox and epoch of observation*. Note that an epoch of observation is either a second epoch in the string (the first is always the equinox) or the epoch string has a suffix '_' which may be follwed by arbitrary characters.

Parameters **`spec`** (*`String`*) – An epoch specification (see below)

Returns Calculated corresponding **Besselian epoch**, **Julian epoch** and **Julian date**. Return in order: *B, J, JD*

Reference Various sources listing Julian dates.

Notes

Examples Some checks:

```
             >>> celestial.epochs('F2008-03-31T8:09')   # should return:
                 (2008.2474210134737, 2008.2459673739454, 2454556.8395833336)
             >>> celestial.epochs('F2007-01-14T13:18:59.9')
                 (2007.0378545262108, 2007.0364267212976, 2454115.0548599539)
             >>> celestial.epochs("j2007.0364267212976")
                 (2007.0378545262108, 2007.0364267212976, 2454115.0548599539)
             >>> celestial.epochs("b2007.0378545262108")
                 (2007.0378545262108, 2007.0364267212976, 2454115.0548599539)
```

4.1.5 Utility functions

`kapteyn.celestial.`**`JD`**(*year*, *month*, *day*)

Calculate Julian day number (Julian date)

Parameters

- **`year`** (*`Integer`*) – Year (nnnn)
- **`month`** (*`Integer`*) – Month (nn)

- **day** (*Floating point number*) – Day (nn.n...)

Returns Julian day number *jd*.

Reference Meeus, Astronomical formula for Calculators, 2nd ed, 1982

Notes Months start at 1. Days start at 1. The Julian day begins at Greenwich mean noon, i.e. at 12h. So Jan 1, 1984 at 0h is entered as *JD(1984,1,1)* and Jan 1, 1984 at 12h is entered as *JD(1984,1,1.5)*

There is a jump at *JD(1582,10,15)* caused by a change of calendars. For dates after 1582-10-15 one enters a date from the Julian calendar and before this date you enter a date from the Gregorian calendar.

Examples

- Julian date of JD reference: `print celestial.JD(-4712,1,1.5) ==> 0.0`
- The first day of 1 B.C.: `print celestial.JD(0,1,1) ==> 1721057.5`
- Last day before Gregorian reform: `print celestial.JD(1582,10,4) ==> 2299159.5`
- First day of Gregorian reform: `print celestial.JD(1582,10,15) ==> 2299170.5`
- Half a day later: `print celestial.JD(1582,10,15.5) ==> 2299161.0`
- Unix reference: `print celestial.JD(1970,1,1) ==> 2440587.5`

`kapteyn.celestial.`**`lon2hms`**(*a*, *prec=1*, *delta=None*, *tex=False*)
Convert an angle in degrees to **hours, minutes, seconds** format.

Parameters

- **a** (*Floating point number*) – Angle (in degrees) for which we want to create a formatted text label.
- **prec** (*Integer*) – The required number of decimals in the seconds part of output. If a value is omitted, then the default is 1.
- **delta** (*None* or a floating point number) – If one labels world coordinates along an axis then the default labels are in hours, minutes and seconds with some decimal number. This is probably not want you want if the step size between subsequent positions is for example an integer number of degrees or minutes. Then you want labels showing only hours or hours and minutes. This function tries to find out whether this is the case (given a value for *delta*) or not. If so, a minimum length label is returned.
- **tex** (*Boolean*) – The default is *False*. If set to *True*, the string is formatted in LaTeX. Such labels can be plotted in, for example, Matplotlib.

Returns Formatted string representing the input angle.

Notes Longitudes are forced into the range, 360 deg. and then converted to hours, minutes and seconds.

Examples Format a position in hms and dms:

```
>>> ra = 359.9999
>>> dec = 0.0000123
>>> print celestial.lon2hms(ra),  celestial.lat2dms(dec)
    00h 00m  0.0s +00d 00m  0.0s
>>> print celestial.lon2hms(ra, 2),  celestial.lat2dms(dec, 2)
    23h 59m 59.98s +00d 00m  0.04s
```

```
>>> print celestial.lon2hms(ra, 4),  celestial.lat2dms(dec, 4)
23h 59m 59.9760s +00d 00m  0.0443s
```

kapteyn.celestial.**lat2dms**(*a*, *prec=1*, *delta=None*, *tex=False*)

Convert an angle in degrees into the **degrees, minutes, seconds** format assuming it was a latitude. Its value should be in the range -90 to 90 degrees

Parameters

- **a** (*Floating point number*) – Angle (in degrees) for which we want to create a formatted text label.
- **prec** (*Integer*) – The required number of decimals in the seconds part of output. If a value is omitted, then the default is 1.
- **delta** (*None* or a floating point number) – If one labels world coordinates along an axis then the default labels are in degrees, minutes and seconds with some decimal number. This is probably not want you want if the step size between subsequent positions is for example an integer number of degrees or minutes. Then you want labels showing only degrees or degrees and minutes. This function tries to find out whether this is the case (given a value for *delta*) or not. If so, a minimum length label is returned.
- **tex** (*Boolean*) – The default is *False*. If set to *True*, the string is formatted in LaTeX. Such labels can be plotted in, for example, Matplotlib.

Returns Formatted string representing the input angle or a string with '#' characters indicating that the input was out of range.

Notes The HMS and DMS format should be treated differently because their ranges in world coordinates are different. Longitudes should be in range of (0,360) degrees. So -10 deg is in fact 350 deg. and 370 deg is in fact 10 deg. Latitudes range from -90 to 90 degrees. Then 91 degrees is in fact 89 degrees but at a longitude that is separated 180 deg. from the stated longitude. But we don't have control over the longitudes here so the only thing we can do is reject the value and return a dummy string.

kapteyn.celestial.**lon2dms**(*a*, *prec=1*, *delta=None*, *tex=False*)

Convert an angle in degrees to **degrees, minutes, seconds** format, assuming the input is a longitude but not associated with an equatorial system.

Parameters

- **a** (*Floating point number*) – Angle (in degrees) for which we want to create a formatted text label
- **prec** (*Integer*) – The required number of decimals in the seconds part of output If a value is omitted, then the default is 1.
- **delta** (*None* or a floating point number) – If one labels world coordinates along an axis then the default labels are in hours, minutes and seconds with some decimal number. This is probably not want you want if the step size between subsequent positions is for example an integer number of degrees or minutes. Then you want labels showing only degrees or degrees and minutes. This function tries to find out whether this is the case (given a value for *delta*) or not. If so, a minimum length label is returned.
- **tex** (*Boolean*) – The default is *False*. If set to *True*, the string is formatted in LaTeX. Such labels can be plotted in, for example, Matplotlib.

Returns Formatted string representing the input angle.

Notes Longitudes are forced into the range 0, 360 deg. and then converted to hours, minutes and seconds.

Examples Format a longitude to dms:

```
>>> print celestial.lon2dms(167.342, 4)
   167d 20m 31.2000s
>>> print celestial.lon2dms(-10, 4)
   350d  0m  0.0000s
```

`kapteyn.celestial.`**`JD2epochBessel`**`(`*JD*`)`

Convert a Julian date to a Besselian epoch.

Parameters **`JD`** (*`Floating point number`*) – Julian date (e.g. 2445700.5)

Returns Besselian epoch (e.g. 1983.9)

Reference Standards Of Fundamental Astronomy,

http://www.iau-sofa.rl.ac.uk/2003_0429/sofa/epb.html

Notes e.g. 2445700.5 -> 1983.99956681

One *Tropical Year* is 365.242198781 days and JD(1900) = 2415020.31352

If we know the JD then the Besselian epoch can be calculated with:

`BE = B[1900 + (JD - 2415020.31352)/365.242198781]`

Expression corresponds to the IAU SOFA expression in the reference with: `2451545-36524.68648 = 2415020.31352`

`kapteyn.celestial.`**`epochBessel2JD`**`(`*Bepoch*`)`

Convert a Besselian epoch to a Julian date

Parameters **`Bepoch`** (*`Floating point number`*) – Besselian epoch in format nnnn.nn

Returns Julian date

Reference See: *`JD2epochBessel()`*

Notes e.g. 1983.99956681 converts into 2445700.5 It's the inverse of *`JD2epochBessel()`*

`kapteyn.celestial.`**`JD2epochJulian`**`(`*JD*`)`

Convert a Julian date to a Julian epoch

Parameters **`JD`** (*`Floating point number`*) – Julian date

Returns Julian epoch

Reference Standards Of Fundamental Astronomy,

http://www.iau-sofa.rl.ac.uk/2003_0429/sofa/epj.html

Notes e.g. `2445700.5 converts into 1983.99863107` Assuming years of exactly 365.25 days, we can calculate a Julian epoch from a Julian date. Expression corresponds to IAU SOFA routine 'epj'

`kapteyn.celestial.`**`epochJulian2JD`**`(`*Jepoch*`)`

Convert a Julian epoch to a Julian date

Parameters **`Jepoch`** (*`Floating point number`*) – Julian epoch (in format nnnn.nn)

Returns Julian date

Reference See *`JD2epochJulian()`*

Notes e.g. `1983.99863107 converts into 2445700.5` It's the inverse of function JD2epochJulian `kapteyn.celestial.`**`obliquity1980`**`(`*jd*`)`

What is the obliquity of the ecliptic at this Julian date? (IAU 1980 model)

Parameters **`jd`** (*`Floating point number`*) – Julian date

Returns Mean obliquity in degrees

Reference Explanatory Supplement to the Astronomical Almanac, P. Kenneth Seidelmann (ed), University Science Books (1992), Expression 3.222-1 (p114).

Notes The epoch is entered in Julian date and the time is calculated w.r.t. J2000.

The obliquity is the angle between the mean equator and ecliptic, or, between the ecliptic pole and mean celestial pole of date

`kapteyn.celestial.`**`obliquity2000`**`(`*jd*`)`

What is the obliquity of the ecliptic at this Julian date? (IAU model 2000)

Parameters **`jd`** (*`Floating point number`*) – Julian date

Returns Mean obliquity in degrees

Reference Fukushima, T. 2003, AJ, 126,1 Kaplan, H., 2005, The IAU Resolutions on Astronomical Reference Systems, Time Scales, and Earth Rotation Models, United States Naval Observatory circular no. 179, http://aa.usno.navy.mil/publications/docs/Circular_179.pdf (page 44)

Notes The epoch is entered in Julian date and the time is calculated w.r.t. J2000.

The obliquity is the angle between the mean equator and ecliptic, or, between the ecliptic pole and mean celestial pole of date.

`kapteyn.celestial.`**`IAU2006precangles`**`(`*epoch*`)`

Calculate IAU 2000 precession angles for precession from input epoch to J2000.

Parameters **`epoch`** (*`Floating point number`*) – Julian epoch of observation.

Returns Angles u03B6 (zeta), z, u03B8 (theta) in degrees to setup a rotation matrix to transform from J2000 to input epoch.

Reference Capitaine N. et al., IAU 2000 precession A&A 412, 567-586 (2003)

Notes Input are Julian epochs! `T = (jd-2451545.0)/36525.0` Combined with `jd = Jepoch-2000.0)*365.25 + 2451545.0` gives: (see module code at function *epochJulian2JD(epoch)*) `T = (epoch-2000.0)/100.0`

This function should be updated as soon as there are IAU2006 adopted angles to replace the angles used in this function.

`kapteyn.celestial.`**`Lieskeprecangles`**`(`*jd1*, *jd2*`)`

Calculate IAU 1976 precession angles for a precession of epoch corresponding to Julian date jd1 to epoch corresponds to Julian date jd2.

Parameters

- **`jd1`** (*`Floating point number`*) – Julian date for start epoch
- **`jd2`** (*`Floating point number`*) – Julian date for end epoch

Returns Angles u03B6 (zeta), z, u03B8 (theta) degrees

Reference Lieske,J.H., 1979. Astron.Astrophys.,73,282. equations (6) & (7), p283.

Notes The ES (Explanatory Supplement to the Astronomical Almanac) lists for a IAU1976 precession from 1984, January 1d0h to J2000 the angles in **arcsec**: `xi_a=368.9985, ze_a=369.0188 and th_a=320.7279` Using the functions in this module, this can be calculated by applying:

```
>>> jd1 = celestial.JD(1984,1,1)
>>> jd2 = celestial.JD(2000,1,1.5)
>>> print celestial.Lieskeprecangles(jd1, jd2)
   (0.10249958598931658, 0.10250522534285664, 0.089091092843880629)
>>> print [a*3600 for a in angles]
    [368.99850956153966, 369.01881123428387, 320.72793423797026]
```

The function returns values in degrees, while literature values often are listed in seconds of arc.

Lieske's fit belongs to the so called Quasi-Linear Types Below a table with the precision (according to IAU SOFA):

- 1960AD to 2040AD: < 0.1"
- 1640AD to 2360AD: < 1"
- 500BC to 3000AD: < 3"
- 1200BC to 3900AD: > 10"
- < 4200BC or > 5600AD: > 100"
- < 6800BC or > 8200AD: > 1000"

`kapteyn.celestial.`**`Newcombprecangles`**`(`*epoch1*, *epoch2*`)`

Calculate precession angles for a precession in FK4, using Newcomb's method (Woolard and Clemence angles)

Parameters

- **epoch1** (*Floating point number*) – Besselian start epoch
- **epoch2** (*Floating point number*) – Besselian end epoch

Returns Angles u03B6 (zeta), z, u03B8 (theta) degrees

Reference ES 3.214 p.106

Notes Newcomb's precession angles for old catalogs (FK4), see ES 3.214 p.106. Input are **Besselian epochs**! Adopted accumulated precession angles from equator and equinox at B1950 to 1984 January 1d 0h according to ES (table 3.214.1, p 107) are: `zeta=783.7092, z=783.8009 and theta=681.3883` The Woolard and Clemence angles (derived in this routine) are: `zeta=783.70925, z=783.80093 and theta=681.38830` (see same ES table as above).

This routine found (in seconds of arc): `zeta,z,theta = 783.709246271 783.800934641 681.388298284` for `t1 = 0.1 and t2 = 0.133999566814` using the lines in the next example.

Examples From an interactive Python session:

```
>>> b1 = 1950.0
>>> b2 = celestial.epochs("F1984-01-01")[0]
>>> print [x*3600 for x in celestial.Newcombprecangles(be1, be2)]
    [783.70924627097793, 783.80093464073127, 681.38829828393466]
```

4.1.6 Rotation matrices

`kapteyn.celestial.`**`MatrixEqB19502Gal`**`()`

Create matrix to convert equatorial fk4 coordinates (without e-terms) to IAU 1958 lII,bII system of galactic coordinates.

Parameters None

Results 3x3 Matrix M as in XYZgal = M * XYZb1950

Reference

1. Blaauw, A., Gum C.S., Pawsey, J.L., Westerhout, G.: 1958,
2. Monthly Notices Roy. Astron. Soc. 121, 123,
3. Blaauw, A., 2007. Private communications.

Notes Original definitions from 1.:

- The new north galactic pole lies in the direction alpha = 12h49m (192.25 deg), delta=27.4 deg (equinox 1950.0).
- The new zero of longitude is the great semicircle originating at the new north galactic pole at the position angle theta = 123 deg with respect to the equatorial pole for 1950.0.
- Longitude increases from 0 to 360 deg. The sense is such that, on the galactic equator increasing galactic longitude corresponds to increasing Right Ascension. Latitude increases from -90 deg through 0 deg to 90 deg at the new galactic pole.

Given the RA and Dec of the galactic pole, and using the Euler angles scheme:

```
M = rotZ(a3).rotY(a2).rotZ(a1)
```

We first rotate the spin vector of the XY plane about an angle a1 = ra_pole and then rotate the spin vector in the XZ plane (i.e. around the Y axis) with an angle a2=90-dec_pole to point it in the right declination.

Now think of a circle with the galactic pole as its center. The radius is equal to the distance between this center and the equatorial pole. The zero point now is on the circle and opposite to this pole.

We need to rotate along this circle (i.e. a rotation around the new Z-axis) in a way that the angle between the zero point and the equatorial pole is equal to 123 deg. So first we need to compensate for the 180 deg of the current zero longitude, opposite to the pole. Then we need to rotate about an angle 123 deg but in a way that increasing galactic longitude corresponds to increasing Right Ascension which is opposite to the standard rotation of this circle (note that we rotated the original X axis about 192.25 deg). The last rotation angle therefore is a3=+180-123:

```
M = rotZ(180-123.0)*rotY(90-27.4)*rotZ(192.25)
```

The composed rotation matrix is the same as in Slalib's 'ge50.f' and the matrix in eq. (32) of Murray (1989).

`kapteyn.celestial.`**`MatrixGal2Sgal`**`()`

Transform galactic to supergalactic coordinates

Parameters None

Returns Matrix M as in XYZsgal = M * XYZgal

Reference Lahav, O., The supergalactic plane revisited with the Optical Redshift Survey Mon. Not. R. Astron. Soc. 312, 166-176 (2000)

Notes The Supergalactic equator is conceptually defined by the plane of the local (Virgo-Hydra-Centaurus) supercluster, and the origin of supergalactic longitude is at the intersection of the supergalactic and galactic planes. (de Vaucouleurs)

North SG pole at l=47.37 deg, b=6.32 deg. Node at l=137.37, sgl=0 (inclination 83.68 deg).

Older references give for he position of the SG node 137.29 which differs from 137.37 deg in the official definition.

For the rotation matrix we chose the scheme *Rz.Ry.Rz* Then first we rotate about 47.37 degrees along the Z-axis followed by a rotation about 90-6.32 degrees is needed to set the pole to the right declination. The new plane intersects the old one at two positions. One of them is l=137.37, b=0 (in galactic coordinates). If we want this to be sgl=0 we have to rotate this plane along the new Z-axis about an angle of 90 degrees. So the composed rotation matrix is:

```
M = Rotz(90)*Roty(90-6.32)*Rotz(47.37)
```

kapteyn.celestial.**MatrixEq2Ecl**(*epoch*, *S1*)

Calculate a rotation matrix to convert equatorial coordinates to ecliptical coordinates

Parameters

- **epoch** (*Floating point number*) – Epoch of the equator and equinox of date
- **S1** (*Integer*) – equatorial system to determine if one entered epoch in B or J coordinates.

Returns 3x3 Matrix M as in XYZecl = M * XYZeq

Reference Representations of celestial coordinates in FITS, Calabretta. M.R., & Greisen, E.W., (2002) Astronomy & Astrophysics, 395, 1077-1122. http://www.atnf.csiro.au/people/mcalabre/WCS/ccs.pdf

Notes

1. The origin for ecliptic longitude is the vernal equinox. Therefore the coordinates of a fixed object is subject to shifts due to precession. The rotation matrix uses the obliquity to do the conversion to the wanted ecliptic coordinates. So we always need to enter an epoch. Usually this is J2000, but it can also be the epoch of date. The additional reference system indicates whether we need a Besselian or a Julian epoch.

2. In the FITS paper of Calabretta and Greisen (2002), one observes the following relations to FITS:

   -Keyword RADESYSa sets the catalog system FK4, FK4-NO-E or FK5 This applies to equatorial and ecliptical coordinates with the exception of FK4-NO-E.

   -FK4 coordinates are not strictly spherical since they include a contribution from the elliptic terms of aberration, the so-called e-terms which amount to max. 343 milliarcsec. FITS paper: *'Strictly speaking, therefore, a map obtained from, say, a radio synthesis telescope, should be regarded as FK4-NO-E unless it has been appropriately re-sampled or a distortion correction provided. In common usage, however, CRVALia for such maps is usually given in FK4 coordinates. In doing so, the e-terms are effectively corrected to first order only.'*. (See also ES, eq. 3.531-1 page 170.

   -Keyword EQUINOX sets the epoch of the mean equator and equinox.

   -Keyword EPOCH is often used in older FITS files. It is a deprecated keyword and should be replaced by EQUINOX. It does not require keyword RADESYS. From its value we derive whether the reference system is FK4 or FK5 (the marker value is 1984.0)

   -Ecliptic coordinates require the epoch of the equator and equinox of date. This will be taken as the time of observation rather than EQUINOX.

   FITS paper: *'The time of observation may also be required for other astrometric purposes in addition to the usual astrophysical uses, for example, to specify when the mean place was correct in accounting for proper motion, including "fictitious" proper motions in the conversion between the FK4 and FK5 systems. The old *DATE-OBS* keyword may be used for this purpose. However, to provide a more convenient specification we here introduce the new keyword MJD-OBS'.*

So MJD-OBS is the modified Julian Date (JD - 2400000.5) of the start of the observation.

3. Equatorial to ecliptic transformations use the time dependent obliquity of the equator (also known as the obliquity of the ecliptic). Again, start with:

```
M = rotZ(0).rotX(eps).rotZ(0) = E.rotX(eps).E = rotX(eps)
```

In fact this is only a rotation around the X axis

`kapteyn.celestial.`**`FK42FK5Matrix`**`(`*t=None*`)`

Create a matrix to precess from B1950 in FK4 to J2000 in FK5 following to Murray's (1989) procedure.

Parameters **t** (*Floating point number*) – Besselian epoch as epoch of observation.

Returns 3x3 matrix M as in XYZfk5 = M * XYZfk4

Reference

- Murray, C.A. The Transformation of coordinates between the systems B1950.0 and J2000.0, and the principal galactic axis referred to J2000.0, Astronomy and Astrophysics (ISSN 0004-6361), vol. 218, no. 1-2, July 1989, p. 325-329.
- Poppe P.C.R.,, Martin, V.A.F., Sobre as Bases de Referencia Celeste SitientibusSerie Ciencias Fisicas

Notes Murray precesses from B1950 to J2000 using a precession matrix by Lieske. Then applies the equinox correction and ends up with a transformation matrix *X(0)* as given in this function.

In Murray's article it is proven that using the procedure as described in the article, `r_fk5 = X(0).r_fk4` for extra galactic sources where we assumed that the proper motion in FK5 is zero. This procedure is independent of the epoch of observation. Note that the matrix is not a rotation matrix.

FK4 is not an inertial coordinate frame (because of the error in precession and the motion of the equinox. This has consequences for the proper motions. e.g. a source with zero proper motion in FK5 has a fictitious proper motion in FK4. This affects the actual positions in a way that the correction is bigger if the epoch of observation is further away from 1950.0 The focus of this library is on data of which we do not have information about the proper motions. So for positions of which we allow non zero proper motion in FK5 one needs to supply the epoch of observation.

Examples Print the difference between the rotation matrix for 1970 and 1980:

```
>>> M1 = celestial.FK42FK5Matrix(1970)
>>> M2 = celestial.FK42FK5Matrix(1980)
>>> M2 - M1
matrix([[ -2.64546940e-10,  -1.15396722e-07,   2.11108953e-07],
        [  1.15403817e-07,  -1.29040234e-09,   2.36016437e-09],
        [ -2.11125281e-07,  -5.60232514e-10,   1.02585540e-09]])
```

`kapteyn.celestial.`**`ICRS2FK5Matrix`**`()`

Create a rotation matrix to convert a position from ICRS to fk5, J2000

Parameters None

Returns 3x3 rotation matrix M as in XYZfk5 = M * XYZicrs

Reference Kaplan G.H., The IAU Resolutions on Astronomical Reference systems, Time scales, and Earth Rotation Models, US Naval Observatory, Circular No. 179

Notes Return a matrix that converts a position vector in ICRS to FK5, J2000. We do not use the first or second order approximations given in the reference, but use the three rotation matrices from the same paper to obtain the exact result:

```
M =  rotX(-eta0)*rotY(xi0)*rotZ(da0)
```

eta0 = -19.9 mas, xi0 = 9.1 mas and da0 = -22.9 mas

`kapteyn.celestial.`**`ICRS2J2000Matrix`**`()`

Return a rotation matrix for conversion of a position in the ICRS to the dynamical reference system based on the dynamical mean equator and equinox of J2000.0 (called the dynamical J2000 system)

Parameters None

Returns Rotation matrix to transform positions from ICRS to dyn J2000

Reference

- Hilton and Hohenkerk (2004), Astronomy and Astrophysics 413, 765-770
- Kaplan G.H., The IAU Resolutions on Astronomical Reference systems, Time scales, and Earth Rotation Models, US Naval Observatory, Circular No. 179

Notes Return a matrix that converts a position vector in ICRS to Dyn. J2000. We do not use the first or second order approximations given in the reference, but use the three rotation matrices to obtain the exact result:

```
M = rotX(-eta0)*rotY(xi0)*rotZ(da0)
```

eta0 = -6.8192 mas, xi0 = -16.617 mas and da0 = -14.6 mas

`kapteyn.celestial.`**`JMatrixEpoch12Epoch2`**`(`*Jepoch1*, *Jepoch2*`)`

Precession from one epoch to another in the fk5 system. It uses *`Lieskeprecangles()`* to calculate the precession angles.

Parameters

- **`Jepoch1`** (*`Floating point number`*) – Julian start epoch
- **`Jepoch2`** (*`Floating point number`*) – Julian epoch to precess to.

Returns 3x3 rotation matrix M as in XYZepoch2 = M * XYZepoch1

Reference Seidelman, P.K., 1992. Explanatory Supplement to the Astronomical Almanac. University Science Books, Mill Valley. 3.214 p 106

Notes The precession matrix is:

```
M = rotZ(-z).rotY(+theta).rotZ(-zeta)
```

`kapteyn.celestial.`**`BMatrixEpoch12Epoch2`**`(`*Bepoch1*, *Bepoch2*`)`

Precession from one epoch to another in the fk4 system. It uses *`Newcombprecangles()`* to calculate the precession angles.

Parameters

- **`Bepoch1`** (*`Floating point number`*) – Besselian start epoch
- **`Bepoch2`** (*`Floating point number`*) – Besselian epoch to precess to.

Returns 3x3 rotation matrix M as in XYZepoch2 = M * XYZepoch1

Reference Seidelman, P.K., 1992. Explanatory Supplement to the Astronomical Almanac. University Science Books, Mill Valley. 3.214 p 106

Notes The precession matrix is:

```
M = rotZ(-z).rotY(+theta).rotZ(-zeta)
```

`kapteyn.celestial.`**`IAU2006MatrixEpoch12Epoch2`**(*epoch1*, *epoch2*)

Create a rotation matrix for a precession based on IAU 2000/2006 expressions, see function *`IAU2006precangles()`*

Parameters

- **`epoch1`** (*`Floating point number`*) – Julian start epoch
- **`epoch2`** (*`Floating point number`*) – Julian epoch to precess to.

Returns Matrix to transform equatorial coordinates from epoch1 to epoch2 as in XYZepoch2 = M * XYZepoch1

Reference Capitaine N. et al.: IAU 2000 precession A&A 412, 567-586 (2003)

Notes Note that we apply this precession only to equatorial coordinates in the system of dynamical J2000 coordinates. When converting from ICRS coordinates this means applying a frame bias. Therefore the angles differ from the precession Fukushima-Williams angles (IAU 2006)

The precession matrix is:

```
M = rotZ(-z).rotY(+theta).rotZ(-zeta)
```

`kapteyn.celestial.`**`MatrixEpoch12Epoch2`**(*epoch1*, *epoch2*, *S1*, *S2*, *epobs=None*)

Helper function for *`skymatrix()`*. It handles precession and the transformation between **equatorial** systems. This function includes also conversions between reference systems.

Parameters

- **`epoch1`** (*`Floating point number`*) – Epoch belonging to system S1 depending on the reference system either Besselian or Julian.
- **`epoch2`** – Epoch belonging to system S2 depending on the reference system either Besselian or Julian.
- **`S1`** (*`Integer`*) – Input reference system
- **`S2`** (*`Integer`*) – Output rreferencesystem
- **`epobs`** (*`Floating point number`*) – Epoch of observation. Only valid for conversions between FK4 and FK5.

Returns Rotation matrix to transform a position in one of the reference systems *S1* with *epoch1* to an equatorial system with equator and equinox at *epoch2* in reference system *S2*.

Notes Return matrix to transform equatorial coordinates from *epoch1* to *epoch2* in either reference system FK4 or FK5. Or transform from epoch, FK4 or FK5 to ICRS or J2000 vice versa. Note that each transformation between FK4 and one of the other reference systems involves a conversion to FK5 and therefore the epoch of observation will be involved.

Note that if no systems are entered and the one epoch is > 1984 and the other < 1984, then the transformation involves both sky reference systems FK4 and FK5.

Examples Calculate rotation matrix for a conversion between FK4, epoch 1940 to FK5, epoch 1960, while the date of observation was 1950.

```
>>> from kapteyn import celestial
>>> celestial.MatrixEpoch12Epoch2(1940, 1960, celestial.fk4, celestial.fk5, 1950)
matrix([[  9.99988107e-01,  -4.47301372e-03,  -1.94362889e-03],
        [  4.47301372e-03,   9.99989996e-01,  -4.34712255e-06],
        [  1.94362889e-03,  -4.34680782e-06,   9.99998111e-01]])
```

4.1.7 Functions related to E-terms

`kapteyn.celestial.`**`getEterms`**(*epoch*)

Compute the E-terms (elliptic terms of aberration) for a given epoch.

Parameters **epoch** (*Floating point number*) – A **Besselian** epoch

Returns A tuple containing the e-terms vector *(DeltaD,DeltaC,DeltaC.tan(e0))*

Reference Seidelman, P.K., 1992. Explanatory Supplement to the Astronomical Almanac. University Science Books, Mill Valley

Notes The method is described on page 170/171 of the ES. One needs to process the e-terms for the appropriate epoch This routine returns the e-term vector for arbitrary epoch.

`kapteyn.celestial.`**`addEterms`**(*xyz*, *a=None*)

Add the elliptic component of annual aberration when the result must be a catalogue fk4 position.

Parameters

- **xyz** (*NumPy (n,2) matrix*) – Cartesian position(s) converted from lonlat = [(a1,d1),(a2,d2), ..., (an,dn)] –> xyz = [(x1,y1,z1), (x2,y2,z2), ..., (xn,yn,zn)]
- **a** (*Tuple with 3 floating point numbers*) – E-terms vector (as returned by getEterms()) If input *a* is omitted (i.e. *a == None*), the e-terms for 1950 will be substituted.

Result **Apparent place**, NumPy (n,2) matrix

Reference

- Seidelman, P.K., 1992. Explanatory Supplement to the Astronomical Almanac. University Science Books, Mill Valley.
- Yallop et al, Transformation of mean star places, AJ, 1989, vol 97, page 274
- Stumpff, On the relation between Classical and Relativistic Theory of Stellar Aberration, Astron, Astrophys, 84, 257-259 (1980)

Notes There is a so called ecliptic component in the stellar aberration. This vector depends on the epoch at which we want to process these terms. It corresponds to the component of the earth's velocity perpendicular to the major axis of the ellipse in the ecliptic. The E-term corrections are as follows. A catalog FK4 position include corrections for elliptic terms of aberration. These positions are apparent places. For precession and/or rotations to other sky systems, one processes only mean places. So to get a mean place, one has to remove the E-terms vector. The ES suggests for the removal to use a decompositions of the E-term vector along the unit circle to get the approximate new vector, which has almost the correct angle and has almost length 1. The advantage is that when we add the E-term vector to this new vector, we obtain a new vector with the original angle, but with a length unequal to 1, which makes it suitable for closure tests. However, the procedure can be made more rigorous: For the subtraction we subtract the E-term vector from the start vector and normalize it afterwards. Then we have an exact new angle (opposed to the approximation in the ES). The procedure to go from a vector in the mean place system to a vector in the system of apparent places is a bit more complicated: Find a value for lambda so that the current vector is adjusted in length so that adding the e-term vector gives a new vector with length 1. This is by definition the new vector with the right angle. For more information, see the background information in Background information module celestial.

`kapteyn.celestial.`**`removeEterms`**(*xyz*, *a=None*)

Remove the elliptic component of annual aberration when this is included in a catalogue fk4 position.

Parameters

- **`xyz`** (*`NumPy (n,2) matrix`*) – Cartesian position(s) converted from lonlat = [(a1,d1),(a2,d2), ..., (an,dn)] –> xyz = [(x1,y1,z1), (x2,y2,z2), ..., (xn,yn,zn)]
- **`a`** (*`Tuple with 3 floating point numbers`*) – E-terms vector (as returned by getEterms()) If input a is omitted (== *None*), the e-terms for 1950 will be substituted.

Result **Mean place**, NumPy (n,2) matrix

Notes Return a new position where the elliptic terms of aberration are removed i.e. convert a apparent position from a catalog to a mean place. The effects of ecliptic aberration were included in the catalog positions to facilitate telescope pointing. See also notes at 'addEterms'.

4.2 SciPy modules

Mainly for convenience, SciPy's modules scipy.ndimage.filters and scipy.ndimage.interpolation have been included in the Kapteyn Package as `kapteyn.filters` and `kapteyn.interpolation`. In this way users of the package do not need to have all of SciPy installed, of which only a few functions are currently used. To these modules the SciPy license applies which is compatible with the Kapteyn Package's license.

Function `map_coordinates()` from module `interpolation` has slightly been modified. If the source array contains one or more NaN values, and the *order* argument is larger than 1, the unmodified function will return an array with all NaN values. The modification prevents this by replacing NaN values by nearby finite values.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

[Ref1] *Representations of world coordinates in FITS* http://www.atnf.csiro.au/people/mcalabre/WCS/wcs.pdf Greisen E.W. and Calabretta M.R.

[Ref2] *Representations of celestial coordinates in FITS* http://www.atnf.csiro.au/people/mcalabre/WCS/ccs.pdf Calabretta M.R. and Greisen E.W.

[Ref3] *Representations of spectral coordinates in FITS* http://www.atnf.csiro.au/people/mcalabre/WCS/scs.pdf E. W. Greisen, M. R. Calabretta, F. G. Valdes, and S. L. Allen

[FITS] *Definition of the Flexible Image Transport System (FITS), FITS Standard Version 3.0* http://fits.gsfc.nasa.gov/fits_standard.html FITS Working Group , Commission 5: Documentation and Astronomical Data, International Astronomical Union

[Alp] Alper, Joseph S., Gelb, Robert I., *Standard Errors and Confidence Intervals in Nonlinear Regression: Comparison of Monte Carlo and Parametric Statistics*, J. Phys. Chem., 1990, 94 (11), pp 4747–4751 (Journal of Physical Chemistry)

[And] Andrae, R, *Error estimation in astronomy: A guide*, arXiv:1009.2755v3 [astro-ph.IM] 29 Oct 2010

[Bev] Bevington, Philip R. , *Data Reduction and Error Analysis for the Physical Sciences*, 1969, McGraw-Hill

[BRo] Bevington, P.R., Robinson D.K., *Data Reduction and Error Analysis for the Physical Sciences*, Version 2.0 RLM (23 August 2003)

[Clu] Clutton-Brock, *Likelihood Distributions for Estimating Functions When Both Variables Are Subject to Error*, Technometrics, Vol. 9, No. 2 (May, 1967), pp. 261-269

[Ds1] DeSerio, R., *Statistical Analysis of Data for PHY48803L*, Advanced Physics Laboratory, University of Florida (version 1) Local copy: `statmain-florida.pdf`

[Ds2] DeSerio, R., *Statistical Analysis of Data for PHY48803L*, Advanced Physics Laboratory, University of Florida (version 2) Local copy: `statmain.pdf`

[Ds3] DeSerio, R., *Regression Algebra*, Local copy: `matproof_statmain.pdf`

[Mar] Marel, P. van der, Franx, M., *A new method for the identification of non-gaussian line profiles in elliptical galaxies*. A.J., **407** 525-539, 1993 April 20

[Mas] Massey, F. J. *The Kolmogorov-Smirnov Test for Goodness of Fit.*, Journal of the American Statistical Association, Vol. 46, No. 253, 1951, pp. 68-78

[Mkw] Markwardt, C. B. 2008, “Non-Linear Least Squares Fitting in IDL with MPFIT,” in proc. Astronomical Data Analysis Software and Systems XVIII, Quebec, Canada, ASP Conference Series, Vol. 411, eds. D. Bohlender, P.

Dowler & D. Durand (Astronomical Society of the Pacific: San Francisco), p. 251-254 (ISBN: 978-1-58381-702-5) Website: http://purl.com/net/mpfit

[Num] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, 2nd edition, Cambridge University Press, 1992

[Ogr] Ogren, J., Norton, J.R., *Applying a Simple Linear Least-Squares Algorithm to Data with Uncertainties in Both Variables*, J. of Chem. Education, Vol 69, Number 4, April 1992

[Ore] Orear, Jay, *Least squares when both variables have uncertainties*, Am. J. Phys. 50(10), Oct 1982

[Pea] Pearson, K. *On lines and planes of closest fit to systems of points in space*. Philosophical Magazine 2:559-572, 1901. A copy of this article can be found at: http://stat.smmu.edu.cn/history

[Scr] Schreier, Franz, *Optimized implementations of rational approximations for the Voigt and complex error function*, Journal of Quantitative Spectroscopy & Radiative Transfer 112 (2011) 1010-1025

[Sch] Schwarz, U.J., *Analysis of an Observed Function into Components, using its Second Derivative*, Bull. Astr. Inst. Netherlands, 1968, 19 405-413 (local copy)

[Sha] Shao, J., *Jackknifing Weighted Least Squares Estimators*, Journal of the Royal Statistical Society. Series B (Methodological), Vol. 51, No. 1(1989), pp. 139-156

[Yor] York, D. *Least-squares fitting of a straight line*, Canadian Journal of Physics. Vol. 44, p.1079, 1966

[Vog] Vogelaar, M.G.R., XGAUPROF, local copy

[Wil] Williamson, *Least-squares fitting of a straight line*, J.A., Can. J. Phys, 1968, 46, 1845-1847

[Wol] Wolberg, J., *Data Analysis Using the Method of Least Squares*, 2006, Springer

[Aoki1] **Aoki**, S., Soma, M., Kinoshita, H., Inoue, K., 1983. *Conversion matrix of epoch B 1950.0 FK4-based positions of stars to epoch J 2000.0 positions in accordance with the new IAU resolutions*, Astron. Astrophys. 128, p.263-267, 1983, ADS Abstract Service 1983

[Aoki2] **Aoki**, S. et al, 1986. *The Conversion from the B1950 FK4 Based Position to the J2000 Position of Celestial Objects*, Astrometric Techniques: IAU SYmp:109 Florida p.123, 1986, ADS Abstract Service 1986

[Blaauw] **Blaauw**, A.; Gum, C. S.; Pawsey, J. L.; Westerhout, G., 1959, *Note: Definition of the New I.A.U. System of Galactic Co-Ordinates* Astrophysical Journal, vol. 130, p.702, ADS Abstract Service 1959

[Brouw] **Brouw**, W.N., 1974. *Synthesis Radio Telescope Project; The SRT Reduction Program*, Internal Technical Report ITR 78 about the Standard Reduction Program for the Westerbork Synthesis Radio Telescope, Astr. Observatory, Leiden, Netherlands

[Calabr] **Calabretta**, M.R., Greisen, E.W., 2002 *Representations of celestial coordinates in FITS* Astronomy and Astrophysics, v.395, p.1077-1122 (2002). PDF version at http://www.atnf.csiro.au/people/mcalabre/WCS/

[Corwin] **Corwin**, H. G.; de Vaucouleurs, A.; de Vaucouleurs, G., 1994. *Southern Galaxy Catalogue (SGC)*, VizieR On-line Data Catalog: VII/116. Originally published in: 1985MAUTx...4....1C, RC3 - Third Reference Catalog of Bright Galaxies

[Cotton] **Cotton**, W. D.; Condon, J. J.; Arbizzani, E. , 1999. *Arcsecond Positions of UGC Galaxies*, The Astrophysical Journal Supplement Series, Volume 125, Issue 2, p.409-412 ADS Abstract Service 1999

[Diebel] **Diebel**, J, 2006. *Representing Attitude: Euler Angles, Quaternions, and Rotation Vectors* (`local copy`)

[Hering] **Hering**, R.; Walter, H. G., 1998. *Updating of B1950 radio star positions by means of J2000 calibrators.* International Spring Meeting of the Astronomische Gesellschaft: The message of the angles - astrometry from 1798 to 1998, p.198 - 200, http://www.astro.uni-bonn.de/~pbrosche/aa/acta/vol03/acta03_198.html

[Hilton] **Hilton**, J.L.; Hohenkerk, C. Y., 2004. *Rotation matrix from the mean dynamical equator and equinox at J2000.0 to the ICRS* Astronomy and Astrophysics, v.413, p.765-770 (2004). ADS Abstract Service 2004

[Kaplan] **Kaplan**, G.H., 2005. *The IAU Resolutions on Astronomical Reference systems, Time scales, and Earth Rotation Models*, US Naval Observatory, Circular No. 179, http://aa.usno.navy.mil/publications/docs/Circular_179.pdf

[Lieske1] **Lieske**, J. H.; Lederle, T.; Fricke, W.; Morando, B., 1977. *Expressions for the precession quantities based upon the IAU /1976/ system of astronomical constants*, Astronomy and Astrophysics, vol. 58, no. 1-2, June 1977, p. 1-16 ADS Abstract Service 1977

[Lieske2] **Lieske**, J.H., 1979. *Precession matrix based on IAU 1976 system of astronomical constants* Astronomy and Astrophysics, vol. 73, no. 3, Mar. 1979, p.282-284, ADS Abstract Service 1979

[Murray] **Murray**, C.A., 1989. *The transformation of coordinates between systems of B1950.0 and J2000.0 and the principal galactic axes referred to J2000.0*, Astron. Astrophys, 218, p.325-329, ADS Abstract Service 1989

[Poppe] **Poppe** P.C.R., Martin, V.A.F., 2005. *Sobre as Bases de Referencia Celeste (On the celestial reference frames)*, Sitientibus Serie Ciencias Fisicas 01: 30-38 (2005), http://www2.uefs.br/depfis/sitientibus/vol1/Vera_Main-SPSS.pdf

[Scott] **Scott** F.P., Hughes J.A. , *Computation of Apparent Places for the Southern Reference Star Program*, The Astronomical Journal, Vol 69, Number 5, 1964, p.368-371, ADS Abstract Service 1964

[Seidel] **Seidelmann**, P.K., 1992. *Explanatory Supplement to the Astronomical Almanac*, University Science Books

[Smart] **Smart**, W.M., 1931, Sixth ed. 1977, reprint 1990. *Textbook on Spherical Astronomy*, Sixth Edition, Revised by R.M. Green, Cambridge University Press

[Smith] **Smith**, C. A.; Kaplan, G. H.; Hughes, J. A.; Seidelmann, P. K.; Yallop, B. D.; Hohenkerk, C. Y., 1989. *Mean and apparent place computations in the new IAU system. I - The transformation of astrometric catalog systems to the equinox J2000.0. II - Transformation of mean star places from FK4 B1950.0 to FK5 J2000.0 using matrices in 6-space*, ADS Abstract Service 1989II

[Soma] **Soma**, M., Aoki, S. 1990. *Transformation of the Mean Place from FK4 to FK5*, Inertial Coordinate System Of/ Sky: IAU SYMP.141 p.131, 1989, ADS Abstract Service 1990

[Wallace1] **Wallace**, P. T., 1994. *The SLALIB Library* , Astronomical Data Analysis Software and Systems III, A.S.P. Conference Series, Vol. 61, 1994, Dennis R. Crabtree, R.J. Hanisch, and Jeannette Barnes, eds., p.481.

[Wallace2] **Wallace, P. (chair)**, IAU SOFA, IAU, 2007, *SOFA Tools for Earth Attitude* sofa_pn.pdf and also: ADS Abstract Service 1994

[Wallace3] **Wallace**, P. T., 2005. *SLALIB – Positional Astronomy Library 2.5-3 Programmer's Manual*, Manual

[Yallop] **Yallop**, B. D.; Hohenkerk, C. Y.; Smith, C. A.; Kaplan, G. H.; Hughes, J. A.; Seidelmann, P. K., 1989. *Mean and apparent place computations in the new IAU system II. Transformation of mean star places from FK4 B1950.0 to FK5 J2000.0 using matrices in 6-space*, Astron. Journal, 97, Number 1, January 1989, ADS Abstract Service 1989 III

[Aipsmemo] AIPS memo 27 Non-Linear Coordinate Systems in AIPS (Eric W. Greisen, NRAO)

Python Module Index

Index