PyAstronomy Documentation

Release 0.15.2

PyA group

Sep 07, 2020
1 About PyAstronomy 3
2 PyAstronomy’s sub-packages 15
3 Spotlights 351
4 PyAstronomy core 353
5 Indices and tables 371
Bibliography 373
Python Module Index 375
Index 377
PyAstronomy (PyA) is a collection of astronomy related packages hosted on github.
1.1 What is it?

PyAstronomy (PyA) is a collection of astronomy related packages. The aim of PyAstronomy is to provide a collection of packages, which fulfill a certain standard both in code and documentation quality. In this way, we (the PyA group) want to ensure that PyA packages constitute an enjoyable tool for enhancing the efficiency of our/your work and not another source of painful experiences.

PyA emerged from the work of some people at the Hamburger Sternwarte. It has not been designed to cover some particular field, but we integrated tools, which we found useful during our work. Other people have different needs and also ours are evolving. Therefore, we expect PyA to grow both by our own contributions and, hopefully, contributions from others.

1.2 Installation and source code of PyAstronomy

PyAstronomy is available from the Python Package Index (PyPI) using pip. Sources, bug tracking, and opportunities for contributions are available on github.

1.2.1 Installation with PyPI and pip

PyAstronomy can be installed via pip

```
Note: Depending on the setup of your Python installation, you may need administrator (root) privileges to install a package.
```

```
pip install PyAstronomy
```

or (if non-Python dependencies are required)
1.2.2 Installation from github via pip

The current development branch can be installed from github via

```
pip install git+https://github.com/sczesla/PyAstronomy.git #egg=PyAstronomy[occult]
```

1.2.3 Installing from source code

PyAstronomy can be installed from the source. Source distributions can be obtained from github. Save it to whatever place you prefer on your system, extract the files, and change into the thus created directory; on linux use, e.g.,:

```
tar xfv PyAstronomy.tar.gz
cd PyAstronomy
```

**Note:** The package and directory name usually contain a version number.

In the directory created by unpacking the tar-ball, you find a script called `setup.py`. This script will do the work of installing the package for you. Execute it by typing:

```
python setup.py [--with-ext] install
```

**Note:** `--with-ext` is an optional flag. If specified, the installer will try to build non-Python extension. Building the extensions requires a fortran compiler.

**Note:** Depending on the setup of your Python installation, you may need administrator (root) privileges to install a package. The default path for installing packages is the `site-packages` directory of your Python installation. You can modify this target location by using "`python setup.py install --home=XYZ`", where `XYZ` is your preferred installation path. Note that this path has to be added to your `PYTHONPATH` environment variable if `XYZ` is a nonstandard path.

1.2.4 Building the documentation

PyAstronomy is distributed including documentation. The latest documentation is available via readthedocs. To build the documentation yourself, change into the directory where you installed (or unpacked) PyA. Change into the subdirectory named `doc` (may not be the first level). In this directory, you find a Makefile, which is responsible for building the documentation.

**Note:** To build the documentation you need to have installed the Sphinx package. In addition, PyAstronomy must be installed.

The HTML documentation is built by using:

```
make html
```
1.3 Members of the PyA group

PyAstronomy is a collaborative effort started by SC and SS during a (really!) bad-weather observing run high-up on island X.

- Stefan Czesla (SC)
- Sebastian Schröter (SS)

Valuable and highly appreciated contributions to the project have meanwhile been provided by the following authors:

- Christian P. Schneider
- Klaus F. Huber
- Fabian Pfeifer
- Daniel Thaagaard Andreasen
- Mathias Zechmeister (GLS)

1.4 PyAstronomy licensing (MIT)

PyAstronomy and all its submodules, if not explicitly stated otherwise, are distributed under the MIT license:

```
Copyright (c) 2011, PyA group

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```
1.5.1 From 0.12.0 to 0.13.0

<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
</table>
| pyasl    | New functionality  
• Added Beta Sigma noise estimate |

1.5.2 From 0.11.0 to 0.12.0

<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
</table>
| pyasl    | New functionality  
• De Jager 1987 spectral types  
• MMSCETS table |
| pyTiming | Update  
• New GLS implementation from MZ |

1.5.3 From 0.10.0 to 0.11.0

<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyAstronomy</td>
<td>Python 3.x compatibility</td>
</tr>
</tbody>
</table>
| pyasl    | New functionality  
• slidingPolyResOutlier  
• New output/input options for daycnv  
• ‘nanHandling’ for binningx0dt  
• ExoplanetEU2 for more convenient access to exoplanet.eu data  
• Photometric passbands |
| funcFit  | Improved functionality  
• Sorting in parameterSummary |
| pyaC     | New functionality  
• Find zero crossings in discrete data |
1.5.5 From 0.8.1 (or 0.8.0) to 0.9.0

<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyaC</td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>• mtools package</td>
</tr>
<tr>
<td></td>
<td>• trapz integration with interpolated boundaries</td>
</tr>
<tr>
<td>pyasl</td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>• Abundance patterns</td>
</tr>
<tr>
<td>modelSuite</td>
<td><strong>Update</strong></td>
</tr>
<tr>
<td></td>
<td>• Maintenance of transit models</td>
</tr>
<tr>
<td></td>
<td>• Full support of keplerian orbits</td>
</tr>
<tr>
<td>funcFit</td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>• emcee sampling</td>
</tr>
</tbody>
</table>

1.5.6 From 0.7.0 to 0.8.0
## Concerns

<table>
<thead>
<tr>
<th>pyasl</th>
<th><strong>New functionality</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• S-index into RHK conversion</td>
</tr>
<tr>
<td></td>
<td>• Convolution with Gaussian (broadGaussFast)</td>
</tr>
<tr>
<td></td>
<td>• Conversion between atomic number and elemental symbol</td>
</tr>
<tr>
<td></td>
<td>• First ionization energies</td>
</tr>
<tr>
<td></td>
<td>• Conversion between decimal and sexagesimal coordinates</td>
</tr>
<tr>
<td></td>
<td>• Outlier detection based on polynomial</td>
</tr>
<tr>
<td></td>
<td>• write1dFitsSpec</td>
</tr>
<tr>
<td></td>
<td>• Exponentially correlated Gaussian random numbers</td>
</tr>
<tr>
<td></td>
<td>• Conversion from decimal year to gregorian date</td>
</tr>
<tr>
<td></td>
<td>• Teff-color conversion</td>
</tr>
<tr>
<td></td>
<td>• Stellar age</td>
</tr>
<tr>
<td></td>
<td><strong>Update</strong></td>
</tr>
<tr>
<td></td>
<td>• Updated air-vacuum wavelength conversion</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>funcFit</th>
<th><strong>New functionality</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Model description</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pyaGui</th>
<th><strong>New functionality</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Interactive normalization</td>
</tr>
<tr>
<td></td>
<td>• Interactive Gauss/Voigt fit</td>
</tr>
</tbody>
</table>

### 1.5.7 From 0.6.0 to 0.7.0
<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyaC</td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>• Formatted output of 2d matrices</td>
</tr>
<tr>
<td></td>
<td>• Numerical derivatives (diffCFD)</td>
</tr>
<tr>
<td>pyasl</td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>• Cross-correlation algorithm (crosscorRV)</td>
</tr>
<tr>
<td></td>
<td>• Finding extreme point by parabolic fit (quadExtreme)</td>
</tr>
<tr>
<td></td>
<td>• Calculate decimal year (decimalYear)</td>
</tr>
<tr>
<td></td>
<td>• Gaussian instrumental broadening (instrBroadGaussFast)</td>
</tr>
<tr>
<td></td>
<td>• Reading spectrum from fits file</td>
</tr>
<tr>
<td></td>
<td>• Distance module to distance conversion</td>
</tr>
<tr>
<td></td>
<td>• Position angle</td>
</tr>
<tr>
<td>funcFit</td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>• Various bug fixes.</td>
</tr>
<tr>
<td></td>
<td>• Introduced external fitters</td>
</tr>
<tr>
<td></td>
<td>• Multicomponent two-dimensional Gaussian</td>
</tr>
<tr>
<td></td>
<td>• Conditional restrictions</td>
</tr>
<tr>
<td>modelSuite</td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>• Rotational broadening profile</td>
</tr>
</tbody>
</table>

### 1.5.8 From 0.5.0 to 0.6.0
<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcFit</td>
<td><em>New functionality</em></td>
</tr>
<tr>
<td></td>
<td>• GUI tool for model exploration</td>
</tr>
<tr>
<td></td>
<td>• Tutorial on TraceAnalysis</td>
</tr>
<tr>
<td>modelSuite</td>
<td><em>New functionality</em></td>
</tr>
<tr>
<td></td>
<td>• Hydrogen Lyman-alpha line profile</td>
</tr>
<tr>
<td>pyasl</td>
<td><em>New routines</em></td>
</tr>
<tr>
<td></td>
<td>• Signal to Noise Ratio (SNR) estimator</td>
</tr>
<tr>
<td></td>
<td>• Outlier detection</td>
</tr>
<tr>
<td></td>
<td>• INTEP interpolation algorithm</td>
</tr>
<tr>
<td></td>
<td>• Magnitude conversion</td>
</tr>
<tr>
<td></td>
<td>• Access to Baraffe et al. 98 evolutionary tracks</td>
</tr>
<tr>
<td></td>
<td>• Ported IDL Astrolib routines (sunpos, nutate, co_nutate, co_aberration, ct2lst, hadec2altaz, co_refract_forward, co_refract, eq2hor, observatory)</td>
</tr>
<tr>
<td></td>
<td>• Access to NASA Exoplanet Archive</td>
</tr>
<tr>
<td></td>
<td>• Access to exoplanet.eu data base</td>
</tr>
<tr>
<td></td>
<td>• Azimuth to cardinal point conversion</td>
</tr>
<tr>
<td></td>
<td>• Name twilight given solar altitude</td>
</tr>
<tr>
<td></td>
<td>• Calculate angular distance</td>
</tr>
<tr>
<td></td>
<td>• Calculate airmass</td>
</tr>
<tr>
<td></td>
<td>• Transit time table</td>
</tr>
<tr>
<td></td>
<td>• Transit visibility plot</td>
</tr>
<tr>
<td>pyaC</td>
<td><em>New functionality</em></td>
</tr>
<tr>
<td></td>
<td>• Invert index selection</td>
</tr>
</tbody>
</table>

1.5.9 From 0.4.0 to 0.5.0
## Concerns

<table>
<thead>
<tr>
<th>funcFit</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>New functionality</strong></td>
</tr>
<tr>
<td></td>
<td>* Now supports n-dimensional (nD) fitting</td>
</tr>
<tr>
<td></td>
<td>* Tutorial for nD fitting added</td>
</tr>
<tr>
<td></td>
<td>* Obtain confidence intervals via <code>steppar</code> and <code>errorConfInterval</code> methods</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>modelSuite</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>New models</strong></td>
</tr>
<tr>
<td></td>
<td>* Added a spectral model based on Gaussian lines (<code>LLGauss</code>).</td>
</tr>
<tr>
<td></td>
<td>* Added a model to fitting the coordinates of a Kepler ellipse (<code>KeplerEllipseModel</code>).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pyasl</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>New routines</strong></td>
</tr>
<tr>
<td></td>
<td>* Ported <code>baryvel</code> from IDL.</td>
</tr>
<tr>
<td></td>
<td>* A wrapper for numpy’s Singular Value Decomposition (SVD)</td>
</tr>
<tr>
<td></td>
<td>* A function to apply Doppler shift to a spectrum.</td>
</tr>
<tr>
<td></td>
<td>* A function to apply rotational broadening to a spectrum.</td>
</tr>
<tr>
<td></td>
<td>* Ported <code>unred</code> method from IDL.</td>
</tr>
<tr>
<td></td>
<td>* Convert between flux units and photons number.</td>
</tr>
<tr>
<td></td>
<td>* Introduced a <code>smooth</code> function for smoothing 1d data.</td>
</tr>
</tbody>
</table>

### 1.5.10 From 0.3.0 to 0.4.0
<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td>constants</td>
<td><strong>New subpackage</strong> Provides easy access to a number of astronomical constants.</td>
</tr>
</tbody>
</table>
| modelSuite | **New models**  
• Transit modeling relying on Mandel and Agol code.  
**Bug fix**  
• Fixed bug in *RmcL* class (Rossiter-McLaughlin effect) causing wrong models for large parameter lambda. |
| pyasl | **New routines**  
• A binning algorithm.  
• Transit timing routines.  
• F-test implementation.  
• Implementation of Keplerian orbit and solution of Kepler’s equation. |
| pyTiming | Improvement of Generalized-Lomb-Scargle algorithm. |

### 1.5.11 From 0.2.0 to 0.3.0

<table>
<thead>
<tr>
<th>Concerns</th>
<th>What happened</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td>Docstrings should now follow the numpy/scipy standard.</td>
</tr>
</tbody>
</table>
| funcFit | **Naming of models** The package now supports a naming scheme for models. This makes it easier to identify and access variables in composed models.  
**anaMCMCTrace** This new class can be used to analyze the outcome of an MCMC sampling run conveniently.  
**SyncFit** Synchronous model fitting. This new class allows to have different models defined on different abscissa to be fitted together. |

### 1.6 Acknowledging PyAstronomy

If you make use of PyAstronomy (PyA) in your work, we are thankful if you acknowledge it by citing its ASCL entry (see bibtex entry below) and give a link to PyAstronomy (https://github.com/sczesla/PyAstronomy), e.g., as a footnote. PyAstronomy is a collection, so the documentation occasionally points to further papers, which may be cited as well.
Cheers, The PyA group

1.6.1 Bibtex entry

```latex
@MISC{pya,
    author = {(Czesla), Stefan and (Schr"{o}ter), Sebastian and (Schneider), Christian P. and (Huber), Klaus F. and (Pfeifer), Fabian and (Andreasen), Daniel T. and (Zechmeister), Mathias),
    title = "{PyA: Python astronomy-related packages}'',
    keywords = {Software},
    year = "2019",
    month = "Jun",
    eid = {ascl:1906.010},
    pages = {ascl:1906.010},
    archivePrefix = {ascl},
    eprint = {1906.010},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

1.7 Contact us

We appreciate your opinion.

If you intend to send us any comments, remarks, bug reports, wishes, or coarse insults, you can do so using PyAstronomy’s project page on

github

or simply send us an e-mail to

stefan.czesla@hs.uni-hamburg.de

In any case, we will try our best to answer appropriately.
2.1 Python AstroLib

The aim of this project is to bundle a number of helpful rather small-scale routines. This comprises porting the well-known “IDL Astronomy User’s Library” (AstroLib) to Python. As the corresponding IDL code has been in use for decades now, it is “tested by application”. Therefore, we try to stay as close as possible to the IDL implementation. Furthermore, helpful routines will be added.

Note: This is not an attempt to port the entire IDL library at once. We do what we need! Help is appreciated.

2.1.1 Available functionality

Below, you find a list of available functionality roughly sorted by category.

General algorithms

Calculate a Keplerian (two body) orbit

Although the two-body problem has long been solved, calculation the orbit position of a body in an eccentric orbit — maybe a planet — as a function of time is not trivial. The major complication is solving Kepler’s Equation. The classes defined here do this job.

The definitions and most of the formulae used in this class derive from the book “Orbital Motion” by A.E. Roy.

Orbital elements and orientation of the orbit

Orientation of the ellipse in the coordinate system For zero inclination the ellipse is located in the x-y plane. If the eccentricity is increased, the periastron will lie in +x direction. If the inclination is
increased, the ellipse will be rotating around the x-axis, so that +y is rotated toward +z. An increase in Omega corresponds to a rotation around the z-axis so that +x is rotated toward +y. Changing w, i.e., the argument of the periastron, will not change the plane of the orbit, but rather represent a rotation of the orbit in the plane. In particular, the periapsis is shifted in the direction of motion.

**Orbital angular momentum** For all parameters but semi-major axis and orbital period set to zero, the (orbital) angular momentum points into the +z direction. For an inclination of 90 deg (the remaining parameters remaining zero), it points in the -y direction.

**Orientation of the ellipse in the sky** To project the ellipse onto the sky, the coordinate system should be oriented so that the +x direction points North and the +y direction points East (direction of increasing right ascension). The +z axis must be chosen so that the coordinate system becomes right handed. If the line of sight (LOS) points in the +z direction, i.e., the observer is located on the negative z axis, the parameters assume their conventional meaning.

**The ascending and descending nodes** For systems outside the Solar System, the ascending node is the point where the body “crosses” the plane of the sky away from the observer. Likewise, the descending node is the point where the plane is crossed with the body approaching the observer. For the coordinate system described above and a value of zero for the longitude of the ascending node, the latter is in the North and rotates toward East (i.e., +y) when the longitude of the ascending node is increased.

**The argument and longitude of periapsis** The argument of periapsis is the angle between the ascending node and the periapsis of the body measured in the direction of motion. For exoplanets with circular orbits, for which no well-defined periapsis exists, the argument of periapsis is often chosen so that time of periapsis and central transit time coincide. For the planet, this is the case if the argument of periapsis is -90 deg. However, in the exoplanet literature, the argument of periapsis often refers to the stellar orbit (see, e.g., Pollacco et al. 2008, MNRAS 385, 1576-1584, Sect. 3.2.1). In this case, the corresponding value is +90 deg. The so-called longitude of the periapsis is given by the sum of the longitude of the ascending node and the argument of periapsis.

**Example: Invoking the solver for Kepler’s Equation**

This example demonstrates how to use the solver for Kepler’s Equation.

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Instantiate the solver
ks = pyasl.MarkleyKESolver()

# Solves Kepler's Equation for a set
# of mean anomaly and eccentricity.
# Uses the algorithm presented by
# Markley 1995.
M = 0.75
e = 0.3
print("Eccentric anomaly: ", ks.getE(M, e))
```

**Example: Calculating the orbit**

Here we show how the orbit can be calculated.
```python
from __future__ import print_function, division
import numpy as np
from PyAstronomy import pyasl
import matplotlib.pylab as plt

# Instantiate a Keplerian elliptical orbit with
# semi-major axis of 1.3 length units,
# a period of 2 time units, eccentricity of 0.5,
# longitude of ascending node of 70 degrees, an inclination
# of 10 deg, and a periapsis argument of 110 deg.
ke = pyasl.KeplerEllipse(1.3, 2., e=0.5, Omega=70., i=10.0, w=110.0)

# Get a time axis
t = np.linspace(0, 1.9, 200)

# Calculate the orbit position at the given points
# in a Cartesian coordinate system.
pos = ke.xyzPos(t)
print("Shape of output array: ", pos.shape)

# x, y, and z coordinates for 50th time point
print("x, y, z for 50th point: ", pos[50, ::])

# Calculate orbit radius as a function of the
radius = ke.radius(t)

# Calculate velocity on orbit
vel = ke.xyzVel(t)

# Find the nodes of the orbit (Observer at -z)
ascn, descn = ke.xyzNodes_LOSZ()

# Plot x and y coordinates of the orbit
plt.subplot(2, 1, 1)
plt.title("Periapsis (red diamond), Asc. node (green circle), desc. node (red circle) 
   \to")
plt.xlabel("East \to")
plt.ylabel("North \to")
plt.plot([0], [0], 'k+', markersize=9)
plt.plot(pos[:, 1], pos[:, 0], 'bp')
# Point of periapsis
plt.plot([pos[0, 1]], [pos[0, 0]], 'rd')
# Nodes of the orbit
plt.plot([ascn[1]], [ascn[0]], 'go', markersize=10)
plt.plot([descn[1]], [descn[0]], 'ro', markersize=10)
# Plot RV
plt.subplot(2, 1, 2)
plt.xlabel("Time")
plt.ylabel("Radial velocity [length/time]")
plt.plot(t, vel[:, 2], 'r.-')
plt.show()
```

**Module API**

The module defines the following classes:

- KeplerEllipse

---

2.1. Python AstroLib 17
• MarkleyKESolver

The *KeplerEllipse* class calculates the orbit and provides some convenience functions. For instance, the foci of the ellipse, and the peri- and apastron positions can be calculated.

The *MarkleyKESolver* class implements a solver for Kepler’s equation, which is needed to calculate the orbit as a function of time.

**Magnitude conversions**

**Translate absolute magnitude to power scale**

**PyAstronomy.pyasl.absMagToPower**

```python
PyAstronomy.pyasl.absMagToPower(am, absMagSun=4.75, absLumSun=3.846e+33)
```

Convert absolute magnitude to power scale

The default values for the absolute magnitude and luminosity of the Sun are adopted from Harmanec and Prsa 2011 (2011PASP..123..976H).

**Parameters**

- `am` [float] Absolute magnitude.
- `absLumSun` [float, optional] Absolute luminosity of the Sun. The default is given in units of erg/s.

**Returns**

- `Power` [float] Total emitted power. Same units as `absLumSun`; the default corresponds to erg/s.

**Example**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

absMagSun = 4.75
print("Absolute bolometric magnitude of the Sun: ", absMagSun)
print(" Absolute luminosity [erg/s]: ", pyasl.absMagToPower(absMagSun))
```

**Translate distance module into distance**

**PyAstronomy.pyasl.absModuleToDist**

```python
PyAstronomy.pyasl.absModuleToDist(magApp, magAbs)
```

Convert apparent and absolute magnitude into distance.

**Parameters**

- `magAbs` [float] Absolute magnitude of object.

**Returns**

- `Distance` [float] The distance resulting from the difference in apparent and absolute magnitude [pc].
Example

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Apparent magnitude
appMag = 11.37

# Absolute (bolometric) magnitude of Sun
absMagSun = 4.75

print("Distance of a sun-like star with apparent bolometric ", end="")
print("brightness of 11.37 mag: ", pyasl.absModuleToDist(appMag, absMagSun))
```

Convert magnitude into flux density

PyAstronomy.pyasl.magToFluxDensity_bessel98 (band, mag, mode='nu')

Convert magnitude into flux density according to Bessell et al. 1998

The conversion implemented here is based on the data given in Table A2 of Bessell et al. 1998, A&A 333, 231-250, which gives “Effective wavelengths (for an A0 star), absolute fluxes (corresponding to zero magnitude) and zeropoint magnitudes for the UBVRI- JHKL Cousins-Glass-Johnson system”. Note that zp(f_nu) and zp(f_lam) are exchanged in the original table.

Parameters

- **mag** [float, array] The magnitude value to be converted
- **mode** [string, {nu, mod}] Determines whether f_nu or f_lam will be calculated.

Returns

- **f_nu/lam** [float] The corresponding flux density in units if erg/cm**2/s/Hz in the case of mode ‘nu’ and erg/cm**2/s/A in the case of ‘lam’.
- **lam_eff** [float] Effective filter wavelength in Angstrom

Example

```python
from __future__ import print_function
from PyAstronomy import pyasl
import numpy as np

mag_R = 15.5

fd_nu, le = pyasl.magToFluxDensity_bessel98("R", mag_R, "nu")
fd_lam, _ = pyasl.magToFluxDensity_bessel98("R", mag_R, "lam")

print("R-band magnitude: ", mag_R)
print("R-band flux density [erg/cm**2/s/Hz]: ", fd_nu)
print("R-band flux density [erg/cm**2/s/A]: ", fd_lam)
print("Effective wavelength of filter [A]: ", le)
```

(continues on next page)
print("Convert f_nu into f_lam [erg/cm**2/s/A] by multiplication with (c/lam**2): ",
f_d_nu * (29979258e2/(le/1e8)**2) / 1e8)

S-Index and RHK

class PyAstronomy.pyasl.SMW_RHK(ccfs='rutten', afc='middelkoop', rphot='noyes')
Converting Mount-Wilson S-index into RHK index.

The Mount-Wilson S-index is a measure of the emission-line cores of the Ca II H and K lines at about 3933 A
and 3968 A in two narrow bands normalized by two adjacent continuum bands.

The activity index RHK is closely related to the S-index. In particular, it gives the emission in the narrow bands
normalized by the bolometric brightness of the star

\[ R_{HK} = \frac{4\pi R^2_s(F_H + F_K)}{4\pi R^2_s\sigma T_{eff}^4} = \frac{F_H + F_K}{\sigma T_{eff}^4} . \]

The stellar surface flux in “arbitrary units” in the narrow H and K bands \( f_H + f_K \) is related to the Mount-Wilson
S-index through the relation

\[ f_H + f_K = S C_{cf} T_{eff}^4 10^{-14} , \]

where \( C_{cf} \) is a conversion factor, which can be parameterized in terms of the B-V color and the luminosity class.
The conversion between arbitrary units and physical units, needed to derive the true surface flux and the RHK
index, has been derived by several authors starting with Middelkoop 1982. Their factor was also used by Noyes
et al. 1984—in particular in their appendix a, where is appears implicitly. Later, the value of the conversion
factor has been revised by several authors, e.g., Oranje 1983 and Rutten 1984, who estimated a value about
70% larger than previously proposed. Hall et al. 2007 derive a value 40% larger than that of Middelkoop 1982
and provide a thorough discussion on the differences between the individual approaches and results given in the
literature.

Finally, the RHK index thus derived still covers a photospheric contribution, which is always present and not
related to the chromosphere. To obtain the purely chromospheric, primed RHK index, an estimate of the photospheric
surface flux in the H and K pass-bands has to be subtracted. For active stars, the photospheric correction
is usually quite irrelevant. For inactive, quiet stars, it can, however, be important.

The issue of the Mount-Wilson S-index conversion has been revisited by Mittag et al. 2013, who provide an
alternative conversion procedure and revised photospheric corrections for various luminosity classes.

Note: In the default configuration, the conversion of the S-index into RHK is identical to the relation stated by
Noyes et al. 1984 in their appendix (a)

\[ R_{HK} = 1.340 \times 10^{-4} C_{cf} S \]

where the factor 1.34e-4 is a combination of the conversion from arbitrary to physical units, 1e-14, and the
Stefan-Boltzmann constant, in particular 1.34e-4 = 7.6e5*1e-14/5.67e-5. The \( C_{cf} \) factor is, however, calculated
according to Rutten 1984.

The relations and coefficients used here are taken from the following publications (and references therein):

• Rutten 1984, A&A 130, 353
• Hall et al. 2007, AJ 133, 862
• Mittag et al. 2013, A&A 549, 117

Parameters

ccfs [string, {rutten, noyes}, optional] Source of the conversion factor between S-index and RHK.

afc [string, {rutten, oranje, middelkoop, hall}, optional] Source of conversion factor between “arbitrary units” and physical units of surface flux.

rphot [string, {noyes}] The source for the photospheric correction for the RHK index.

Methods

 FHFK(S, Teff, log10ccf) Calculate the FH+FK flux in arbitrary units.

 SMWtoRHK(S, Teff, bv[, lc, verbose]) Convert Mount-Wilson S-index into R_HK.

 log10ccfNoyes(bv[, **kwargs]) Ccf conversion factor according to Noyes et al.

 log10ccfRutten(bv[, lc]) Ccf conversion factor from Rutten 1984 (Eqs.

 logRphotNoyes(bv[, lc]) Photospheric contribution to surface flux in the H and K pass-bands.

FHFK (S, Teff, log10ccf)
Calculate the FH+FK flux in arbitrary units.

Parameters


Teff [float] The effective temperature [K].

log10ccf [float] The logarithm of the Ccf conversion factor.

Returns


SMWtoRHK (S, Teff, bv, lc='ms', verbose=False)
Convert Mount-Wilson S-index into R_HK.

Parameters


Teff [float] Effective temperature [K].

bv [float] B-V color [mag]

lc [String, {ms, g}, optional] Luminosity class; Main-sequence (ms) or giants (g)

verbose [boolean, optional] If True, the details of the calculation are printed to stdout.

Returns

RHK prime [float] RHK parameter corrected for photospheric contribution. The primed number measures the purely chromospheric emission.

RHK [float] RHK parameter without correction for photospheric contribution.
The Ccf conversion factor used.

- **fhfk** [float] The FH+FK surface flux in arbitrary units.
- **fhfk (physical)** [float] The FH+FK surface flux in physical units [erg/cm²/s].
- **R_phot** [float] Photospheric flux contribution used in translating RHK into RHK prime.

**log10ccfNoyes**(bv, **kwargs)

Ccf conversion factor according to Noyes et al. 1984.

Parameters

- bv [float] The B-V color [mag].

Returns

- log10(Ccf) [float] The logarithm of the conversion factor.

**log10ccfRutten**(bv, lc='ms')

Ccf conversion factor from Rutten 1984 (Eqs. 10a and 10b).

Parameters

- bv [float] B - V color [mag].
- lc [string, {ms, g}, optional] Specifies whether the relation for main-sequence (ms) or giant (g) stars shall be evaluated.

Returns

- log10(Ccf) [float] The logarithm of the conversion factor.

**logRphotNoyes**(bv, lc='ms')

Photospheric contribution to surface flux in the H and K pass-bands.


Parameters

- bv [float] B-V color [mag]
- lc [string, {ms, g}, optional] Luminosity class.

Returns

- log10(Rphot) [float] Logarithm of the photospheric contribution.

### Convert Mount-Wilson S-index into RHK

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

ss = pyasl.SMW_RHK()

bv = 0.8
teff = 5100.0
s = 0.4

print("Convert S-index to RHK assuming a giant")
ss.SMWtoRHK(s, teff, bv, lc="g", verbose=True)

print()
```

(continues on next page)
print()
print("Convert S-index to RHK assuming a main-sequence star")
ss.SMWtoRHK(s, teff, bv, lc="ms", verbose=True)

Show the Ccf conversion factor

```python
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pylab as plt

ss = pyasl.SMW_RHK()

bv = np.arange(0.4, 0.9, 0.05)
ccfn = bv * 0.0
ccfr = bv * 0.0
ccfrg = bv * 0.0

for i in range(len(bv)):
    ccfn[i] = ss.log10ccfNoyes(bv[i])
    ccfr[i] = ss.log10ccfRutten(bv[i])
    ccfrg[i] = ss.log10ccfRutten(bv[i], lc="g")

plt.plot(bv, ccfn, 'b.-', label="Noyes")
plt.plot(bv, ccfr, 'r.-', label="Rutten (ms)")
plt.plot(bv, ccfrg, 'g.-', label="Rutten (g)")
plt.xlabel("B - V [mag]")
plt.ylabel("Ccf")
plt.legend()
plt.show()
```

Atomic number and elemental symbols

```python
class PyAstronomy.pyasl.AtomicNo

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAtomicNo(sym)</td>
<td>Convert elemental symbol into atomic number.</td>
</tr>
<tr>
<td>getElSymbol(atn)</td>
<td>Convert atomic number into elemental symbol.</td>
</tr>
<tr>
<td>getElementName(atn)</td>
<td>Convert atomic number into elemental name.</td>
</tr>
<tr>
<td>showAll()</td>
<td>Print atomic number, elemental symbol, and element name.</td>
</tr>
</tbody>
</table>

getAtomicNo(sym)

Convert elemental symbol into atomic number.

Parameters

- **sym [string]** Elemental symbol

Returns

- **Atomic number [int]** Atomic number

2.1. Python AstroLib
**getElSymbol** *(atn)*  
Convert atomic number into elemental symbol.

**Parameters**

- **atn** [int] Atomic number

**Returns**

- **Symbol** [string] Elemental symbol

**getElementName** *(atn)*  
Convert atomic number into elemental name.

**Parameters**

- **atn** [int] Atomic number

**Returns**

- **Name** [string] Name of element

**showAll** ()  
Print atomic number, elemental symbol, and element name.

**Example**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
an = pyasl.AtomicNo()

# Show table with all atomic numbers, elemental symbols, and the names of the elements
an.showAll()

print(an.getElSymbol(26))
print(an.getAtomicNo("He"))
print(an.getElementName(25))
```

**Elemental abundances**

**Abundance patterns**

**class** PyAstronomy.pyasl.AbundancePatterns  
This class provides widely adopted patterns of (number) abundances.

Data are from [https://heasarc.gsfc.nasa.gov/docs/xanadu/xspec/manual/XSabund.html](https://heasarc.gsfc.nasa.gov/docs/xanadu/xspec/manual/XSabund.html)

The following abundance patterns are available:

- **feld**, from Feldman U.(1992, Physica Scripta 46, 202 except for elements not listed which are given grsa abundances),
• aneb, from Anders E. & Ebihara (1982, Geochimica et Cosmochimica Acta 46, 2363),
• grsa from Grevesse, N. & Sauval, A.J. (1998, Space Science Reviews 85, 161)
• wilm from Wilms, Allen & McCray (2000, ApJ 542, 914 except for elements not listed which are given zero abundance)

Methods

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abundance(element[, pat])</code></td>
<td>Get the abundance of a specific element.</td>
</tr>
<tr>
<td><code>availablePatterns()</code></td>
<td>Returns the names (abbreviations) of the available abundance patterns.</td>
</tr>
<tr>
<td><code>pattern(name[, form, key])</code></td>
<td>Get the elemental abundance pattern</td>
</tr>
</tbody>
</table>

```python
abundance(element, pat='angr')
```
Get the abundance of a specific element.

**Parameters**

- **element** [int or string] If an integer is given, it is interpreted as the atomic number. If a string is given, it is the elemental symbol (e.g., H, He).
- **pat** [string, optional] The name of the abundance table.

**Returns**

- **abundance** [float] Number abundance of specified element relative to hydrogen.

```python
availablePatterns()
```
Returns the names (abbreviations) of the available abundance patterns.

**Returns**

- **names** [array of strings] The names of the available abundance patterns.

```python
pattern(name, form='array', key='symbol')
```
Get the elemental abundance pattern

**Parameters**

- **name** [string] The abbreviation of the abundance pattern.
- **form** [string, {array, dict}, optional] Return the abundances as a plain array or as a dictionary. The default is ‘array’.
- **key** [string, {symbol, number}, optional] If return type is a dictionary, this parameter determined whether the elemental symbol or the atomic number is used as the key. The default is “symbol”.

**Returns**

- **abundances** [numpy array or dict (depending on ‘form’)] Number abundances (not mass) relative to H.

Example of usage

2.1. Python AstroLib
from __future__ import print_function, division
from PyAstronomy import pyasl

ap = pyasl.AbundancePatterns()

print("Names of the available abundance patterns:")
print(ap.availablePatterns())

print()
print("Get the Asplund et al. pattern (aspl) as a dictionary using")
print("atomic number as a key:")
print(ap.pattern("aspl", form="dict", key="number"))

print()
print("Get (relative) number abundance of oxygen using elemental symbol:")
print(ap.abundance("O", pat="wilm"))
print("or atomic number")
print(ap.abundance(8, pat="wilm"))

X-ray luminosity / rotation period relations

class PyAstronomy.pyasl.Pizzolato2003
X-ray luminosity/rotation period relations by Pizzolato et al. 2003.

Implementation of the relations between X-ray luminosity and rotation period given by Pizzolato et al. 2003, A&A 397, 147P.

To find the X-ray luminosity, the following expression is evaluated

\[ x = \log_{10}(x_{sat}) + 2 \log_{10}(P_{r, sat}) - 2 \log_{10}(P_r) \]

where x denotes either \( L_x \) or \( L_x/L_{bol} \) and 'sat' indicates the value at saturation. The coefficients can be found in Tables 3 and 4 of Pizzolato et al. 2003.

Methods

- \texttt{log10lxbv(bv, pr)} Estimate \( \log_{10}(L_x) \)
- \texttt{log10lxlbolbv(bv, pr)} Estimate \( \log_{10}(L_x/L_{bol}) \)
- \texttt{log10lxlbolmass(m, pr)} Estimate \( \log_{10}(L_x/L_{bol}) \)
- \texttt{log10lxmass(m, pr)} Estimate \( \log_{10}(L_x) \)

\texttt{log10lxbv (bv, pr)}

\begin{verbatim}
Estimate \( \log_{10}(L_x) \)
\end{verbatim}

\textbf{Parameters}

- \texttt{bv} [float] B-V color [mag]
- \texttt{pr} [float] Rotation period [d]

\textbf{Returns}

- \texttt{log10(Lx)} [float] X-ray luminosity estimate
- \texttt{Error log10(Lx)} [float] Uncertainty
log10lxlbolbv \((bv, pr)\)

Estimate \(\log_{10}(L_x/L_{bol})\)

**Parameters**
- \(bv\) [float] B-V color [mag]
- \(pr\) [float] Rotation period [d]

**Returns**
- \(\log_{10}(L_x/L_{bol})\) [float] X-ray luminosity estimate
- \(\text{Error } \log_{10}(L_x/L_{bol})\) [float] Uncertainty

log10lxlbolmass \((m, pr)\)

Estimate \(\log_{10}(L_x/L_{bol})\)

**Parameters**
- \(m\) [float] Stellar mass [M_{Sun}]
- \(pr\) [float] Rotation period [d]

**Returns**
- \(\log_{10}(L_x/L_{bol})\) [float] X-ray luminosity estimate
- \(\text{Error } \log_{10}(L_x/L_{bol})\) [float] Uncertainty

log10lxmass \((m, pr)\)

Estimate \(\log_{10}(L_x)\)

**Parameters**
- \(m\) [float] Stellar mass [M_{Sun}]
- \(pr\) [float] Rotation period [d]

**Returns**
- \(\log_{10}(L_x)\) [float] X-ray luminosity estimate
- \(\text{Error } \log_{10}(L_x)\) [float] Uncertainty

**Example: Plot \(L_x\) and \(L_x/L_{bol}\) as a function of rotation period**

```python
from PyAstronomy import pyasl
import matplotlib.pyplot as plt
import numpy as np

p = pyasl.Pizzolato2003()

# Define array of rotation periods [days]
prot = np.arange(0.2, 30, 0.1)

lx = np.zeros(prot.size)
lxlbol = np.zeros(prot.size)

# B-V color of star
bv = 0.7

# Obtain ...
```

(continues on next page)
for i in range(prot.size):
    # ... log10 of X-ray luminosity
    lx[i] = p.log10lxbv(bv, prot[i])[0]
    # ... and log10(Lx/Lbol)
    lxlbol[i] = p.log10lxlbolbv(bv, prot[i])[0]

# Plot result
plt.subplot(2, 1, 1)
plt.plot(prot, lx, 'bp-')
plt.subplot(2, 1, 2)
plt.plot(prot, lxlbol, 'bp-')
plt.show()

Converting between effective temperature and stellar color

Conversion between effective temperature and stellar color according to Ramirez and Melendez (several bands, metallicity dependent) and Ballesteros 2012 (black-body approximation).

Conversion according to Ramirez and Melendez 2005

In their 2005 publication, Ramírez and Meléndez (ApJ 626, 465-485) present metallicity-dependent relations between stellar effective temperature and color. Based on these relations, the class Ramirez2005 allows to convert between effective temperature and color. All 17 color indices given by the authors can be used.

Example:

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Create class instance
r = pyasl.Ramirez2005()

# Which color bands are available
print("Available color bands: ", r.availableBands())

# Convert B-V to effective temperature and back
bv = 0.75
feh = 0.0
teff = r.colorToTeff("B-V", bv, feh)
bv1 = r.teffToColor("B-V", teff, feh)
print("B-V = ", bv, ", Teff = ", teff, ", bv1 = ", bv1, ", bv-bv1 = ", bv-bv1)
```

Conversion according to Ballesteros 2012

Ballesteros 2012 (EPL 97, 34008) present a conversion between effective temperature and B-V color index based on a black body spectrum and the filter functions.
Comparison to Ramirez and Mendelez 2005

Below, a comparison between the effective temperatures derived using the Ballesteros 2012 and Ramirez and Mendelez 2005 procedures is given. Solar metallicity main-sequence stars were assumed in the conversion. Clearly, the difference reaches about 200 K for hot stars in the 7000 K range and becomes smaller for cooler stars.

```python
import print_function, division
from PyAstronomy import pyasl
b = pyasl.BallesterosBV_T()
r = pyasl.Ramirez2005()
# Convert B-V to effective temperature and back
for bv in [0.35, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95, 1.05, 1.15, 1.25, 1.35, 1.45]:
    tr = r.colorToTeff("B-V", bv, 0.0)
    tb = b.bv2T(bv)
    print("B-V [mag] = {0:4.2f} : Teff (R05) = {1:4.0f} K, Teff (B12) = {2:4.0f} K, dTeff = {3:4.0f} K".format(bv, tr, tb, tr-tb))
```

Output:

```
-------
B-V [mag] = 0.35 : Teff (R05) = 6952 K, Teff (B12) = 7158 K, dTeff = -206 K
B-V [mag] = 0.45 : Teff (R05) = 6453 K, Teff (B12) = 6625 K, dTeff = -171 K
B-V [mag] = 0.55 : Teff (R05) = 6033 K, Teff (B12) = 6170 K, dTeff = -138 K
B-V [mag] = 0.65 : Teff (R05) = 5672 K, Teff (B12) = 5778 K, dTeff = -106 K
B-V [mag] = 0.75 : Teff (R05) = 5357 K, Teff (B12) = 5436 K, dTeff = -78 K
B-V [mag] = 0.85 : Teff (R05) = 5082 K, Teff (B12) = 5134 K, dTeff = -53 K
B-V [mag] = 0.95 : Teff (R05) = 4835 K, Teff (B12) = 4866 K, dTeff = -31 K
B-V [mag] = 1.05 : Teff (R05) = 4612 K, Teff (B12) = 4626 K, dTeff = -13 K
B-V [mag] = 1.15 : Teff (R05) = 4410 K, Teff (B12) = 4409 K, dTeff = 1 K
B-V [mag] = 1.25 : Teff (R05) = 4225 K, Teff (B12) = 4213 K, dTeff = 13 K
B-V [mag] = 1.35 : Teff (R05) = 4055 K, Teff (B12) = 4034 K, dTeff = 21 K
B-V [mag] = 1.45 : Teff (R05) = 3897 K, Teff (B12) = 3870 K, dTeff = 27 K
```

Example:

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
b = pyasl.BallesterosBV_T()
bv = 0.65
# Convert B-V into effective temperature
teff = b.bv2T(0.65)
print("B-V = {0:4.2f} K -> Teff = {1:4.0f} K".format(bv, teff))
# Convert effective temperature into B-V color
teff = 4568.0
bv = b.t2bv(teff)
print("Teff = {0:4.0f} K -> B-V = {1:4.2f} mag".format(teff, bv))
```

2.1. Python AstroLib
API documentation (Ramirez2005)

class PyAstronomy.pyasl.Ramirez2005

Relation between effective temperature and color given by Ramirez and Melendez 2005.

Ramirez and Melendez 2015, ApJ 626, 465-485 (please note that Ramirez has a non-ASCII accent on the i and Melendez an accent on the second e) give a relation between the stellar effective temperature and the color concerning various bands. This class allows to carry out the conversion in both directions.

Methods

availableBands() Get a list of available band identifiers.

colorToTeff(band, X, feH[, stype, ignoreRange]) Converts color into effective temperature according to Eq.

colorToTeff_nop(band, X, feH[, stype]) Converts color into effective temperature according to Eq.

teffToColor(band, teff, feH[, stype, dteff, ...]) Converts effective temperature into color according to Eq.

teffToColor_nop(band, teff, feH[, stype, ...]) Converts effective temperature into color according to Eq.

_checkBand(band) Check whether band identifier is valid.

_checkST(st) Check whether stellar type (main-sequence/giant) is valid.

_convertBandName(bn) Convert band name used in tables to internal representation.

Parameters

bn [string] Band name used in table.

Returns

Band ID [string] Identifier used in the class.

_extractTableData(lines, tableno) Extract lines pertaining to specified table.

Parameters

lines [list of strings] Content of the data file.

tableno [int] Number of the table to be extracted.

Returns

Table data [list of strings] Part of the file belonging to the specified table.

_readData(fn) Read the table data.

Parameters

fn [string] Filename.

_readTab23(lines, tableno) Read tables 2 and 3.
Parameters

lines [list of strings] Content of the data file.

tableno [int] Number of the table to be extracted.

Returns

bands [list of strings] IDs of all bands in the table.

result [array] Table data as array.

__readTab45__(lines, tableno)
**Read tables 4 and 5.**

Parameters

lines [list of strings] Content of the data file.

tableno [int] Number of the table to be extracted.

Returns

result [array] Table data as array.

__resolveMetallicityIndex__(feh)
**Determine where to find coefficients for given metallicity in Tables 4 and 5.**

Parameters

feh [float] Metallicity

availableBands()
**Get a list of available band identifiers.**

Returns

Band IDs [list of strings] All strings used to identify bands.

colorToTeff(band, X, feH, stype='ms', ignoreRange=False)
**Converts color into effective temperature according to Eq. 2.**

This method takes the polynomial correction into account. Note that no interpolation is implemented between the polynomials defined in Tables 4 and 5, but the appropriate polynomial (according to footnote (a) on under the tables) is used.

Parameters

band [string] Band identifier.

X [float] The color index (e.g., value of B-V).

feH [float] Metallicity

stype [string, {ms, g}] Type of star (main sequence or giant).

ignoreRange [boolean, optional] If True, the validity range of the relations will be ignored. Otherwise (default) an exception will be raised when a value outside the range is encountered.

Returns

Teff [float] The effective temperature in K.

colorToTeff_nop(band, X, feH, stype='ms')
**Converts color into effective temperature according to Eq. 1.**
The conversion using to Eq. 1 neglects a polynomial correction for metallicity. According to RM05, this causes a systematic error on the order of ‘30 or 40 K’.

**Parameters**

- **band** [string] Band identifier.
- **X** [float] The color index (e.g., value of B-V).
- **feH** [float] Metallicity
- **stype** [string, {ms, g}] Type of star (main sequence or giant).

**Returns**

- **Teff** [float] The effective temperature in K.

**teffToColor** *(band, teff, feH, stype='ms', dteff=0.01, maxiter=100)*

Converts effective temperature into color according to Eq. 2.

This method inverts Eq. 2 using an iterative scheme.

**Parameters**

- **band** [string] Band identifier.
- **teff** [float] Effective temperature in K.
- **feH** [float] Metallicity
- **stype** [string, {ms, g}, optional] Type of star (main sequence or giant).
- **dteff** [float, optional] Temperature difference to be reached by the iteration [K]. Default is 0.01.
- **maxiter** [int, optional] The maximum number of iterations to be carried out. Default is 100.

**Returns**

- **X** [float] Color in the specified band.

**teffToColor_nop** *(band, teff, feH, stype='ms', noRaise=False)*

Converts effective temperature into color according to Eq. 1.

This method inverts Eq. 1. Note that the equation is parabolic in the color (i.e., X). Therefore, there are two solutions of which the one falling within the validity ranges specified in Tables 4 and 5 of RM05 is selected. If none or both of the solutions are valid, an exception is raised.

**Parameters**

- **band** [string] Band identifier.
- **teff** [float] Effective temperature in K.
- **feH** [float] Metallicity
- **stype** [string, {ms, g}] Type of star (main sequence or giant).
- **noRaise** [boolean, optional] If True, no exceptions will be raised, but warnings will be given. Both candidate solutions will be returned in this case.

**Returns**

- **X** [float] Color in the specified band.
API documentation (BallesterosBV_T)

class PyAstronomy.pyasl.BallesterosBV_T
Black-body based conversion between effective temperature and B-V color.
Ballesteros 2012 (EPL 97, 34008) present a conversion between effective temperature and B-V color index based on a black body spectrum and the filter functions.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bv2T(bv)</code></td>
<td>Convert B-V color into temperature.</td>
</tr>
<tr>
<td><code>t2bv(T)</code></td>
<td>Convert temperature into B-V color.</td>
</tr>
</tbody>
</table>

`bv2T(bv)`
Convert B-V color into temperature.

Parameters

- `bv` [float] B-V color index [mag]

Returns

- `T` [float] Temperature [K]

`t2bv(T)`
Convert temperature into B-V color.

Parameters

- `T` [float] Temperature in K.

Returns

- `bv` [float] B-V color index [mag].

Stellar ages

Below, algorithms for estimating stellar ages based on rotation and chromospheric activity are given.

Gyrochronological age

PyAstronomy.pyasl.gyroAgeBarnes(p, bv)
Calculate gyrochronological age according to Barnes 2007.

The gyrochronological age is calculated according to Eq. 3 in Barnes 2007 (ApJ 669, 1167). The derivation of the error follows Eq. 16.

Parameters

- `p` [float] Stellar rotation period [d].
- `bv` [float] B-V color [mag]. Supported range is 0.4-1.6 mag.

Returns

- Stellar age [float] The gyrochronological age [Ga].
- Age error [float] The error on the age [Ga].
Example

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Parameters of the Sun (Barnes 2007, p 1174)
bv = 0.642
p = 26.09

# Obtain solar age ...
age = pyasl.gyroAgeBarnes(p, bv)
# ... and print it
print("Solar age: {0:4.2f} +/- {1:4.2f} Ga".format(*age))
```

Chromospheric age

PyAstronomy.pyasl.chromoAgeRHK(log10RHKprime)

Calculate the chromospheric age according to Donahue 1998.

Donahue 1998 (ASPC 154, 1235) give a relation between chromospheric activity as measured by the R’HK index and the age of late-type stars (Eq. 1).

As the level of stellar activity undergoes continuous change, Donahue cautions that individual measurements of the activity level yield notoriously poor age estimates. As an example, the spread in chromospheric solar age resulting from the 11 yr activity cycle is given, which amounts to about 2.5 Ga. This gives an idea of the accuracy of the estimates.

Parameters

- **log10RHKprime** [float] Chromospheric activity index log10(R’HK).

Returns

- **Age** [float] Stellar age [Ga].

Example

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Approximate chromospheric age of the Sun
print("Solar age: {0:4.2f} Ga".format(pyasl.chromoAgeRHK(-4.95)))
```

Spectral type vs. Teff and luminosity


Example: Basic Usage
from __future__ import print_function
from PyAstronomy import pyasl

# Instantiate class object
sdj = pyasl.SpecTypeDeJager()

llum, lteff = sdj.lumAndTeff("K0", "V")

print("Luminosity = {0:4.2f} Lsun".format(10.0**llum))
print("Effective temperature = {0:6.1f} K".format(10.0**lteff))

Example: Teff and luminosity as a function of spectral type

from __future__ import print_function
from PyAstronomy import pyasl
import matplotlib.pyplot as plt

# Instantiate class object
sdj = pyasl.SpecTypeDeJager()

# Set luminosity class
lk = "V"

# Save spectral types, log(teff), and log(luminosity)
spts = []
lteffs = []
llums = []

# Save information to annotate abscissa
xt = []
xtl = []

for t in "OBAFGKM":
    for n in range(10):
        if (t == "O") and (n == 0):
            # Skip the invalid "O0" type
            continue

        # Save the spectral type
        spts.append(t + str(n))

        # Get log10 of luminosity and effective temperature
        ll, lt = sdj.lumAndTeff(spts[-1], lk)
        # and save to lists
        llums.append(ll)
lteffs.append(lt)

        # Save location (i.e., number in the list) and
        # spectral for annotating the abscissa
        if (n == 0) or (n == 5):
            xt.append(len(spts)-1)
            xtl.append(spts[-1])

ax1 = plt.subplot(2, 1, 1)
(continues on next page)
# Plot log10(effective temperature)
plt.plot(lteffs)
plt.ylabel(r'$\log_{10}(T_{\text{eff}})$')
plt.setp(ax1, xticks=xt, xticklabels=xtl)
ax2 = plt.subplot(2, 1, 2)
# Plot log10(luminosity)
plt.plot(llums)
plt.ylabel(r'$\log_{10}(L/L_{\odot})$')
plt.setp(ax2, xticks=xt, xticklabels=xtl)
plt.xlabel('Spectral type')
plt.show()

API

class PyAstronomy.pyasl.SpecTypeDeJager
Spectral type calibration from de Jager and Nieuwenhuijzen.

This class implements the spectral type calibration presented by de Jager and Nieuwenhuijzen 1987, A&A 177, 217-227 (DJ87). Specifically, the authors calibrate the relation between spectral type and stellar luminosity and effective temperature for a wide range of spectral types and luminosity classes.

DJ87 give their results in the form of a polynomial relation with 20 coefficients (Eqs. 2a and 2b) as well as in the form of their Tables 5 and 6.

Note: Tables 5 and 6 in DJ87 are calculated on the basis of a 40 parameter solution, which is not given in the paper. The numbers based on the 20 parameter polynomial solution deviate from the tabulated numbers by typically a few percent for the effective temperature and 10 percent for the luminosities (not their logarithm).

Methods

lumAndTeff(spt, lk) Get effective temperature and luminosity.

_cheby(i, x) Evaluate the Chebychev polynomial.

_decomposeSPT(spt) Decomposes spectral type into ‘basic type’ and subtype.

Parameters

spt [string] Spectral type (e.g, F8.5)

Returns

Basic type [string] Type with integer subtype (e.g., F8)

Subtype [float] Subtype as a float (e.g., 8.5)

_indexSpecType(spt) Get index of spectral type in list and check validity.

Returns

index [int] Position in self._types list
_resolve_b(lk)
Resolve numerical value of variable b.

Parameters
lk [string] Luminosity class (e.g., IV)

Returns
b [float] The value of b

_resolve_s(bt, st)
Resolve numerical value of variable s.

Parameters
bt [string] Spectral type (e.g., F3), no LK included
st [float] Subtype (e.g., 3.5 from F3.5)

Returns
s [float] Value of s

lumAndTeff(spt, lk)
Get effective temperature and luminosity.

Parameters
spt [string] The spectral type (may include float subtype). For instance, F3, F3.5, or K2
lk [string, {V, IV, III, II, Ib, Iab, Ia, Ia+}] The luminosity class.

Returns
log10(L) [float] The base-10 logarithm of the luminosity in units of the solar luminosity.
log10(teff) [float] The base-10 logarithm of the effective temperature.

Roche potential

Let there be two masses $m_1$ and $m_2$ with $m_1 \geq m_2$ and $m_1$ centered at $(x, y, z) = (0, 0, 0)$ and $(x, y, z) = (a, 0, 0)$, where $a$ is the separation between the masses. Both objects are located in the x,y plane with x-axis counting distance along the connecting line and z measuring distance perpendicular to the orbital plane.

For synchronous rotation and a circular orbit, the Roche potential reads (e.g., Hilditch 2001 “An introduction to close binary stars”)

$$
\Phi = -\frac{G m_1}{r_1} - \frac{G m_2}{r_2} - \frac{\omega^2}{2} \left[ \left( x - a \frac{m_2}{m_1 + m_2} \right)^2 + y^2 \right]
$$
with

\[ r_1 = \sqrt{x^2 + y^2 + z^2} \]
\[ r_2 = \sqrt{(x - a)^2 + y^2 + z^2} \]
\[ q = \frac{m_2}{m_1} \]
\[ M = m_1 + m_2 \]
\[ m_1 = \frac{M}{1 + q} \]
\[ m_2 = \frac{Mq}{1 + q} \]

\[ \omega^2 = \frac{GM}{a^3} \quad \text{(Kepler’s third law)} \]

\[ x', y', z', r'_1, r'_2 = \frac{x}{a}, \frac{y}{a}, \frac{z}{a}, \frac{r_1}{a}, \frac{r_2}{a} \]

we write

\[ \Phi = -\frac{GM}{(1 + q)r_1} - \frac{GMq}{(1 + q)r_2} - \frac{GM}{2a^3} \left[ \left( x - \frac{aq}{1 + q} \right)^2 + y^2 \right] \]
\[ \Phi = -\frac{GM}{(1 + q)r'_1 a} - \frac{GMq}{(1 + q)r'_2 a} - \frac{GM}{2a^3} \left[ \left( x' - \frac{aq}{1 + q} \right)^2 + y'^2 a^2 \right] \]
\[ \Phi = -\frac{GM}{2a} \left( \frac{2}{(1 + q)r'_1} + \frac{2q}{(1 + q)r'_2} + \left( x' - \frac{q}{1 + q} \right)^2 + y'^2 \right) \]
\[ \Phi = -\frac{GM}{2a} \Phi_n(x', y', z', q) \]

where \( \Phi_n \) is the dimensionless Roche potential.

**Example of usage**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pylab as plt

x, y = np.linspace(-1.5, 2, 300), np.linspace(-1.6, 1.6, 300)
xx, yy = np.meshgrid(x, y)
# Coordinates in orbital plain
z = 0
# Mass ratio
q = 0.2
# Get dimensional values of Roche potential
p = pyasl.rochepot_dl(xx, yy, z, q)
# Positions (and potentials) of Lagrange points
l1, l1pot = pyasl.get_lagrange_1(q)
l2, l2pot = pyasl.get_lagrange_2(q)
l3, l3pot = pyasl.get_lagrange_3(q)
l4, l5 = pyasl.get_lagrange_4(), pyasl.get_lagrange_5()
```

(continues on next page)
14pot = pyasl.rochepot_dl(14[0], 14[1], 14[2], q)
15pot = pyasl.rochepot_dl(15[0], 15[1], 15[2], q)

print("Effective (dimensionless) radii of first and second mass")
print("According to the approximation of Eggleton 1983:")
r1eff = pyasl.roche_lobe_radius_eggleton(q, 1)
r2eff = pyasl.roche_lobe_radius_eggleton(q, 2)
print(" Reff1: $5.3f" % r1eff)
print(" Reff2: $5.3f" % r2eff)
print()
print("Roche volume and effective radius from Monte Carlo integration:")
mcvol1 = pyasl.roche_vol_MC(q, 1)
mcvol2 = pyasl.roche_vol_MC(q, 2)
print(" MC Roche lobe volume 1: $6.4f +/- $6.4f" % (mcvol1[0:2]))
print(" MC Roche lobe volume 2: $6.4f +/- $6.4f" % (mcvol2[0:2]))
print(" MC effective radius 1: $6.4f +/- $6.4f" % (mcvol1[2:]))
print(" MC effective radius 2: $6.4f +/- $6.4f" % (mcvol2[2:])))

plt.contour(p, [15pot*1.02, 13pot, 12pot, 11pot], colors=['g', 'c', 'b', 'r'],
extent=[-1.5, 2, -1.6, 1.6])
plt.text(l1, 0, 'L1', horizontalalignment='center')
plt.text(l2, 0, 'L2', horizontalalignment='center')
plt.text(l3, 0, 'L3', horizontalalignment='center')
plt.text(l4[0], l4[1], 'L4', horizontalalignment='center')
plt.text(l5[0], l5[1], 'L5', horizontalalignment='center')
plt.show()

Functionality and API

- Roche lobe potential
- Partial derivatives
- Roche lobe volume and radius
- Lagrange points

Roche lobe potential

PyAstronomy.pyasl.rochepot_dl(x, y, z, q)

Dimensionless Roche potential ($\Phi_n$, synchronous rotation)

More massive component ($m_1$) is centered at (x,y,z) = (0,0,0). Less massive component ($m_2$) is at (1,0,0). The unit of length is the distance between the objects. Both objects are in the x,y plane (x-axis along the connecting line and z perpendicular to the orbital plane).

Parameters

- x, y, z [float or array] Location(s) at which to calculate the potential. Unit of length is the distance between the masses m1 and m2.
- q [float] Mass ratio (0 <= m2/m1 <= 1)

Returns

- Potential [float or array] The potential at the specified location(s)
**PyAstronomy.pyasl.rochepot** *(x, y, z, m1, m2, a)*

Roche potential ($\Phi$, synchronous rotation)

More massive component ($m_1$) is at $(x,y,z) = (0,0,0)$. Less massive component ($m_2$) is at $(a,0,0)$.

**Parameters**
- $x$, $y$, $z$ [float or array] Position [m]
- $a$ [float] Separation of masses [m]
- $m_1$, $m_2$ [float] Masses of objects [kg] ($m_1 \geq m_2$)

**Returns**
- **Roche potential** [float or array] At specified locations [$m^2/\text{s}^2$]

**Partial derivatives**

**PyAstronomy.pyasl.ddx_rochepot_dl** *(x, q, y=0, z=0)*

Derive of dimensionless Roche potential along x-axis

**Parameters**
- $x$, $y$, $z$ [float or array] Position (default for $y$ and $z$ is zero)
- $q$ [float] Mass ratio $m_2/m_1$

**Returns**
- **Derivative**: float or array $d/dx$ of dimensionless Roche potential

**PyAstronomy.pyasl.ddy_rochepot_dl** *(y, q, x=1, z=0)*

Derive of dimensionless Roche potential along y-axis

**Parameters**
- $x$, $y$, $z$ [float or array] Position (default for $x$ is one and zero for $z$)
- $q$ [float] Mass ratio $m_2/m_1$

**Returns**
- **Derivative**: float or array $d/dy$ of dimensionless Roche potential

**PyAstronomy.pyasl.ddz_rochepot_dl** *(z, q, x=1, y=0)*

Derive of dimensionless Roche potential along z-axis

**Parameters**
- $x$, $y$, $z$ [float or array] Position (default for $x$ is one and zero for $z$)
- $q$ [float] Mass ratio $m_2/m_1$

**Returns**
- **Derivative**: float or array $d/dz$ of dimensionless Roche potential

**Roche lobe volume and radius**

**PyAstronomy.pyasl.roche_lobe_radius_eggleton** *(q, m)*

The effective Roche lobe radius is the radius of a sphere with the same volume as the actual equipotential surface defining the Roche lobe.

\[ r_L \approx 0.49 \frac{q^{2/3}}{0.6 q^{2/3} + \ln(1 + q^{1/3})} \]

Parameters
- \( q \) [float or array] Mass ratio (m2/m1)
- \( m \) [int, \{1,2\}] Calculate radius for mass 1 or 2

PyAstronomy.pyasl.roche_vol_MC \((q, m=2, n=100000, pl=None, eps=0.0001, fullout=True)\)
Calculate (dimensionless) volume of equipotential surface such as the Roche lobe

Uses Monte Carlo (MC) integration

Parameters
- \( q \) [float] Mass ratio (m2/m1)
- \( m \) [int, \{1,2\}] Whether to calculate volume for m1 or m2
- \( n \) [int, optional] Number of samples to be used in the Monte Carlo integration. Default is 100000. Increase to improve accuracy.
- \( pl \) [float, optional] The (dimensionless) potential level bounding the volume. If None (default), the Roche lobe potential (L1) is used.
- \( eps \) [float, optional] Margin used to avoid singularities of Roche potential. Default is 1e-4.
- \( fullout \) [boolean, optional] If True (default), provides more output than volume.

Returns
- \( V, Verr \) [floats] Dimensionless volume of Roche lobe and estimate of uncertainty
- \( Reff, Refferr \) [floats, optional] Radius of a sphere with the same radius (dimensionless) and estimate of uncertainty. Provided if fullout is True (default).

PyAstronomy.pyasl.roche_yz_extent \((q, m=2, pl=None, eps=0.0001)\)
Extent of equipotential surface in y and z direction

Returns
- \( dy, dz \) [float] Distance to equipotential level along y and z direction.

Lagrange points

PyAstronomy.pyasl.get_lagrange_1 \((q, getdlrp=True, eps=0.0001)\)
Get location of first Lagrange point

Parameters
- \( q \) [float] Mass ratio
- \( getdlrp \) [boolean, optional] If True (default), also the dimensionless Roche potential at the first Lagrange point is returned.
- \( eps \) [float, optional] The potential diverges at x=0 and x=1. The search for a root of the derivative is performed in the interval [eps, 1-eps].

Returns
- \( xL1 \) [float] The location of the first Lagrange point (xL1, 0, 0).
Potential [float, optional] The dimensionless potential at the first Lagrange point (default is True).

PyAstronomy.pyasl.get_lagrange_2 (q, getdlrp=True, eps=0.0001)
Get location of second Lagrange point

Parameters
q [float] Mass ratio
getdlrp [boolean, optional] If True (default), also the dimensionless Roche potential at the first Lagrange point is returned.
eps [float, optional] The potential diverges at x=0 and x=1. The search for a root of the derivative is performed in the interval [1+eps, 1+10*rL], where rL is the estimated (effective) Roche lobe radius according to Eggleton 1983.

Returns
xL2 [float] The location of the second Lagrange point (xL2, 0, 0).
Potential [float, optional] The dimensionless potential at the second Lagrange point (default is True).

PyAstronomy.pyasl.get_lagrange_3 (q, getdlrp=True, eps=0.0001)
Get location of third Lagrange point

Parameters
q [float] Mass ratio
getdlrp [boolean, optional] If True (default), also the dimensionless Roche potential at the third Lagrange point is returned.
eps [float, optional] The potential diverges at x=0. The search for a root of the derivative is performed in the interval [-10, -eps].

Returns
xL3 [float] The location of the first Lagrange point (xL3, 0, 0).
Potential [float, optional] The dimensionless potential at the third Lagrange point (default is True).

PyAstronomy.pyasl.get_lagrange_4 ()
Get location of forth Lagrange point
Orbital angular momentum is supposed to point into +z direction. L4 is in direction of motion of m2.

Returns
x,y,z [float] The location of the fourth Lagrange point (xL3, 0, 0).

PyAstronomy.pyasl.get_lagrange_5 (getdlrp=True)
Get location of fifth Lagrange point
Orbital angular momentum is supposed to point into +z direction. L5 is behind motion of m2.

Returns
x,y,z [float] The location of the fifth Lagrange point (xL3, 0, 0).
General data analysis

Folding time series

PyAstronomy.pyasl.foldAt(time, period, T0=0.0, getEpoch=False)

Fold time series with a particular period.

Calculate the phase, \( P \), from time, period, and reference point, \( T_0 \), according to

\[
P = \frac{(time - T_0)}{period} - \left\lfloor \frac{(time - T_0)}{period} \right\rfloor.
\]

Here, square brackets indicate Gaussian brackets (i.e., the floor function), and the phase is a number between 0 and 1 by definition (and not between 0 and 2pi).

Optionally, also the epoch, \( E \), can be returned, which is an integer corresponding to the second term in the above equation. For any point of the series, therefore, the following relation applies

\[
time = T_0 + (E + P) \times period.
\]

Of course the series to be folded does not necessarily have to be a time series although this particular example guides the naming convention here.

Parameters

data  
array The time stamps.

period  
float The period to fold with (same units as time stamps).

T0  
float Time reference point. The point \( T_0 \) as well as all points \( T_0+n\times period \) with integer \( n \) are mapped to phase zero. Default is 0.0.

getEpoch  
boolean, optional If True, an array holding the epoch for every point in time will be returned; the default is False. Note that the first epoch, corresponding to times between \( T_0 \) and \( T_0+\text{per} \), is 0.

Returns

Phases  
array The (unsorted) phase array pertaining to the input time axis.

Epoch  
array, optional An array holding the epoch for every given point in time. The counting starts at zero. Only returned if \( \text{getEpoch} \) is True.

Folding example

```python
from PyAstronomy.pyasl import foldAt
import matplotlib.pyplot as plt
import numpy as np

# Generate some data ...
plt
plt.plot(time, flux, label='Original Data')
plt.legend()  # (continues on next page)
```

2.1. Python AstroLib

43
# Sort with respect to phase
# First, get the order of indices ...
sortIndi = np.argsort(phases)
# ... and, second, rearrange the arrays.
phases = phases[sortIndi]
flux = flux[sortIndi]

# Plot the result
plt.plot(phases, flux, 'bp')
plt.show()

## Binning algorithms
Create binned data sets.

### Constant bin width

PyAstronomy.pyasl.binningx0dt(x, y, yerr=None, x0=None, dt=None, nbins=None, reduceBy=None, removeEmpty=True, removeNoError=False, useBinCenter=True, useMeanX=False, nanHandling=None, yvalFunc=<function mean>)

A simple binning algorithm.

This algorithm uses a fixed bin-width to produce a binned data set. Either the bin-width, \( dt \), or the number of bins, \( nbins \), must be specified. The number of output bins may also depend on other flags such as, for example, \( removeNoError \).

If no errors are specified via \( yerr \), the errors for the binned data are estimated as the standard deviation of the input data points divided by the square root of their number. If \( yerr \) has been specified, error propagation is used to determine the error.

The behavior of the x-axis can be controlled via the \( useBinCenter \) flag.

Values which cannot be determined will be indicated by NaN. Various flags can be used to remove such bins from the binned data set.

**Parameters**

- **x, y** [array] The x and y data values.
- **yerr** [array, optional] Errors on the data values.
- **x0** [float, optional] Starting time of first bin. Default is lowest given x value.
- **dt** [float, optional] Width of a bin (either \( dt \), \( nbins \) or \( reduceBy \) must be given).
- **nbins** [int, optional] Number of bins to use (either \( dt \), \( nbins \) or \( reduceBy \) must be given). Note that this specifies the number of bins into which the range from \( x0 \) to the last data point is subdivided.
- **reduceBy** [int, optional] Reduce the number of elements in the array by the given factor (either \( dt \), \( nbins \) or \( reduceBy \) must be given). Note that in this case, \( x0 \) is set to the first (minimum x-value) and the number of bins, \( n \), is calculated according to the prescription: \( n = \text{int}(\text{round}(\text{len}(x)/\text{reduceBy})) \)
- **removeEmpty** [boolean, optional] If True (default), bins with no data points will be removed from the result.
**removeNoError** [boolean, optional] If True, bins for which no error can be determined will be removed from the result. Default is False.

**useBinCenter** [boolean, optional] If True (default), the time axis will refer to the center of the bins. Otherwise the numbers refer to the start of the bins.

**useMeanX** [boolean, optional] If True, the binned x-values refer to the mean x-value of all points in that bin. Therefore, the new time axis does not have to be equidistant.

**yvalFunc** [callable, optional] Function used to determine the value in a bin based on input data. Default is the mean value (np.mean). An alternative choice could, e.g., be np.median.

**nanHandling** [None, “ignore”, float, (optional)]

  Controls how NaNs in the data are handled.
  
  - None: By default (None), nothing is done and NaNs are treated as if they were valid input data, so that they are carried over into the binned data. This means that output bins containing NaN(s) will also end up as NaN(s). If ‘ignore’
  
  - ‘ignore’: In this case, NaNs contained in the input data are removed from the data prior binning. Note however, that x0, unless specified explicitly, will still refer to the first data point, whether or not this holds a NaN value.
  
  - float: If a float is given, input data values containing NaNs are replaced by the given float before binning. Note that no error on the data (yerr) can be considered in this case, to avoid erroneous treatment of un- or misspecified error values.

**Returns**

  **Binned data set** [array] An array with four columns: 1) The new x-axis, 2) The binned data (the mean value of the data points located in the individual bins), 3) Error of binned data, 4) The number of input data points used to create the bin. For instance, the new x-values can be accessed using result[:,0].

  **dt** [float] The width of the bins.

**Examples**

**Basic binning**

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy.pyasl import binningx0dt

# Generate some data
x = np.arange(999)
y = np.sin(x/100.)
y += np.random.normal(0, 0.1, len(x))

# Bin using fixed number of bins and start at x0 = -10.
# Use beginning of bin as starting value.
r1, dt1 = binningx0dt(x, y, nbins=50, x0=-10, useBinCenter=False)
# Use fixed bin width. Specify another (wrong) error estimate and
# use bin center.
r2, dt2 = binningx0dt(x, y, yerr=np.ones(len(x))*0.2, dt=dt1,
                      x0=-10, useBinCenter=True, removeNoError=True)
```

(continues on next page)
print("dt1, dt2: ", dt1, dt2)
print("Input data points in last bin: ", r2[-1, 3])

# Use the reducedBy flag to indicate the binning. In this case, x0
# will be set to the lowest x value in the data, and the number of
# bins will be calculated as: int(round(len(x)/float(reduceBy))).
# Here, we will, thus, obtain 100 bins.
r3, dt3 = binningx0dt(x, y,
    useBinCenter=True, removeNoError=True, reduceBy=10)

print("dt3: ", dt3)
print("Number of bins in third version: ", len(r3[:, 0]))

# Plot the output
plt.plot(x, y)
plt.errorbar(r1[:, 0], r1[:, 1], yerr=r1[:, 2], fmt='kp--')
plt.errorbar(r2[:, 0], r2[:, 1], yerr=r2[:, 2], fmt='rp--')
plt.errorbar(r3[:, 0], r3[:, 1], yerr=r3[:, 2], fmt='gp--')
plt.show()
Handling NaN values in data

```python
from PyAstronomy.pyasl import binningx0dt
import matplotlib.pyplot as plt
import numpy as np

# Set up figures
fig = plt.figure()
ax0 = fig.add_subplot(411)
ax1 = fig.add_subplot(412, sharex=ax0, sharey=ax0)
ax2 = fig.add_subplot(413, sharex=ax0, sharey=ax0)
ax3 = fig.add_subplot(414, sharey=ax0)

# Set up data
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [0.3, 0.5, 0.7, 0.2, 0.5, 0.9, 0.2, 0.7, 0.8, 0.6]
yerr = [0.1]*len(x)
r, dt = binningx0dt(x, y, yerr=yerr, x0=0.5, dt=2)
ax0.plot(x, y, marker='o')
ax0.plot(r[:, 0], r[:, 1], linestyle='--', drawstyle='steps-mid', marker='s')
ax0.set_title("Normal (w/o NaNs)")
ax0.set_xticklabels([])

y = [0.3, 0.5, np.nan, 0.2, 0.5, 0.9, np.nan, np.nan, 0.8, 0.6]
x, y = np.array(x), np.array(y)
r, dt = binningx0dt(x, y, yerr=yerr, x0=0.5, dt=2)
ax1.plot(x, y, marker='o')
ax1.plot(r[:, 0], r[:, 1], linestyle='--', drawstyle='steps-mid', marker='s')
ax1.set_title("With NaNs and nanHandling='None' (default)")
# ax1.set_xticklabels([])

r, dt = binningx0dt(x, y, yerr=yerr, x0=0.5, dt=2, nanHandling="ignore")
ax2.plot(x, y, marker='o')
ax2.plot(r[:, 0], r[:, 1], linestyle='--', drawstyle='steps-mid', marker='s')
ax2.set_title("With NaNs and nanHandling='ignore'")

r, dt = binningx0dt(x, y, x0=0.5, dt=2, nanHandling=0.5)
ax3.plot(x, y, marker='o')
ax3.plot(r[:, 0], r[:, 1], linestyle='--', drawstyle='steps-mid', marker='s')
ax3.set_title("With NaNs and nanHandling='0.5'")

ax0.set_xlim(0, 11.5)
ax3.set_xlim(0, 11.5)
plt.show()
```
Smoothing data

PyAstronomy.pyasl.smooth(x, windowLen, window='flat')

Smooth data using a window function.

This method is based on the convolution of a window function with the signal. The window function is normalized so that the sum of its entries amounts to one. The signal is prepared by adding reflected copies of the signal (with the window size) to both ends of the input array, so that the output array can have the same length as the input. Consequently the smoothing at the edges is actually based on extrapolation.

Note: This algorithm was adopted from the scipy cookbook (http://www.scipy.org/Cookbook/SignalSmooth). The copyright of the original algorithm belongs to the authors of that cookbook algorithm.

Parameters

x [array] The input signal

windowLen [int] The dimension of the smoothing window. It must be an odd integer.

window [string, {'flat', 'hanning', 'hamming', 'bartlett', 'blackman'}] The window function to be used. A flat window will produce a moving average smoothing.

Returns

Smoothed signal [array] The smoothed signal. Same length as input array.

Example of usage

```python
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pyplot as plt

# Create same "data" using a sine and random noise
x = np.linspace(0, 10, 100)
y = 0.5*np.sin(x/3.0*2.0*np.pi + 1.7)
y += np.random.normal(0.0, 0.2, len(y))

# Use two smoothing windows with the same window size
sm1 = pyasl.smooth(y, 11, 'flat')
sm2 = pyasl.smooth(y, 11, 'hamming')

# Plot the outcome
plt.title("Data and smoothed curves: flat (blue) and hamming window (red)")
plt.plot(x, y, 'bp')
plt.plot(x, sm1, 'b--')
plt.plot(x, sm2, 'r--')
plt.show()
```

Signal to Noise Ratio (SNR) estimation (Beta Sigma Procedure)

Estimating the amplitude of noise can often only be done after the data have been taken, because not all noise sources may be known with sufficient accuracy (if at all) prior to the measurement. Similarly, the details of the signal to be
measured are usually not known as well, which is the point of taking the measurements in the first place.

In the analysis of data, a reasonable estimate of the amplitude of noise is often required to carry out, e.g., a test of goodness-of-fit. Thus arises the problem of estimating the amplitude (e.g., the variance or standard deviation) of the noise contribution lacking knowledge about both the noise and the actual signal.

A widely-used technique to estimate the amplitude of noise is fitting, e.g., a polynomial to either all or a subsample of the measurements. The polynomial (or some other function) supposedly represents the (unknown) signal and the resulting residuals can be studied to estimate the amplitude of noise; an implementation of such a technique can be found here: estimateSNRFCT. The $\beta\sigma$ procedure provides an approach to noise estimation without the explicit need to evaluate an approximative function.

**Noise estimation via the $\beta\sigma$ procedure**

The $\beta\sigma$ procedure is based on the analysis of the distribution of numerical derivatives and is described in the corresponding paper entitled “A posteriori noise estimation in variable data sets”. The idea is the following: If the signal can appropriately be approximated by an N-th degree polynomial, its (N+1)-th derivative vanishes, leaving only noise terms to contribute to the value of the numerical derivative. The same idea underlies the DER_SNR algorithm; both are, in fact, identical for specific settings of the $\beta\sigma$ procedure.

The basic assumptions made in the current implementation of the $\beta\sigma$ procedure are (i) Gaussian noise with standard deviation $\sigma_0$ in all analyzed bins and (ii) independent noise contributions in individual measurements. The assumption of Gaussian noise can be considered a rather weak one. In practice, a Gaussian distribution often provides a reasonable approximation of the noise properties, although the noise itself may almost never be expected to be exactly Gaussian. If noise contributions, e.g., in adjacent measurements, are not independent, subsamples containing more distant measurements may be selected (jump parameter) to estimate the noise properties.

**Quick start**

Estimating the noise contribution in a data set $x$, $y$:

```python
from __future__ import print_function
from PyAstronomy import pyasl
import numpy as np

# Set up artificial data set
x = np.linspace(0., 1., 200)
y = 10.2 + x + 4 * x**2
# Add some independent, Gaussian noise
gstd = 0.1
y += np.random.normal(0., gstd, len(x))

# Estimate noise using robust estimate
beq = pyasl.BSEqSamp()
# Define order of approximation (use larger values such as 2,3, or 4 for
# faster varying or less well sampled data sets; also 0 is a valid order)
N = 1
# Define 'jump parameter' (use larger values such as 2,3, or 4 if correlation
# between adjacent data point is suspected)
j = 1
# Estimate noise assuming equidistant sampling (often a good approximation even
# if data are not strictly equidistant) and robust estimation (often advantageous
# in working with real data)
nstd, nstdstd = beq.betaSigma(y, N, j, returnMAD=True)
```

(continues on next page)
print("Input noise std: ", gstd)
print("Estimated noise std = \$5.3f +/- \$5.3f\" % (nstd, nstdstd))

Note: As the noise properties are usually not known a priori as in this example, it is advisable to compare two estimates, e.g., by increasing N by one to corroborate the validity of the result.

More detailed examples

Examples of Beta Sigma Noise Estimation

In the following, a number of examples demonstrate aspects of the $\beta\sigma$ procedure. Please see the corresponding paper for further discussion.

- Noise estimates with different orders of approximations (N)
- The effect of outliers: A case for robust estimation
- Equidistant vs. arbitrary sampling

Noise estimates with different orders of approximations (N)

The required order of approximation (N) depends on the level of variation in the data (or, equivalently, the sampling cadence in relation to the variability time scale of the signal). Comparing estimates obtained from various orders of approximation provides a useful cross-check for the plausibility of the result. In the example below, it is clearly seen that the zeroth order yields a too high estimate for the amplitude of noise because the signal varies too quickly.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyasl

def g(t):
    ""
    Function determining the behavior of the data.
    ""
    return 1.3 - 0.003*t + 1.3*np.sin(t/5.) * np.exp(-t/100.)

# Number of data points
nd = 200
ti = np.arange(float(nd))
gi = g(ti)
mdiff = np.max(np.abs(gi[1:] - gi[0:-1]))
print("Maximum absolute difference between consecutive values of g(t): \"", mdiff)

# Standard deviation of noise
istd = 0.02
```
print("Input standard deviation: ", istd)
print("Number of 'data points': ", nd)
print()

# Add Gaussian noise to data
yi = gi + np.random.normal(0.0, istd, nd)

# Create class instance for equidistant sampling
bseq = pyasl.BSEqSamp()

# Specify jump parameter (j) for construction of beta sample
j = 1

# Order of approximation to use
Ns = [0, 1, 2, 3]

# Use to store noise estimates
smads, dsmads = [], []

# Loop over orders of approximation between 0 and 3
for N in Ns:
    # Get estimates of standard deviation based on robust (MAD-based) estimator
    smad, dsmad = bseq.betaSigma(yi, N, j, returnMAD=True)
    print("Order of approximation (N): ", N)
    print(" Size of beta sample: ", len(bseq.betaSample))
    print(" Robust estimate of noise std: %6.3f +/- %6.3f" % (smad, dsmad))
    # Save result
    smads.append(smad)
    dsmads.append(dsmad)

# Plot g(t) and the synthetic data
plt.subplot(2, 1, 1)
plt.title("Data (top) and noise estimates (bottom)")
plt.plot(ti, gi, 'b.-', label="$g(t_i)$")
plt.errorbar(ti, yi, yerr=np.ones(nd)*istd, fmt='r+', label="$y_i$")
plt.legend()
plt.subplot(2, 1, 2)
plt.title("N=0 is insufficient")
plt.errorbar(Ns, smads, yerr=dsmads, fmt='k+', label="Noise estimates")
plt.plot([min(Ns)-0.5, max(Ns)+0.5], [istd]*2, 'k--', label="Input value")
plt.legend()
plt.xlabel("Order of approximation (N)")
plt.ylabel("Noise STD")
plt.tight_layout()
plt.show()
from __future__ import print_function, division
import numpy as np
from PyAstronomy import pyasl
import matplotlib.pylab as plt

# Number of data points
nd = 200

# Input standard deviation
istd = 0.2

# Create some synthetic data (Gaussian noise) with 
# input standard deviation.
y = np.random.normal(1.0, istd, nd)

# Introduce some outliers
# Number of outliers
no = 15
for _ in range(no):
    # Choose an index (could occur more than once)
    index = np.random.randint(0, high=nd)
    # Add point from normal distribution with
    # STD of 50
    y[index] = np.random.normal(1.0, 50.0)

# Create class instance for equidistant sampling
bseq = pyasl.BSEqSamp()

# Specify order of approximation (N) and jump parameter (j) for
# construction of beta sample
N = 0
j = 1
print("Order of approximation: ", N)
print("Jump parameter: ", j)
print()

# Get estimates of standard deviation based on MV estimator and ...
smv, dsmv = bseq.betaSigma(y, N, j, returnMAD=False)
# ... robust (MAD-based) estimator
smad, dsmad = bseq.betaSigma(y, N, j, returnMAD=True)

print("Input standard deviation: ", istd)
print("Number of 'data points': ", nd)
print("Size of beta sample: ", len(bseq.betaSample))
print()
print("Minimum-variance estimate: 
6.3f +/- 6.3f" % (smv, dsmv))
print("Robust estimate: 
6.3f +/- 6.3f" % (smad, dsmad))

plt.subplot(2, 1, 1)
plt.title("Synthetic data")
plt.plot(y, 'bp')
plt.subplot(2, 1, 2)
plt.title("Histogram of $\beta$ sample")
plt.hist(bseq.betaSample, 30)
plt.show()
Equidistant vs. arbitrary sampling

The classes \texttt{BSEqSamp()} and \texttt{BSArbSamp()} treat the cases of equidistant and arbitrary sampling of the signal explicitly. Equidistant sampling is technically more simple to treat and is often a good approximation \textit{even if} the actual sampling is not equidistant. This is true when there is a "not too complicated" transformation relating the actual and an equidistant sampling axes. The following example demonstrates a case, where equidistant sampling can be assumed but a higher order of approximation is required.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyasl

def g(t):
    """
    Function determining the behavior of the data.
    """
    return 1.3 - 10.0*t

# Number of data points
nd = 30

# Creating non-equidistant sampling axis (ti)
te = np.arange(float(nd))
ti = (te**3) / float(nd**2)

# Get values of g(t)
gi = g(ti)

# Standard deviation of noise
istd = 0.3

# Add Gaussian noise to data
yi = gi + np.random.normal(0.0, istd, nd)

print("Input standard deviation: ", istd)
print("Number of 'data points': ", nd)

# Create class instance for equidistant sampling
bseq = pyasl.BSEqSamp()

# Create class instance for arbitrary sampling
bsar = pyasl.BSArbSamp()

# Get estimates assuming equidistant and arbitrary sampling
# using \(N = 1\) and \(j = 1\). From the definition of \(g(t)\), \(N = 1\)
# will be sufficient for the case of arbitrary sampling, but
# not necessarily for (assumed) equidistant sampling.
smv_es, dsmv_es = bseq.betaSigma(yi, 1, 1)
smv_as, dsmv_as = bsar.betaSigma(ti, yi, 1, 1)

print("Estimates for N=1 and j=1")
print(" Equidistant sampling: \$5.3f +/- 5.3f\" % (smv_es, dsmv_es))
print(" Arbitrary sampling: \$5.3f +/- 5.3f\" % (smv_as, dsmv_as))
```

(continues on next page)
# Get estimates for N=2 and 3 assuming equidistant sampling

smv_es2, dsmv_es2 = bseq.betaSigma(yi, 2, 1)
smv_es3, dsmv_es3 = bseq.betaSigma(yi, 3, 1)

print("Estimates for N=2 and 3 based on equidistant sampling")
print(" N = 2: \$5.3f +/- 5.3f\$ \% (smv_es2, dsmv_es2))
print(" N = 3: \$5.3f +/- 5.3f\$ \% (smv_es3, dsmv_es3))

plt.subplot(2, 1, 1)
plt.title("Data with true sampling")
plt.plot(ti, gi, 'b-')
plt.errorbar(ti, yi, yerr=np.ones(nd)*istd, fmt='b+')
plt.subplot(2, 1, 2)
plt.title("Same data assuming equidistant sampling")
plt.plot(te, gi, 'r-')
plt.errorbar(te, yi, yerr=np.ones(nd)*istd, fmt='r+')
plt.tight_layout()
plt.show()

## API documentation

### Beta-Sigma API

#### BSBase (Base class for BSEqSamp and BSArbSamp)

```python
class PyAstronomy.pyasl.BSBase
```

Functionality to estimate noise from equidistantly and arbitrarily sampled data.

#### Methods

- **estimateStdMAD(x, mode)** Estimate standard deviation based on median absolute deviation (MAD)
- **meanUnbStd(x)** Mean and (unbiased) standard deviation of the mean.
- **stdUnbiased(y)** Get unbiased estimate of the standard deviation and its standard deviation.
- **stdc4(n)** Calculate c4 factor.
- **subsetIndexDissection(ndp, N, j)** Find array indices of (N+2)-length subsets.
- **subsetIndices(cd)** Compose indices sets defining the subsets to construct the beta sample.
- **variance(x, mode)** Estimate variance from sample

- **_checkJP (j)** Check validity of specified jump parameter.
- **_checkN (N)** Check validity of specified order of approximation.
- **estimateStdMAD(x, mode)** Estimate standard deviation based on median absolute deviation (MAD)

**Parameters**

---

54 Chapter 2. PyAstronomy's sub-packages
x  [array] Sample from which to determine the standard deviation.

mode  [string, {zm, em}] If ‘zm’, the population median is assumed to be zero. If ‘em’, the population median is estimated as the sample median.

Returns
std  [float] Estimate of the standard deviation

meanUnbStd(x)
Mean and (unbiased) standard deviation of the mean.

Parameters
x  [array] Sample from which to calculate mean and std

Returns
mean, std  [floats] Mean and unbiased standard deviation of the mean

stdUnbiased(y)
Get unbiased estimate of the standard deviation and its standard deviation.

Parameters
y  [array] Sample from which to estimate standard deviation.

Returns
Std  [float] Unbiased estimate of the standard deviation.
Std of std  [float] Standard deviation of the unbiased standard deviation estimator.

stdc4(n)
Calculate c4 factor.
The c4 factor is required to obtain an unbiased estimator for the standard deviation.
It is proportional to the factor B used in Kenney 1940, who started from a slightly different definition of the sample variance.

Parameters
n  [int] Number of points in the sample

Returns
c4, ln(c4)  [float] The c4 factor and its natural logarithm.

subsetIndexDissection(ndp, N, j)
Find array indices of (N+2)-length subsets.
Here, (N+2)-sized sets of indices are constructed, which are required in the construction of the beta sample.

Parameters
ndp  [int] Number of available data points
N  [int] Last order of the Taylor series taken into account. Chunk length will be N+2.
j  [int] Jump parameter (>0)

Returns
k-indices  [list] A list holding N+2 elements. Each element of the list is an array holding the indices of the data point no. one, two, ⋯, N+2 of the subsets required to construct the beta sample.
subsetIndices \((cd)\)
Compose indices sets defining the subsets to construct the beta sample.

Parameters

\(cd\) [list] The output of subsetIndexDissection

Returns

Subset indices [2d array] A 2d-array holding the indices of all subsets, arranged so that result\([i,::]\) holds the \(N+2\) indices pertaining to the \(i\)-th subset.

variance \((x, mode)\)
Estimate variance from sample

Parameters

\(x\) [array] sample

\(mode\) [string, \{n, nmo\}] Estimator to use: ‘n’ for \(1/n\) version with zero mean (not estimated) and ‘nmo’ for \(1/(n-1)\)

BSEqSamp (equidistant sampling)

class PyAstronomy.pyasl.BSEqSamp

Methods

betaSigma\((y, N, j[, ignoreNaN, returnMAD, ibs])\) Estimate standard deviation of noise term in data set \(y\).

estimateStdMAD\((x, mode)\) Estimate standard deviation based on median absolute deviation (MAD)

getBetaSample\((y, N, j)\) Construct the beta sample.

getBetaSampleShift\((y, N, j)\) Construct beta sample using shifting procedure.

geta_k\((N)\) Calculate the required coefficients \(a_k\)

geta_hok\((N)\) Calculate (auto)correlation function for beta sample

meanUnbStd\((x)\) Mean and (unbiased) standard deviation of the mean.

stdUnbiased\((y)\) Get unbiased estimate of the standard deviation and its standard deviation.

stdc4\((n)\) Calculate c4 factor.

subsetIndexDissection\((ndp, N, j)\) Find array indices of \((N+2)\)-length subsets.

subsetIndices\((cd)\) Compose indices sets defining the subsets to construct the beta sample.

variance\((x, mode)\) Estimate variance from sample

betaSigma \((y, N, j, ignoreNaN=True, returnMAD=False, ibs=False)\) Estimate standard deviation of noise term in data set \(y\).

It is explicitly assumed that the data are equidistantly sampled.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>betaSample</td>
<td>array</td>
<td>The beta sample</td>
</tr>
<tr>
<td>estimates</td>
<td>dict</td>
<td>Summary of the estimates obtained from the beta sample (bs).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“s2E”: variance estimate of the bs (expectation value of zero), “s2Evar”: variance of s2E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“sE”: Estimate of std of bs, “sEstd”: Std of sE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“s2”: variance estimate of the bs (mean estimated from bs), “s2var”: variance of s2, “s”: sqrt(s2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“sME”: Standard deviation based on MAD with zero expectation, “sMEstd”: Estimation of std of sME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“sM”: Standard deviation based on MAD with median estimated from bs, “sMstd”: Estimation of std of sM</td>
</tr>
</tbody>
</table>

**Parameters**

- y [array] Data values from which to estimate standard deviation of noise.
- N [int] Last order of the Taylor series to be taken into account.
- j [int] Jump parameter
- ignoreNaN [boolean, optional] If True (default), NaN values in the beta sample are ignored in the calculation.
- returnMAD [boolean, optional] If True, the estimate obtained using the MAD is returned instead of that of the MVU estimator (default is False). The standard error is estimated by scaling the standard deviation of the MVU estimator by a factor of 1.64.
- ibs [boolean, optional] If True, an independent beta sample is constructed. Default is False.

**Returns**

- Estimate of STD in beta sample and the STD of the estimate: float, float The standard deviation determined in the beta sample. Non-robust estimates sE and sEstd if returnMAD is False (default) or robust estimates sME and sMEstd if returnMAD is True. The estimates attribute holds a more comprehensive summary of the estimates.

**getBetaSample** (y, N, j)
Construct the beta sample.

**Parameters**

- y [array] Values from which to estimate noise
- N [int] Last order of the Taylor series taken into account
- j [int] Jump parameter (>0)

**Returns**

- Beta sample [array] An array holding all available beta values.

**getBetaSampleShift** (y, N, j)
Construct beta sample using shifting procedure.

**Parameters**

- y [array] Values from which to estimate noise
- N [int] Last order of the Taylor series taken into account
j  [int] Jump parameter (>0)

Returns

Beta sample  [array] An array holding all available beta values.

get_ak  (N)
Calculate the required coefficients (a_k)

Parameters

N  [int] Order of approximation

Returns

ak  [array] The coefficients

get_rhok  (N)
Calculate (auto)correlation function for beta sample

Parameters

N  [int] Order of approximation

Returns

correlation function  [array] Correlation function

BSArbSamp (arbitrary sampling)

class PyAstronomy.pyasl.BSArbSamp
Estimate noise in equidistantly sampled data.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>betaSigma(x, y, N, j[, ignoreNaN, ...])</td>
<td>Estimate standard deviation of noise term in data set y.</td>
</tr>
<tr>
<td>estimateStdMAD(x, mode)</td>
<td>Estimate standard deviation based on median absolute deviation (MAD)</td>
</tr>
<tr>
<td>getBetaSample(x, y, N, j)</td>
<td>Combine data points to calculate beta values.</td>
</tr>
<tr>
<td>getBetaSampleShift(x, y, N, j)</td>
<td>Construct beta sample using shifting procedure.</td>
</tr>
<tr>
<td>getCoeffsArbSamp(t[, gamma])</td>
<td>Calculate coefficients (ak) for arbitrary sampling.</td>
</tr>
<tr>
<td>meanUnbStd(x)</td>
<td>Mean and (unbiased) standard deviation of the mean.</td>
</tr>
<tr>
<td>stdUnbiased(y)</td>
<td>Get unbiased estimate of the standard deviation and its standard deviation.</td>
</tr>
<tr>
<td>stdc4(n)</td>
<td>Calculate c4 factor.</td>
</tr>
<tr>
<td>subsetIndexDissection(ndp, N, j)</td>
<td>Find array indices of (N+2)-length subsets.</td>
</tr>
<tr>
<td>subsetIndices(cd)</td>
<td>Compose indices sets defining the subsets to construct the beta sample.</td>
</tr>
<tr>
<td>variance(x, mode)</td>
<td>Estimate variance from sample</td>
</tr>
</tbody>
</table>

betaSigma  (x, y, N, j, ignoreNaN=True, returnMAD=False, ibs=False)
Estimate standard deviation of noise term in data set y.

In this implementation, arbitrary sampling is taken into account.

The method assigns the following attributes, which may be accessed after execution to work with the result:
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>betaSample</td>
<td>array</td>
<td>The beta sample</td>
</tr>
<tr>
<td>estimates</td>
<td>dict</td>
<td>Summary of the estimates obtained from the beta sample (bs). &quot;s2E&quot;: variance estimate of the bs (expectation value of zero), &quot;s2Evar&quot;: variance of s2E &quot;sE&quot;: Estimate of std of bs, &quot;sEstd&quot;: Std of sE &quot;s2&quot;: variance estimate of the bs (mean estimated from bs), &quot;s2var&quot;: variance of s2, &quot;s&quot;: sqrt(s2) &quot;sME&quot;: Standard deviation based on MAD with zero expectation, &quot;sMEstd&quot;: Estimation of std of sME &quot;sM&quot;: Standard deviation based on MAD with median estimated from bs, &quot;sMstd&quot;: Estimation of std of sM</td>
</tr>
</tbody>
</table>

### Parameters

- **x** [array] Sampling of the data.
- **y** [array] Data values from which to estimate standard deviation of noise.
- **N** [int] Last order of the Taylor series to be taken into account.
- **j** [int, optional] Jump parameter (default is one, i.e., consecutive data points are combined to estimate the noise).
- **ignoreNaN** [boolean, optional] If True (default), NaN values in the beta sample are ignored in the calculation.
- **returnMAD** [boolean, optional] If True, the estimate obtained using the MAD is returned instead of that of the MVU estimator (default is False). The standard error is estimated by scaling the standard deviation of the MVU estimator by a factor of 1.64.
- **ibs** [boolean, optional] If True, an independent beta sample is constructed. Default is False.

### Returns

**Estimate of STD in beta sample and the STD of the estimate:** float, float The standard deviation determined in the beta sample. Non-robust estimates sE and sEstd if returnMAD is False (default) or robust estimates sME and sMEstd if returnMAD is True. The estimates attribute holds a more comprehensive summary of the estimates.

**getBetaSample** *(x, y, N, j)*

Combine data points to calculate beta values.

### Parameters

- **x** [array] Sampling of data
- **y** [array] Values from which to estimate noise
- **N** [int] Last order of the Taylor series taken into account
- **j** [int] Jump parameter (>0)

### Returns

**Betas** [array] An array holding all available beta values.

**getBetaSampleShift** *(x, y, N, j)*

Construct beta sample using shifting procedure.
Parameters

x, y [array] Values from which to estimate noise
N [int] Last order of the Taylor series taken into account
j [int] Jump parameter (>0)

Returns

Beta sample [array] An array holding all available beta values.

`getCoeffsArbSamp(t, gamma=1.0)`
Calculate coefficients (ak) for arbitrary sampling.

Parameters

t [array] Sampling instants of the subset.
gamma [float, optional] Scaling of the coefficients (default is one).

Returns

ak [array] Set of coefficients.

SampCorr (sample correlation)

```python
class PyAstronomy.pyasl.SampCorr

Methods

get_nb(N, ss) BH46 Eq.
```

Outlier detection

Accounting for “outliers” can be an integral part in any kind of data analysis, yet, it remains basically unclear what such an outlier actually is. Citing Hawkins “an outlier is an observation that deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism”.

Below, we give algorithms to detect outliers based on the Generalized Extreme Studentized Deviate (ESD) test

- `generalizedESD`
- `pointDistGESD`

and polynomial fits

- `polyResOutlier`
- `slidingPolyResOutlier`
The generalized ESD test

`PyAstronomy.pyasl.generalizedESD(x, maxOLS, alpha=0.05, fullOutput=False, ubvar=False)`

Carry out a Generalized ESD Test for Outliers.

The Generalized Extreme Studentized Deviate (ESD) test for outliers can be used to search for outliers in a univariate data set, which approximately follows a normal distribution. A description of the algorithm is, e.g., given at Nist or [Rosner1983].

**Parameters**

- `maxOLS` [int] Maximum number of outliers in the data set.
- `alpha` [float, optional] Significance (default is 0.05).
- `fullOutput` [boolean, optional] Determines whether additional return values are provided. Default is False.
- `ubvar` [boolean, optional] If True, an unbiased estimate for the variance will be used in the calculation; this provides compatibility with the R implementation of NIST. If False (default), the maximum-likelihood variance estimate will be used.

**Returns**

- **Number of outliers** [int] The number of data points characterized as outliers by the test.
- **Indices** [list of ints] The indices of the data points found to be outliers.
- **R** [list of floats, optional] The values of the “R statistics”. Only provided if `fullOutput` is set to True.
- **L** [list of floats, optional] The lambda values needed to test whether a point should be regarded an outlier. Only provided if `fullOutput` is set to True.

**Example**

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyasl

# Convert data given at:
# to array.
x = np.array([float(x) for x in "-0.25 0.68 0.94 1.15 1.20 1.26 1.26 1.34 1.38 1.43 1.49 1.55 1.56 1.58 1.65 1.69 1.70 1.76 1.81 1.91 1.94 1.96 1.99 2.06 2.09 2.10 2.14 2.15 2.23 2.24 2.26 2.35 2.37 2.40 2.47 2.54 2.62 2.64 2.90 2.92 2.93 3.21 3.26 3.30 3.59 3.68 4.30 4.64 5.34 5.42 6.01".split()])

# Apply the generalized ESD
r = pyasl.generalizedESD(x, 10, 0.05, fullOutput=True)

print("Number of outliers: ", r[0])
print("Indices of outliers: ", r[1])
print("R Lambda")
for i in range(len(r[2])):
    print("%2d %8.5f %8.5f" % ((i+1), r[2][i], r[3][i]))
```

(continues on next page)
# Plot the "data"
plt.plot(x, 'b."
# and mark the outliers.
for i in range(len(x)):
    plt.plot(x[i], x[i], 'rv')
plt.show()

Distance-based outlier detection (e.g., for spectra)

The generalized ESD test requires approximate normal distribution for the data points, which—for example in the case of a spectrum—can be a harsh limitation.

This function applies the generalized ESD test to the distances between adjacent data points, which are then required to be distributed approximately normally. It will characterize a data point as an outlier, only if the distances to its right and left neighbor are abnormal as judged by the generalized ESD.

PyAstronomy.pysal.pointDistGESD(x, maxOLs, alpha=0.05, ubvar=False)

Search for outliers by comparing distance of adjacent points.

This method computes the “distance” between adjacent points, e.g., d = x[i] - x[i-1]. It then uses PyAstronomy.pysal.generalizedESD() to search for outliers in the list of distances. A point will be characterized as being an outlier, if (and only if) the distance to its left and right neighbor are abnormal.

Parameters
x [array] The data set to be searched for outliers.
maxOLs [int] The number of outliers. Note that the number passed to generalizedESD is actually 2*maxOLs.
alpha [float, optional] The significance level to be used in applying generalizedESD. The default is 0.05.
ubvar [boolean, optional] If True, an unbiased estimate for the variance will be used in the calculation; this provides compatibility with the R implementation of NIST. If False (default), the maximum-likelihood variance estimate will be used.

Returns
n [int] The number of outliers detected.
Indices [list of ints] The indices of the points found to be outliers.

Example

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyasl

# Get some data
x = np.random.normal(0., 0.1, 50)

# Introduce outliers
x[27] = 1.0
```
x[43] = -0.66

# Run distance based outlier detection
r = pyasl.pointDistGESD(x, 5)

print("Number of outliers detected: ", r[0])
print("Indices of these outliers: ", r[1])

plt.plot(x, 'b.'
for i in range(len(r[1])):
    plt.plot(r[1][i], x[r[1][i]], 'rp')
plt.show()

Outlier detection based on polynomial fit

The algorithm implemented here is based on a polynomial fit to the data. After the fit is subtracted, the residuals are calculated. Based on their standard deviation, points with residuals deviating by more than the specified number of standard deviations from the fit are identified. Implementations with a single polynomial fit and a sliding window are available.

PyAstronomy.pyasl.polyResOutlier(x, y, deg=0, stdlim=3.0, controlPlot=False, fullOutput=False, mode='both')

Simple outlier detection based on residuals.
This algorithm fits a polynomial of the specified degree to the data, subtracts it to find the residuals, determines the standard deviations of the residuals, and, finally, identifies all points with residuals further than the specified number of standard deviations from the fit.

Parameters

x, y [arrays] The abscissa and ordinate of the data.

deg [int, optional] The degree of the polynomial to be fitted. The default is 0, i.e., a constant.

stdlim [float, optional] The number of standard deviations acceptable for points not categorized as outliers.

mode [string, {both, above, below}] If ‘both’ (default), outliers may be located on both sides of the polynomial. If ‘above/below’, outliers are only expected above/below it.

cntrolPlot [boolean, optional] If True, a control plot will be generated showing the location of outliers (default is False).

fullOutput [boolean, optional] If True, the fitted polynomial and the resulting model will be returned.

Returns

indiin [array] The indices of the points not being categorized as outliers.

indiout [array] Indices of the outlier points.

p [array, optional] Coefficients of the fitted polynomial (only returned if fullOutput is True).

model [array, optional] The polynomial model (only returned if fullOutput is True).

PyAstronomy.pyasl.slidingPolyResOutlier(x, y, points, count=1, deg=0, stdlim=3.0, controlPlot=False, dx=1, mode='both')

Outlier detection based on polynomial fit in sliding box.
This algorithm fits a polynomial of the specified degree to a sliding chunk of the data, subtracts it to find the residuals, determines the standard deviations of the residuals, and, finally, identifies all points with residuals further than the specified number of standard deviations from the fit.

The length of the chunk is determined by `points`. In each step, the chunk is advanced by `dx` data points (default is one). To be finally marked as an outlier, a point must be detected as an outlier in at least `count` instances, when the chunk slides over it. By default, a single such detection is sufficient to establish its outlier status.

**Parameters**

- `x, y` [arrays] The abscissa and ordinate of the data.
- `points` [int] Number of points for the sliding box
- `count` [int, optional] Number of “slides” in which the point shall deviate from the fit by the stdlim
- `deg` [int, optional] The degree of the polynomial to be fitted. The default is 0, i.e., a constant.
- `stdlim` [float, optional] The number of standard deviations acceptable for points not categorized as outliers.
- `mode` [string, {both, above, below}] If ‘both’ (default), outliers may be located on both sides of the polynomial. If ‘above/below’, outliers are only expected above/below it.
- `controlPlot` [boolean, optional] If True, a control plot will be generated showing the location of outliers (default is False).
- `dx` [int, optional] The number of data points by which the chunk is advanced in each step.

**Returns**

- `indiin` [array] The indices of the points not categorized as outliers.
- `indiout` [array] Indices of the outlier points.

**Example: polyResOutlier**

The `polyResOutlier` method relies on a single polynomial fit.

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pylab as plt

# Generate some "data"
x = np.arange(100)
y = np.random.normal(x*0.067, 1.0, len(x))

# Introduce an outliers
y[14] = -5.0
y[67] = +9.8

# Find outliers based on a linear (deg = 1) fit.
# Assign outlier status to all points deviating by
# more than 3.0 standard deviations from the fit,
# and show a control plot.
iin, iout = pyasl.polyResOutlier(x, y, deg=1, stdlim=3.0, controlPlot=True)

# What about the outliers
print("Number of outliers: ", len(iout))
```

(continues on next page)
print("Indices of outliers: ", iout)

# Remove outliers
xnew, ynew = x[iin], y[iin]

# Plot result (outlier in red)
plt.plot(x, y, 'r.')
plt.plot(xnew, ynew, 'bp')
plt.show()

Example: slidingPolyResOutlier

The *slidingPolyResOutlier* method uses polynomial fits in a sliding window to detect outliers.

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pylab as plt

# Generate some "data"
x = np.arange(100)
y = np.random.normal(x*0.067 + 0.01*x**2, 1.0, len(x))

# Introduce an outlier
y[14] = -5.0
y[67] = +9.8

# Find outliers based on a linear (deg = 1) fit.
# Assign outlier status to all points deviating by
# more than 3.0 standard deviations from the fit,
# and show a control plot.
iin, iout = pyasl.slidingPolyResOutlier(x, y, 20, deg=1, stdlim=3.0, controlPlot=True)

# What about the outliers
print("Number of outliers: ", len(iout))
print("Indices of outliers: ", iout)

# Remove outliers
xnew, ynew = x[iin], y[iin]

# Plot result (outlier in red)
plt.plot(x, y, 'r.')
plt.plot(xnew, ynew, 'bp')
plt.show()
```

References

INTEP interpolation algorithm

PyAstronomy.pyasl.intep(x, y, xinter, boundsError=True, fillValue=None)

The INTEP interpolation algorithm
The INTEP interpolation algorithm is described by Hill 1982, PDAO 16, 67 (“Intep - an Effective Interpolation Subroutine”). The implementation at hand is based on the FORTRAN code stated therein.

The aim of the algorithm is to imitate the curve “an experienced scientist” would draw through a given set of points.

**Parameters**

- **x** [array] Independent values.
- **y** [array] Dependent values.
- **xinter** [array] Values at which to interpolate the tabulated data given by x and y.
- **boundsError** [boolean, optional] If True, an exception will be raised if values need to be extrapolated beyond the limits of the given tabulated data. Values beyond the limits are simply replaced with the closest valid value available, which might not be a good approximation. Set this flag to False to suppress the exception.
- **fillValue** [float, optional] If given (i.e., not None), this value will be used to represent values outside of the given bounds. Note that **boundsError** must be set False for this to have an effect. For instance, use np.NaN.

**Returns**

- **Interpolated values** [array] Interpolated values at the locations specified by **xinter**.

**Example**

```python
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pyplot as plt

# Create some tabulated data
x = np.arange(10.)
y = np.sin(x/5.*2.*np.pi)

# Choose the values at which to interpolate
xx = np.arange(120.)/10.-2.0
# Interpolate and suppress the exception that
# would indicate that some of our requested
# values (xx) are beyond the range covered by x.
yy = pyasl.intep(x, y, xx, boundsError=False)

# Plot the result
plt.plot(x, y, 'bp')
plt.plot(xx, yy, 'r--')
plt.show()
```

**Finding extreme point by parabolic approximation**

PyAstronomy.pyasl.\texttt{quadExtreme}(x, y, \texttt{mode}='max', \texttt{dp}=(1, 1), \texttt{exInd}=None, \texttt{fullOutput}=False, \texttt{fullPoint}=False)

Find the extreme (minimum or maximum) by means of a parabolic fit.

This function searches for the maximum (or minimum) in the given ordinate values, fits a second-order polynomial to the surroundings of that point, and, finally, determines the thus approximated abscissa value of the extreme point.
Parameters

- **x** [array] Abscissa values.
- **y** [array] Ordinate values.
- **mode** [string, [min, max], optional] Determines whether a minimum or a maximum is searched for. The default is a maximum.
- **dp** [tuple with two integers, optional] Determines the width around the extreme point, which will be used in the fitting. The default is one point to the right and left, i.e., \( dp = (1,1) \).
- **extInd** [integer, optional] If given, the function will assume that this is the index of the extreme point and not search for it.
- **fullOutput** [boolean, optional] If True, the output will also cover the points used in the fitting and the resulting polynomial.
- **fullPoint** [boolean, optional] If True, the return value *epos* will be a tuple holding the abscissa and ordinate values of the extreme point. The default is False.

Returns

- **epos** [float or tuple] Position of the extreme point. If *fullPoint* is True, a tuple with the abscissa and ordinate values of the extreme point.
- **mi** [int] The index of the extreme point (maximum or minimum).
- **xb** [array, optional] Only returned if *fullOutput* is True. The abscissa values used in the polynomial fit. Note the the x-value of the extreme point has been subtracted.
- **yb** [array, optional] Only returned if *fullOutput* is True. The ordinate values used in the polynomial fit.
- **p** [numpy polynomial, optional] Only returned if *fullOutput* is True. The best-fit polynomial. Note that the fit refers to the \( xb \) axis, where the x-value of the extreme point has been subtracted.

Example

```python
from __future__ import print_function, division
import numpy as np
from PyAstronomy import pyasl
import matplotlib.pyplot as plt

# Create some data (a Gaussian)
x = np.arange(100.0)
y = np.exp(-(x-50.2714)**2/(2.*5.**2))

# Find the maximum
epos, mi = pyasl.quadExtreme(x, y, mode="max")
print("Maximum found at index: ", mi, ", value at maximum: ", y[mi])
print("Maximum found by parabolic fit: ", epos)
print()

# Find the maximum, use a wider range for the parabolic fit.
print("Using 5 points to each side of the maximum")
epos, mi = pyasl.quadExtreme(x, y, mode="max", dp=(5, 5))
print("Maximum found at index: ", mi, ", value at maximum: ", y[mi])
```

(continues on next page)
print("Maximum found by parabolic fit: ", epos)
print()

# Do as above, but get the full output
print("Using 2 points to each side of the maximum")
epos, mi, xb, yb, p = pyasl.quadExtreme(x, y, mode="max", dp=(2, 2), fullOutput=True)

# Evaluate polynomial at a number of points.
# Note that, internally, the x-value of the extreme point has
# been subtracted before the fit. Therefore, we need to re-shift
# it in the plot.
newx = np.linspace(min(xb), max(xb), 100)
model = np.polyval(p, newx)

# Plot the "data"
plt.plot(x, y, 'bp')
# Mark the points used in the fitting (shifted, because xb is shifted)
plt.plot(xb+x[mi], yb, 'rp')
# Overplot the model (shifted, because xb is shifted)
plt.plot(newx+x[mi], model, 'r--')
plt.show()

The Singular Value Decomposition (SVD)

PyA's SVD class provides the means to carry out a singular value decomposition as is, e.g., useful to recover line profiles.

class PyAstronomy.pyasl.SVD
    Use Singular Value Decomposition (SVD) to obtain broadening function.

    The technique is, e.g., described in Rucinski 1999, “Determination of Broadening Functions Using the Singular-Value Decomposition (SVD) Technique” and references therein.

    Methods

    decompose(template, bn)  Carry out the SVD of the “design matrix”.
    getBroadeningFunction(flux[, wlimit, asarray])  Calculate the broadening function.
    getModel(broad[, asarray, modelIndices])  Calculates a model resulting from template and broadening function.
    getRVAxis(binsize, refWvl)  Calculate a radial velocity axis for the broadening function.
    getSingularValues()  Access the singular values.

    decompose (template, bn)
    Carry out the SVD of the “design matrix”.

    This method creates the “design matrix” by applying a bin-wise shift to the template and uses numpy’s svd algorithm to carry out the decomposition.

    The design matrix, des is written in the form: “des = u * w * transpose(v)”. The matrices des, w, and u are stored in homonymous attributes.

    Parameters
**template** [array or matrix] The template with respect to which the broadening function shall be calculated.

**bn** [int] Width (number of elements) of the broadening function. Needs to be odd.

**getBroadeningFunction** *(flux, wlimit=0.0, asarray=False)*

Calculate the broadening function.

**Note:** The *decompose* method has to be called first. On this call, the template is specified.

**Parameters**

- **flux** [array or matrix] The observed function (flux).
- **wlimit** [float, optional] A limit to be applied to the singular values. Values smaller than the specified limit will be discarded in calculating the broadening function.
- **asarray** [bool, optional] If True, the broadening function will be returned as an array instead of a matrix. The default is False.

**Returns**

- **Broadening function** [matrix, array] The broadening function. The return type can be set by the *asarray* flag.

**getModel** *(broad, asarray=False, modelIndices=False)*

Calculates a model resulting from template and broadening function.

**Parameters**

- **broad** [array or matrix] The broadening function
- **asarray** [bool, optional] If True, the broadening function will be returned as an array instead of a matrix. The default is False.
- **modelIndices** [bool, optional] If True, the method returns also an array of indices, which refer to the template, for which the model is calculated. Note that the model cannot be calculated at the edges. The default is False.

**getRVAxis** *(binsize, refWvl)*

Calculate a radial velocity axis for the broadening function.

**Note:** Here, it is assumed that the broadening function refers to a spectrum in wavelength space.

**Parameters**

- **binsize** [float] Size of spectral bins.
- **refWvl** [float] The reference wavelength.

**Returns**

- **Radial velocity axis** [array] An array containing the radial velocity shift pertaining to the individual bins in the broadening function. The unit is km/s.

**getSingularValues** ()

Access the singular values.

**Returns**
Singular values [matrix] The singular values.

Example: Delta functions and rotational broadening

```python
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyasl

# Get some "data"
wvl = np.arange(5000., 5010., 0.02)
template = np.ones(len(wvl))

# There are two sharp lines in the template
template[100] = 0.5
template[115] = 0.3
# Apply rotational broadening to the delta spectrum
nflux = pyasl.rotBroad(wvl, template, 0.5, 23.45, edgeHandling="firstlast")

# Carry out decomposition
svd = pyasl.SVD()
# Use 51 bins for the width of the broadening function.
# Needs to be an odd number.
svd.decompose(template, 51)
# Obtain the broadening function needed to
# recover "observed" spectrum. Note that the
# edges (51/2 bins) will actually be neglected.
b = svd.getBroadeningFunction(nflux)
# Get the model, which results from the broadening
# function and the template; obtain the indices
# where it applies, too.
m, mind = svd.getModel(b, modelIndices=True)

# Plot the outcome
plt.plot(b, 'bp-')
plt.plot(mind, m, 'r.' )
plt.plot(nflux, 'g--')
plt.show()
```

Example: Adding noise and neglecting singular values

```python
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyasl

# Get some "data"
wvl = np.arange(5000., 5010., 0.02)
template = np.ones(len(wvl))

# There are two sharp lines in the template
template[100] = 0.5
template[115] = 0.3
# Apply rotational broadening to the delta spectrum
nflux = pyasl.rotBroad(wvl, template, 0.5, 23.45, edgeHandling="firstlast")
nflux += np.random.normal(0., 0.005, len(nflux))
```

(continues on next page)
# Carry out decomposition
svd = pyasl.SVD()
svd.decompose(template, 51)

# Access the singular values
sv = svd.getSingularValues()

# Calculate the reduced chi square as a function of the number
# of singular values neglected in the calculation of the
# model.
chi = []
for i in range(1, len(sv), 5):
    b = svd.getBroadeningFunction(nflux, wlimit=sorted(sv)[i])
    m, mind = svd.getModel(b, modelIndices=True, asarray=True)
    chi.append(((nflux[mind] - m)**2/0.005**2).sum() / len(mind))

plt.title("Reduced $\chi^2$ vs. number of neglected singular values")
plt.plot(range(1, len(sv), 5), chi, 'bp-')
plt.show()

Spectral routines

Wavelength conversion (air and vacuum)

PyA provides tools to convert wavelengths from air into vacuum conditions and vice versa:

- airtovac2
- vactoair2

These functions allow to use the conversions specified by:

- Peck and Reeder 1972, J. Opt. Soc. 62 no. 8
- Ciddor 1996, Applied Optics 35 no. 9

**Warning:** The function airtovac and vactoair are based on the formula by Edlen 1953. It seems that air wavelengths (or wavenumbers) are used, where vacuum wavenumbers should be used.

- airtovac
- vactoair

Moreover, you can use:

- specAirVacConvert

to convert from vacuum into air conditions or vice versa, but retain the input wavelength axis.

**Usage examples for airtovac2 and vactoair2**
Example: From air to vacuum and back

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np

# Define wavelength array
wvl = np.arange(10) + 5000.0
print("Input wavelengths: ", wvl)

# Convert wavelength in air to wavelength in vacuum. By default, the conversion specified by Ciddor 1996 are used.
wvlVac = pyasl.airtovac2(wvl)
print("Wavelength in vacuum: ", wvlVac)

# Convert wavelength from vacuum to air
wvlAir = pyasl.vactoair2(wvlVac)
print("Wavelength in air: ", wvlAir)
```

Example: Compare Edlen and Ciddor conversions

```python
from PyAstronomy import pyasl
import numpy
import matplotlib.pylab as plt

# Compare the Edlen and Ciddor conversions
wvl = numpy.arange(3000, 10000, 1.0)
wvlVacCiddor = pyasl.airtovac2(wvl, mode="ciddor")
wvlVacEdlen = pyasl.airtovac2(wvl, mode="edlen53")

plt.subplot(2, 1, 1)
plt.title("Difference in air wavelength (Ciddor-Edlen)")
plt.ylabel("dWvl [\AA"])
plt.xlabel("Vacuum wvl [\AA"])
plt.plot(wvl, wvlVacCiddor-wvlVacEdlen, 'b.-')
plt.subplot(2, 1, 2)
plt.title("Difference in air wavelength (Ciddor-Edlen, in RV)")
plt.ylabel("dRV [m/s]"
plt.xlabel("Vacuum wvl [\AA"])
plt.plot(wvl, (wvlVacCiddor-wvlVacEdlen)/wvlVacCiddor*299792458., 'b.-')
plt.show()
```

API documentation: `airtovac2` and `vactoair2`

PyAstronomy.pyasl.**airtovac2**(wvl, **mode**='ciddor', **kwargs)

Converts wavelength in air into wavelength in vacuum.

This function is a wrapper around the `RefractiveIndexAV` class.

**Parameters**

- **wvl** : float or array  Wavelength in vacuum [\AA].
mode [string, {edlen53, peckReeder, ciddor}, optional] The source for the wavelength conversion. The default is “ciddor”.

kwargs [dict] Additional parameters handed to the airtovac method of RefractiveIndexAV.

Returns

wvl [float or array] Wavelength in vacuum [Å].

PyAstronomy.pyasl.vactoair2(wvl, mode='ciddor', **kwargs)

Converts wavelength in vacuum into wavelength in air.

This function is a wrapper around the RefractiveIndexAV class.

Parameters

wvl :float or array Wavelength in vacuum [Å].

mode [string, {edlen53, peckReeder, ciddor}, optional] The source for the wavelength conversion. The default is “ciddor”.

kwargs [dict] Additional parameters handed to the vactoair method of RefractiveIndexAV.

Returns

wvl [float or array] Wavelength in air [Å].

Obtaining the refractive index

Although the conversion of wavelength between air and vacuum conditions is the primary application targeted here, what is really needed to carry out the conversion is the refractive index.

class PyAstronomy.pyasl.RefractiveIndexAV (mode='ciddor')

Refractive index for conversion between air- and vacuum wavelengths.

The conversion is based on the formulae provided either by:

• Peck and Reeder 1972, J. Opt. Soc. 62 no. 8
• Ciddor 1996, Applied Optics 35 no. 9

This class provides the routine refractiveIndex, vactoair, and airtovac to convert between wavelengths in air and vacuum.

The method refractiveIndex points to either of _edlen53, _ciddor, or _peckReeder depending on the mode chosen.

To conversion from vacuum to air wavelengths is given by the respective formulae provided by the authors of the above papers, who all specify the refractive index of “standard air” as a function of vacuum wave number. The reverse conversion from air into vacuum is done by applying an iterative scheme. Standard air is at 15 degrees Celsius, dry, and at a pressure of 750 mmHg = 101325 Pa. Ciddor assumes a CO2 concentration of 450 ppm, while Edlen, and Peck and Reeder have assumed 300 ppm.

Parameters

mode [string, {edlen53, peckReeder, ciddor}, optional] The source for the wavelength conversion. The default is “ciddor”.

Methods

2.1. Python AstroLib
**airtovac**(wvl[, precision, maxiter]) Convert air wavelength into wavelength in vacuum.

**vactoair**(wvl, **kwargs) Convert vacuum wavelength into wavelength in air.

**_ciddor**(wvl, xc=None)
Refractive index given by Ciddor 1996.

**Parameters**
- wvl [float or array] Vacuum wavelength [A]
- xc [float, optional] Concentration of CO2 [ppm]

**Returns**
- n [float or array] Refractive index

**_edlen53**(wvl)
Refractive index according to Edlen 1953.

**Parameters**
- wvl [float or array] Vacuum wavelength [A]

**Returns**
- n [float or array] The refractive index.

**_peckReeder**(wvl)
Refractive index from Peck and Reeder 1972.

**Parameters**
- wvl [float or array] Vacuum wavelength [A]

**Returns**
- n [float or array] The refractive index.

**airtovac**(wvl, precision=1e-12, maxiter=30, **kwargs)
Convert air wavelength into wavelength in vacuum.

An iterative scheme is applied to invert the formula.

**Parameters**
- wvl [float or array] Wavelength in vacuum [A].
- precision [float, optional] The target precision beyond which iteration should be stopped; default is 1e-12.
- maxiter [int, optional] Maximum number of iterations used to invert the conversion; default is 30.
- kwargs [dict] Additional keywords eventually needed in the call to the refractiveIndex function (depends on the specified ‘mode’).

**Returns**
- wvl [float or array] Wavelength in air.

**vactoair**(wvl, **kwargs)
Convert vacuum wavelength into wavelength in air.

**Parameters**
- wvl [float or array] Wavelength in vacuum [A].
kwarg: Additional keywords eventually needed in the call to the refractiveIndex function (depends on the specified ‘mode’).

**Returns**

- `wvl` [float or array] Wavelength in air.

**Example: Obtain refractive index of standard air**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np

# Define wavelength array
wvl = np.arange(10) + 5000.0

# Obtain refractive index according to Ciddor 1996
ri = pyasl.RefractiveIndexAV(mode="ciddor")
n = ri.refractiveIndex(wvl)

print("Wavelength and 1.0 - Refractive index of 'standard air':")
for w, nc in zip(wvl, n):
    print("{0:5.1f} {1:10.8e}".format(w, nc-1.0))
```

**Transform spectrum, but retain wavelength axis**

PyAstronomy.pyasl.specAirVacConvert(wvl, flux, direction, dontUse2=False)

Transform spectrum between air and vacuum wavelengths.

In contrast to `airtovac()` and `vactoair()`, which only transform the wavelength axis, this function converts the wavelength axis and interpolates the shifted spectrum to retain the input wavelength axis.

Due to the shift, a fraction of the spectrum is lost on one side, while there is no valid information on the other. Therefore, the function returns both a new flux array obtained by linear interpolation and an array of valid indices, i.e., those indices in the new spectrum, which are not NaN.

**Warning:** Linear interpolation of the spectrum can affect its noise properties. This function does nothing to prevent or correct for that effect.

**Parameters**

- `wvl` [array] Input wavelength axis.
- `flux` [array] Input flux array.
- `direction` [string, {"airtovac","vactoair"}] If the direction is "vactoair", the input spectrum is assumed to be obtained in vacuum conditions. The opposite is true otherwise.
- `dontUse2` [boolean, optional] If True, the now deprecated function airtovac and vactoair will be used. Otherwise (default), airtovac2 and vactoair2 will be used with default settings.

**Returns**

- `New flux` [array] The shifted flux array, defined on the input wavelength axis.
- `Valid indices` [array] The array of valid, i.e., not NaN, indices.
Example: Convert spectrum from vacuum into air conditions

```python
from PyAstronomy import pyasl
from PyAstronomy import funcFit as fuf
import numpy as np
import matplotlib.pyplot as plt

# Create a spectrum ...
g = fuf.GaussFit1d()
g["A"] = -0.1
g["sig"] = 0.03
g["mu"] = 5004.4752
g["off"] = 1.0

wvl = np.linspace(5000., 5010., 1000)
flux = g.evaluate(wvl)

# ... and add some noise
flux += np.random.normal(0., 0.02, len(wvl))

# Assume that this spectrum is one observed
# or modeled in vacuum conditions. We want
# to convert it into a spectrum in air, but
# retain the old wavelength axis.
airflux, vind = pyasl.specAirVacConvert(wvl, flux,
                                       direction="vactoair")

# Plot the result
plt.plot(wvl, flux, "b.-")
plt.plot(wvl, airflux, "r.-")
plt.show()
```

API documentation: `airtovac` and `vactoair`

**Note:** These functions are now considered deprecated.

Both functions have been ported from IDL’s astrolib.

**PyAstronomy.pyasl.** `airtovac` (*wave, depWarn=True*)

Convert air wavelengths to vacuum wavelengths

**Warning:** The conversion implemented here is based on the older formulae given by Edlen 1953. Furthermore, it seems that wave numbers in air are used, where vacuum wave numbers should be used, which, however, produces only a second-order deviation. Consider using `airtovac2()` instead.

**Parameters**

- `wave` [float, array] The wavelength in air [Angstrom]
- `depWarn` [boolean, optional] If True (default), a deprecation warning will be given.

**Returns**

- `Wavelength` [array] Wavelength in vacuum [Angstrom]
PyAstronomy Documentation, Release 0.15.2

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: AIRTOVAC

PURPOSE: Convert air wavelengths to vacuum wavelengths

EXPLANATION: Wavelengths are corrected for the index of refraction of air under standard conditions. Wavelength values below 2000 A will not be altered. Uses the IAU standard for conversion given in Morton (1991 Ap.J. Suppl. 77, 119)

CALLING SEQUENCE: AIRTOVAC, WAVE

INPUT/OUTPUT:

WAVE - Wavelength in Angstroms, scalar or vector WAVE should be input as air wavelength(s), it will be returned as vacuum wavelength(s). WAVE is always converted to double precision upon return.

EXAMPLE: If the air wavelength is W = 6056.125 (a Krypton line), then AIRTOVAC, W yields an vacuum wavelength of W = 6057.8019

METHOD: See Morton (Ap. J. Suppl. 77, 119) for the formula used

REVISION HISTORY Written W. Landsman November 1991 Converted to IDL V5.0 W. Landsman September 1997

PyAstronomy.pyasl.vactoair(wave, depWarn=True)
Convert vacuum wavelengths to air wavelengths

Warning: The conversion implemented here is based on the older formulae given by Edlen 1953. Furthermore, it seems that wave numbers in air are used, where vacuum wave numbers should be used, which, however, produces only a second-order deviation. Consider using vactoair2() instead.

Parameters

wave [float, array] The wavelength in vacuum [Angstrom]

Returns

Wavelength [array.] Wavelength in air [Angstrom]

depWarn [boolean, optional] If True (default), a deprecation warning will be given.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: VACTOAIR
**PURPOSE:** Convert vacuum wavelengths to air wavelengths

**EXPLANATION:** Corrects for the index of refraction of air under standard conditions. Wavelength values below 2000 Å will not be altered. Accurate to about 0.005 Å

**CALLING SEQUENCE:** VACTOAIR, WAVE

**INPUT/OUTPUT:**

- **WAVE** - Wavelength in Angstroms, scalar or vector. WAVE should be input as vacuum wavelength(s), it will be returned as air wavelength(s). WAVE is always converted to double precision.

**EXAMPLE:** If the vacuum wavelength is \( W = 2000 \), then

```idl```
IDL> VACTOAIR, W
```

yields an air wavelength of \( W = 1999.353 \) Angstroms.

**METHOD:** An approximation to the 4th power of inverse wavenumber is used. See IUE Image Processing Manual Page 6-15.

**REVISION HISTORY** Written, D. Lindler 1982 Documentation W. Landsman Feb. 1989 Converted to IDL V5.0 W. Landsman September 1997

---

**Barycentric velocity correction**

The `baryvel()` and `baryCorr()` functions allow to calculate Earth's helio- and barycentric motion and project it onto a given direction toward a star. The `helcorr()` includes a barycentric correction including the effect caused by the rotating Earth.

PyA's `baryvel` function is a port of its IDL Astrolib's counterpart. The `helcorr` function has been ported from the REDUCE package (see Piskunov & Valenti 2002, A&A 385, 1095).

PyAstronomy.pyasl.baryvel(dje, deq)

Calculate helio- and barycentric velocity.

---

**Note:** The “JPL” option present in IDL is not provided here.

---

**Parameters**

- **dje** [float] Julian ephemeris date
- **deq** [float] Epoch of mean equinox of helio- and barycentric velocity output. If `deq` is zero, `deq` is assumed to be equal to `dje`.

**Returns**

- **dvelh** [array] Heliocentric velocity vector [km/s].
- **dvelb** [array] Barycentric velocity vector [km/s].

---

**Notes**

**Note:** This function was ported from the IDL Astronomy User’s Library.

---

**IDL - Documentation**
NAME: BARYVEL

PURPOSE: Calculates heliocentric and barycentric velocity components of Earth.

EXPLANATION: BARYVEL takes into account the Earth-Moon motion, and is useful for radial velocity work to an accuracy of ~1 m/s.

CALLING SEQUENCE: BARYVEL, dje, deq, dvelh, dvelb, [ JPL = ]

INPUTS: DJE - (scalar) Julian ephemeris date. DEQ - (scalar) epoch of mean equinox of dvelh and dvelb. If deq=0 then deq is assumed to be equal to dje.

OUTPUTS: DVELH: (vector(3)) heliocentric velocity component. in km/s DVELB: (vector(3)) barycentric velocity component. in km/s

The 3-vectors DVELH and DVELB are given in a right-handed coordinate system with the +X axis toward the Vernal Equinox, and +Z axis toward the celestial pole.

OPTIONAL KEYWORD SET:

JPL - if /JPL set, then BARYVEL will call the procedure JPLEPHINTERP to compute the Earth velocity using the full JPL ephemeris. The JPL ephemeris FITS file JPLEPH.405 must exist in either the current directory, or in the directory specified by the environment variable ASTRO_DATA. Alternatively, the JPL keyword can be set to the full path and name of the ephemeris file. A copy of the JPL ephemeris FITS file is available in

http://idlastro.gsfc.nasa.gov/ftp/data/

PROCEDURES CALLED: Function PREMAT() – computes precession matrix JPLEPHREAD, JPLEPHINTERP, TDB2TDT - if /JPL keyword is set

NOTES: Algorithm taken from FORTRAN program of Stumpff (1980, A&A Suppl, 41,1) Stumpf claimed an accuracy of 42 cm/s for the velocity. A comparison with the JPL FORTRAN planetary ephemeris program PLEPH found agreement to within about 65 cm/s between 1986 and 1994

If /JPL is set (using JPLEPH.405 ephemeris file) then velocities are given in the ICRS system; otherwise in the FK4 system.

EXAMPLE:

Compute the radial velocity of the Earth toward Altair on 15-Feb-1994 using both the original Stumpf algorithm and the JPL ephemeris

IDL> jdcnv, 1994, 2, 15, 0, jd ;==> JD = 2449398.5 IDL> baryvel, jd, 2000, vh, vb ;Original algorithm


IDL> ra = ten(19,50,46.77)*15!/RADEG ;RA in radians IDL> dec = ten(08,52,3.5)!/RADEG ;Dec in radians IDL> v = vb[0]*cos(dec)*cos(ra) + $ ;Project velocity toward star vb[1]*cos(dec)*sin(ra) + vb[2]*sin(dec)

REVISION HISTORY: Jeff Valenti, U.C. Berkeley Translated BARVEL.FOR to IDL. W. Landsman, Cleaned up program sent by Chris McCarthy (SfSU) June 1994 Converted to IDL V5.0 W. Landsman September 1997 Added /JPL keyword W. Landsman July 2001 Documentation update W. Landsman Dec 2005
PyAstronomy.pyasl.\texttt{baryCorr} \((jd, ra, dec, deq=0.0)\)

Calculate barycentric correction.

This function uses the \texttt{baryvel()} function to calculate the helio- and barycentric motion of the Earth and projects it onto the direction to the star.

\textbf{Note:} Positive return values indicate that the Earth moves toward the star.

\begin{description}
\item[Parameters]
\begin{itemize}
\item \texttt{jd} \ (float) The time at which to calculate the correction.
\item \texttt{ra} \ (float) Right ascension in degrees.
\item \texttt{dec} \ (float) Declination in degrees.
\item \texttt{deq} \ (float, optional) The mean equinox of barycentric velocity calculation (see \texttt{bryvel()}). If zero, it is assumed to be the same as \texttt{jd}.
\end{itemize}
\end{description}

\textbf{Returns}

\begin{description}
\item[Projected heliocentric velocity] \ (float) Heliocentric velocity toward star \([\text{km/s}]\)
\item[Projected barycentric velocity] \ (float) Barycentric velocity toward star \([\text{km/s}]\)
\end{description}

PyAstronomy.pyasl.\texttt{helcorr} \((obs\_long, obs\_lat, obs\_alt, ra2000, dec2000, jd, debug=False)\)

Calculate barycentric velocity correction.

This function calculates the motion of an observer in the direction of a star. In contrast to \texttt{baryvel()} and \texttt{baryCorr()}, the rotation of the Earth is taken into account.

The coordinates (ra2000, dec2000) are precessed to the epoch defined by \texttt{jd}. These coordinates are used in the calculation.

\textbf{Note:} This function was ported from the REDUCE IDL package. See Piskunov & Valenti 2002, A&A 385, 1095 for a detailed description of the package and/or visit http://www.astro.uu.se/~piskunov/RESEARCH/REDUCE/

\begin{description}
\item[Warning:] Contrary to the original implementation the longitude increases toward the East and the right ascension is given in degrees instead of hours. The JD is given as is, in particular, nothing needs to be subtracted.
\end{description}

\begin{description}
\item[Parameters]
\begin{itemize}
\item \texttt{obs\_long} \ (float) Longitude of observatory (degrees, eastern direction is positive)
\item \texttt{obs\_lat} \ (float) Latitude of observatory \([\text{deg}]\)
\item \texttt{obs\_alt} \ (float) Altitude of observatory \([\text{m}]\)
\item \texttt{ra2000} \ (float) Right ascension of object for epoch 2000.0 \([\text{deg}]\)
\item \texttt{dec2000} \ (float) Declination of object for epoch 2000.0 \([\text{deg}]\)
\item \texttt{jd} \ (float) Julian date for the middle of exposure.
\end{itemize}
\end{description}

\textbf{Returns}
**Barycentric correction** [float] The barycentric correction accounting for the rotation of the Earth, the rotation of the Earth’s center around the Earth-Moon barycenter, and the motion of the Earth-Moon barycenter around the center of the Sun [km/s].

**HJD** [float] Heliocentric Julian date for middle of exposure.

**Notes**

**IDL REDUCE - Documentation**

Calculates heliocentric Julian date, barycentric and heliocentric radial velocity corrections from:

**INPUT:**
<OLBSLON> Longitude of observatory (degrees, western direction is positive)
<OBSLAT> Latitude of observatory (degrees)
<OBSALT> Altitude of observatory (meters)
<RA2000> Right ascension of object for epoch 2000.0 (hours)
<DE2000> Declination of object for epoch 2000.0 (degrees)
<JD> Julian date for the middle of exposure [DEBUG=] set keyword to get additional results for debugging

**OUTPUT:**
<CORRECTION> barycentric correction - correction for rotation of earth,
rotation of earth center about the earth-moon barycenter, earth-moon barycenter about the center of the Sun.

<HJD> Heliocentric Julian date for middle of exposure

Algorithms used are taken from the IRAF task noao.astutils.rvcorrect and some procedures of the IDL Astrolib are used as well. Accuracy is about 0.5 seconds in time and about 1 m/s in velocity.

History: written by Peter Mittermayer, Nov 8,2003 2005-January-13 Kudryavtsev Made more accurate calculation of the sidereal time.

Conformity with MIDAS compute/barycorr is checked.

2005-June-20 Kochukhov Included precession of RA2000 and DEC2000 to current epoch

**Example: Applying baryvel and baryCorr**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

jd = 2.476468576e6
heli, bary = pyasl.baryvel(jd, deq=2000.0)

print("Earth's velocity at JD: ", jd)
print("Heliocentric velocity [km/s]: ", heli)
print("Barycentric velocity [km/s] : ", bary)

# Coordinates of Sirius
ra = 101.28715535
dec = -16.71611587

vh, vb = pyasl.baryCorr(jd, ra, dec, deq=2000.0)
print("Barycentric velocity of Earth toward Sirius: ", vb)
```
Example: Obtaining a barycentric correction

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Coordinates of European Southern Observatory
# (Coordinates of UT1)
longitude = 289.5967661
latitude = -24.62586583
altitude = 2635.43

# Coordinates of HD 12345 (J2000)
ra2000 = 030.20313477
dec2000 = -12.87498346

# (Mid-)Time of observation
jd = 2450528.2335

# Calculate barycentric correction (debug=True show various intermediate results)
corr, hjd = pyasl.helcorr(longitude, latitude, altitude, ra2000, dec2000, jd, debug=True)

print("Barycentric correction [km/s]: ", corr)
print("Heliocentric Julian day: ", hjd)
```

Planck’s radiation law

PyAstronomy.pyasl.planck(T, lam=None, nu=None)

Evaluate Planck’s radiation law.

Depending on whether wavelength or frequency is specified as input, the function evaluates:

\[
B_\nu = \frac{2\pi h \nu^3}{c^2} \frac{1}{e^{\frac{h \nu}{k T}} - 1}
\]

or

\[
B_\lambda = \frac{2\pi hc^2}{\lambda^5} \frac{1}{e^{\frac{hc}{\lambda k T}} - 1}.
\]

If lambda is given (in meters), the output units are W/(m^2 m). To convert into erg/(cm^2 A s), the output has to be multiplied by a factor of 1e-7.

Parameters

- **T** [float] Temperature in Kelvin.
- **lam** [float or array, optional] Wavelength in meters.
- **nu** [float or array, optional] Frequency in Hz.

Returns

**Spectral radiance** [float or array] Depending on whether lam or nu were specified, returns the spectral radiance per area and wavelength or frequency. The unit (SI) will be W/(m^2 m) if lam was given and W/(m^2 Hz) if nu was specified.
Example

```python
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy.pyasl import planck

# Define wavelength in meters
lam = np.arange(1000.0*1e-10, 20000.*1e-10, 20e-10)

# Get the Planck spectrum in [W/(m**2 m)] for a temperature of 7000 K
s7 = planck(7000., lam=lam)
# Get the Planck spectrum in [W/(m**2 m)] for a temperature of 5000 K
s5 = planck(5000., lam=lam)

# Convert into erg/(cm**2 * A * s)
s5erg = s5 * 1e-7
s7erg = s7 * 1e-7

# Integrate the spectrum and compare with Stefan-Boltzmann law
i5 = np.sum(s5) * (lam[1] - lam[0])
i7 = np.sum(s7) * (lam[1] - lam[0])

print("5000 K integral: %.3e W/m**2 (Stefan-Boltzmann predicts %.3e W/m**2)" % (i5,
                                                                                   (5.67e-8*5000.**4)))
print("7000 K integral: %.3e W/m**2 (Stefan-Boltzmann predicts %.3e W/m**2)" % (i7,
                                                                                   (5.67e-8*7000.**4)))

plt.xlabel("Wavelength [$\AA$]")
plt.ylabel("Flux [erg/cm$^2$/A/s]")
plt.plot(lam*1e10, s5erg, 'r-')
plt.plot(lam*1e10, s7erg, 'b-')
plt.show()
```

Cross-correlation

PyAstronomy.pyasl.crosscorrRV(w, f, tw, tf, rvmin, rvmax, drv, mode='doppler', skipedge=0, edgeTapering=None)

Cross-correlate a spectrum with a template.

The algorithm implemented here works as follows: For each RV shift to be considered, the wavelength axis of the template is shifted, either linearly or using a proper Doppler shift depending on the mode. The shifted template is then linearly interpolated at the wavelength points of the observation (spectrum) to calculate the cross-correlation function.

Parameters

- **w** [array] The wavelength axis of the observation.
- **f** [array] The flux axis of the observation.
- **tw** [array] The wavelength axis of the template.
- **tf** [array] The flux axis of the template.
- **rvmin** [float] Minimum radial velocity for which to calculate the cross-correlation function [km/s].
- **rvmax** [float] Maximum radial velocity for which to calculate the cross-correlation function [km/s].
- **drv** [float] Increment of radial velocity.
- **mode** [string] Mode for the RV shift: 'linear' or 'doppler'.
- **skipedge** [int] Number of pixels to skip on the edge. (default: 0)
- **edgeTapering** [None] Type of edge tapering to use. (default: None)
rvmax  [float] Maximum radial velocity for which to calculate the cross-correlation function [km/s].

drv  [float] The width of the radial-velocity steps to be applied in the calculation of the cross-correlation function [km/s].

mode  [string, {lin, doppler}, optional] The mode determines how the wavelength axis will be modified to represent a RV shift. If “lin” is specified, a mean wavelength shift will be calculated based on the mean wavelength of the observation. The wavelength axis will then be shifted by that amount. If “doppler” is specified (the default), the wavelength axis will properly be Doppler-shifted.

skipedge  [int, optional] If larger zero, the specified number of bins will be skipped from the begin and end of the observation. This may be useful if the template does not provide sufficient coverage of the observation.

edgeTapering  [float or tuple of two floats] If not None, the method will “taper off” the edges of the observed spectrum by multiplying with a sine function. If a float number is specified, this will define the width (in wavelength units) to be used for tapering on both sides. If different tapering widths shall be used, a tuple with two (positive) numbers must be given, specifying the width to be used on the low- and high wavelength end. If a nonzero ‘skipedge’ is given, it will be applied first. Edge tapering can help to avoid edge effects (see, e.g., Gullberg and Lindegren 2002, A&A 390).

Returns

dRV  [array] The RV axis of the cross-correlation function. The radial velocity refer to a shift of the template, i.e., positive values indicate that the template has been red-shifted and negative numbers indicate a blue-shift of the template. The numbers are given in km/s.

CC  [array] The cross-correlation function.

See also:
An algorithm for finding the extreme points by parabolic approximation (quadExtreme()).

Example: Cross-correlation with a Gaussian

```
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pyplot as plt

# Create the template
tw = np.linspace(5000, 5010, 1000)
tf = np.exp(-(tw-5004.0)**2/(2.*0.1**2))

# Create data, which are not that well sampled
dw = np.linspace(5000, 5010, 200)
df = np.exp(-(dw-5004.17)**2/(2.*0.1**2))

# Plot template and data
plt.title("Template (blue) and data (red)")
plt.plot(tw, tf, 'b.-')
plt.plot(dw, df, 'r.-')
plt.show()

# Carry out the cross-correlation.
```
# The RV-range is -30 - +30 km/s in steps of 0.6 km/s.
# The first and last 20 points of the data are skipped.
rv, cc = pyasl.crosscorrRV(dw, df, tw, tf, -30., 30., 30./50., skipedge=20)

# Find the index of maximum cross-correlation function
maxind = np.argmax(cc)

print("Cross-correlation function is maximized at dRV = ", rv[maxind], " km/s")
if rv[maxind] > 0.0:
    print(" A red-shift with respect to the template")
else:
    print(" A blue-shift with respect to the template")

plt.plot(rv, cc, 'bp-')
plt.plot(rv[maxind], cc[maxind], 'ro')
plt.show()

Doppler shifting a spectrum

PyAstronomy.pyasl.\texttt{dopplerShift}(wvl, flux, v, edgeHandling=None, fillValue=None, vlim=0.05)

Doppler shift a given spectrum.

An algorithm to apply a Doppler shift to a spectrum. The idea here is to obtain a shifted spectrum without losing the wavelength axis. Therefore, this function, first, calculates the shifted wavelength axis and, second, obtains the new, shifted flux array at the old, unshifted wavelength points by linearly interpolating. No relativistic effects are considered.

Due to the shift, some bins at the edge of the spectrum cannot be interpolated, because they are outside the given input range. The default behavior of this function is to return numpy.NAN values at those points. One can, however, specify the \texttt{edgeHandling} parameter to choose a different handling of these points.

If “firstlast” is specified for \texttt{edgeHandling}, the out-of-range points at the red or blue edge of the spectrum will be filled using the first (at the blue edge) and last (at the red edge) valid point in the shifted, i.e., the interpolated, spectrum.

If “fillValue” is chosen for edge handling, the points under consideration will be filled with the value given through the \texttt{fillValue} keyword.

\textbf{Warning:} Shifting a spectrum using linear interpolation has an effect on the noise of the spectrum. No treatment of such effects is implemented in this function.

**Parameters**

- \texttt{wvl} [array] Input wavelengths in A.
- \texttt{flux} [array] Input flux.
- \texttt{v} [float] Doppler shift in km/s
- \texttt{edgeHandling} [string, \{“fillValue”, “firstlast”\}, optional] The method used to handle the edges of the output spectrum.
- \texttt{fillValue} [float, optional] If the “fillValue” is specified as edge handling method, the value used to fill the edges of the output spectrum.
vlim [float, optional] Maximal fraction of the speed of light allowed for Doppler shift, v. Default is 0.05.

Returns

nflux [array] The shifted flux array at the old input locations.

wlprime [array] The shifted wavelength axis.

Note on linear interpolation and noise

Linear interpolation can (apparently) affect the noise in the data set. Say we start with a number of iid (identically and independently distributed) residuals, which obey a Gaussian distribution with some fixed standard deviation characterizing the noise. This is what is typically and often implicitly assumed in noise estimation. Now we generate a new data set by averaging adjacent points, i.e., we apply a special case of linear interpolation with the new data points half way between the initial ones. If we now estimate the noise in the new data set under the same iid assumption, we will find that it decreased by a factor of $\sqrt{2}$. Now, it seems unlikely that we can increase the information content in our data by interpolation alone. In fact, we must acknowledge that the iid assumption does no longer hold, because the information previously contained in a single data point is now, by construction, contained in two of the new, interpolated data points, so that these points must no longer be considered independent.

While the effect is strongest for the case outlined above, it also holds for the more general case, where interpolated data points are not half way between the original ones.

Example

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import matplotlib.pyplot as plt
import numpy as np

# Create a "spectrum" with 0.01 Å binning ...
wvl = np.linspace(6000., 6100., 10000)
# ... a gradient in the continuum ...
flux = np.ones(len(wvl)) + (wvl/wvl.min())*0.05
# ... and a Gaussian absorption line
flux -= np.exp(-(wvl-6050.)**2/(2.*0.5**2))*0.05

# Shift that spectrum redward by 20 km/s using
# "firstlast" as edge handling method.
nflux1, wlprime1 = pyasl.dopplerShift(wvl, flux, 20., edgeHandling="firstlast")

# Shift the red-shifted spectrum blueward by 20 km/s, i.e.,
# back on the initial spectrum.
nflux2, wlprime = pyasl.dopplerShift(wvl, nflux1, -20.,
                                       edgeHandling="fillValue", fillValue=1.0)

# Check the maximum difference in the central part
indi = np.arange(len(flux)-200) + 100
print("Maximal difference (without outer 100 bins): ",
      max(np.abs(flux[indi]-nflux2[indi])))

# Plot the outcome
plt.title("Initial (blue), shifted (red), and back-shifted (green) spectrum")
plt.plot(wvl, flux, 'b-')
```

plt.plot(wwl, nflux1, 'r.-')
plt.plot(wwl, nflux2, 'g.-')
plt.show()

Broadening mechanisms and routines

Instrumental (Gaussian kernel) broadening

PyAstronomy.pyasl.broadGaussFast(x, y, sigma, edgeHandling=None, maxsig=None)

Apply Gaussian broadening.

This function broadens the given data using a Gaussian kernel.

Parameters

x, y [arrays] The abscissa and ordinate of the data.
sigma [float] The width (i.e., standard deviation) of the Gaussian profile used in the convolution.
edgeHandling [string, {None, "firstlast"}, optional] Determines the way edges will be handled.
   If None, nothing will be done about it. If set to "firstlast", the spectrum will be extended by
   using the first and last value at the start or end. Note that this is not necessarily appropriate.
   The default is None.
maxsig [float, optional] The extent of the broadening kernel in terms of standard deviations.
   By default, the Gaussian broadening kernel will be extended over the entire given spectrum,
   which can cause slow evaluation in the case of large spectra. A reasonable choice could,
   e.g., be five.

Returns

Broadened data [array] The input data convolved with the Gaussian kernel.

PyAstronomy.pyasl.instrBroadGaussFast(wvl, flux, resolution, edgeHandling=None, fullout=False, maxsig=None)

Apply Gaussian instrumental broadening.

This function broadens a spectrum assuming a Gaussian kernel. The width of the kernel is determined by
the resolution. In particular, the function will determine the mean wavelength and set the Full Width at Half
Maximum (FWHM) of the Gaussian to (mean wavelength)/resolution.

Parameters

wvl [array] The wavelength
flux [array] The spectrum
resolution [int] The spectral resolution.
edgeHandling [string, {None, "firstlast"}, optional] Determines the way edges will be handled.
   If None, nothing will be done about it. If set to "firstlast", the spectrum will be extended by
   using the first and last value at the start or end. Note that this is not necessarily appropriate.
   The default is None.
fullout [boolean, optional] If True, also the FWHM of the Gaussian will be returned.
maxsig [float, optional] The extent of the broadening kernel in terms of standard deviations.
   By default, the Gaussian broadening kernel will be extended over the entire given spectrum,
   which can cause slow evaluation in the case of large spectra. A reasonable choice could,
   e.g., be five.
Returns

**Broadened spectrum** [array] The input spectrum convolved with a Gaussian kernel.

**FWHM** [float, optional] The Full Width at Half Maximum (FWHM) of the used Gaussian kernel.

**Example of usage**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import matplotlib.pylab as plt
import numpy as np

# Set up an input spectrum
x = np.linspace(5000.0, 5100.0, 1000)
y = np.ones(x.size)

# Introduce some delta-peaked lines
y[165] = 0.7
y[187] = 0.3
y[505] = 0.1
y[610] = 0.1
y[615] = 0.7

# Apply Gaussian instrumental broadening, setting the resolution to 10000.
r, fwhm = pyasl.instrBroadGaussFast(x, y, 10000,
                                   edgeHandling="firstlast", fullout=True)

# Apply Gaussian instrumental broadening, setting the resolution to 10000.
# Limit the extent of the Gaussian broadening kernel to five standard deviations.
r2, fwhm = pyasl.instrBroadGaussFast(x, y, 10000,
                                   edgeHandling="firstlast", fullout=True, maxsig=5.0)

print("FWHM used for the Gaussian kernel: ", fwhm, " Å")

# Plot the output
plt.plot(x, r, 'r--p', label="Broadened curve (full)")
plt.plot(x, r2, 'k:', label="Broadened curve (5 stds)")
plt.plot(x, y, 'b-', label="Input")
plt.legend(loc=4)
plt.show()
```

**Rotational broadening**

The functions here implement rotational broadening as described in D.F. Gray’s book “The Observation and Analysis of Stellar Photospheres”. Stellar limb-darkening is accounted for using the linear law.

To apply rotational broadening, either `rotBroad()` or `fastRotBroad()` can be used. The “fast” algorithm uses a single broadening kernel, which is appropriate as long as the wavelength range remains small. Otherwise the slower but accurate algorithm implemented in `rotBroad()` should be used.

```python
PyAstronomy.pyasl.rotBroad(wvl, flux, epsilon, vsini, edgeHandling='firstlast')
```

Apply rotational broadening to a spectrum.
This function applies rotational broadening to a given spectrum using the formulae given in Gray’s “The Observation and Analysis of Stellar Photospheres”. It allows for limb darkening parameterized by the linear limb-darkening law.

The `edgeHandling` parameter determines how the effects at the edges of the input spectrum are handled. If the default option, “firstlast”, is used, the input spectrum is internally extended on both sides; on the blue edge of the spectrum, the first flux value is used and on the red edge, the last value is used to extend the flux array. The extension is neglected in the return array. If “None” is specified, no special care will be taken to handle edge effects.

Note: Currently, the wavelength array as to be regularly spaced.

Parameters

- `wvl` [array] The wavelength array [Å]. Note that a regularly spaced array is required.
- `vsini` [float] Projected rotational velocity [km/s].
- `epsilon` [float] Linear limb-darkening coefficient (0-1).
- `edgeHandling` [string, {“firstlast”, “None”}] The method used to handle edge effects.

Returns

- **Broadened spectrum** [array] An array of the same size as the input flux array, which contains the broadened spectrum.

Example

Apply rotational broadening to a Gaussian and check the result equivalent widths.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy import funcFit as fuf
from PyAstronomy import pyasl
import scipy.integrate as sci

# Create a spectrum with a single Gaussian line using funcFit's GaussFit1d object.
# Note that this object is not used for fitting here, but only a calculate a Gaussian.
g = fuf.GaussFit1d()
g["mu"] = 5005.
g["A"] = -0.1

g["sig"] = 0.1
g["off"] = 1.0

# Evaluate the spectrum with 0.01 Å bin size
wvl = np.linspace(5003., 5007., 400)
flux = g.evaluate(wvl)

# Obtain the broadened spectrum using
# vsini = 13.3 km/s and no limb-darkening
```
rflux = pyasl.rotBroad(wvl, flux, 0.0, 13.3)

# Obtain the broadened spectrum using
# vsini = 13.3 km/s and strong limb-darkening
lflux = pyasl.rotBroad(wvl, flux, 0.9, 13.3)

# Check that the area of the line did not change
# in response to the broadening
print("Initial EW [A]: ", 4. - sci.trapz(flux, wvl))
print("After broadening without LD: ", 4. - sci.trapz(rflux, wvl))
print("After broadening with LD: ", 4. - sci.trapz(lflux, wvl))

# Plot the results
plt.title("Rotational broadening")
plt.xlabel("Wavelength [Å]")
plt.ylabel("Normalized flux")
plt.plot(wvl, flux, 'b-')
plt.plot(wvl, rflux, 'r-')
plt.plot(wvl, lflux, 'g-')
plt.show()

**fastRotBroad—a faster algorithm**

PyAstronomy.pyasl. fastRotBroad (wvl, flux, epsilon, vsini, effWvl=None)

Apply rotational broadening using a single broadening kernel.

The effect of rotational broadening on the spectrum is wavelength dependent, because the Doppler shift depends on wavelength. This function neglects this dependence, which is weak if the wavelength range is not too large.

**Note:** numpy.convolve is used to carry out the convolution and “mode = same” is used. Therefore, the output will be of the same size as the input, but it will show edge effects.

**Parameters**

- **wvl** [array] The wavelength
- **flux** [array] The flux
- **epsilon** [float] Linear limb-darkening coefficient
- **vsini** [float] Projected rotational velocity in km/s.
- **effWvl** [float, optional] The wavelength at which the broadening kernel is evaluated. If not specified, the mean wavelength of the input will be used.

**Returns**

- **Broadened spectrum** [array] The rotationally broadened output spectrum.

**Example: Using the fast algorithm**

```python
import numpy as np
import matplotlib.pylab as plt
```
from PyAstronomy import pyasl

# Create data with a Gaussian absorption line
wvl = np.arange(4999., 5011., 0.04)
flux = np.zeros(len(wvl))

# The Gaussian
A = -0.05
s = 0.1
mu = 5004.1635788
flux += A/np.sqrt(2.*np.pi*s**2) * np.exp(-(wvl-mu)**2/(2.*s**2))

# Apply the fast algorithm and ...
bfast = pyasl.fastRotBroad(wvl, flux, 0.81, 11.37)
# ... the slower one
bslow = pyasl.rotBroad(wvl, flux, 0.81, 11.37)

plt.xlabel("Wvl [A]"")
plt.ylabel("Flux [au]"")
plt.title("Initial spectrum (black), fast (blue), slow (red, shifted)"")
plt.plot(wvl, flux, 'k.-')
plt.plot(wvl, bfast, 'b.-')
plt.plot(wvl, bslow+0.01, 'r.-')
plt.show()

Thermal broadening

PyAstronomy.pyasl.thermalBroadeningWidth(lam0, T, m=None, fwhm=True)
Calculate the width for thermal broadening.

Thermal motion of particles causes a Doppler broadening of the line profile. The resulting line profile is Gaussian with FWHM given by

\[ fwhm = \frac{\lambda_0}{\sqrt{8k_B T \ln(2)/mc^2}} \]

See, e.g., http://hyperphysics.phy-astr.gsu.edu/hbase/atomic/broaden.html

Parameters

- **lam0** [float] Wavelength at which to calculate the width.
- **T** [float] Temperature [K].
- **m** [float, optional] Mass of the particles [kg]. If not specified, the proton mass is assumed.
- **fwhm** [boolean, optional] If True (default), the FWHM of the Gaussian broadening kernel will be returned. Otherwise, the standard deviation is returned.

Returns

- **Width** [float] The width of the Gaussian broadening kernel. By default, the FWHM is returned.

PyAstronomy.pyasl.tempFromthermalBroadeningWidth(lam0, width, m=None, awidth=0.0)
Calculate the temperature required to obtain thermal broadening width.
Thermal motion of particles causes Doppler broadening of the line profile. The resulting line profile is Gaussian with FWHM given by

\[ f_{\text{wm}} = \lambda_0 \sqrt{\frac{8k_B T \ln(2)}{mc^2}} \]

See, e.g., http://hyperphysics.phy-astr.gsu.edu/hbase/atomic/broaden.html

Here, the relation is reversed, so that the temperature, T, can be obtained from the width. The relation is only strictly valid in the case of Gaussian lines.

**Note:** Units of \( \lambda_0 \), width, and \( a_{\text{width}} \) have to be consistent.

### Parameters

- \( \lambda_0 \) [float] Wavelength at which to width (FWHM) was measured.
- \( \text{width} \) [float] Measured width of the feature (FWHM)
- \( m \) [float, optional] Mass of the particles [kg]. If not specified, the proton mass is assumed.
- \( a_{\text{width}} \) [float] Additional width contributed by other effects such as instrumental resolution. The thermal width, ft, is obtained by quadratic subtraction, i.e., \( ft^2 = width^2 - a_{\text{width}}^2 \). Default is zero.

### Returns

- **Temperature** [float] The temperature required to achieve the specified broadening width.
- **Thermal width** [float] The thermal width used in the calculations (may be modified if \( a_{\text{width}} \) has been specified.

#### Example

```python
from PyAstronomy import pyasl

w0 = 6564.0
T = 9567.0

linefwhm = pyasl.thermalBroadeningWidth(w0, T)
tbroad = pyasl.tempFromthermalBroadeningWidth(w0, linefwhm, awidth=0.0)

print("Line width [FWHM]: \$5.2f" % linefwhm)
print("Thermal broadening temperature: \$6.1f" % tbroad)
```

**Deredden flux vector (unred)**

\texttt{PyAstronomy.pyasl.unred} (\texttt{wave, flux, ebv, R_V=3.1, LMC2=False, AVGLMC=False})

Deredden a flux vector using the Fitzpatrick (1999) parameterization

### Parameters

- \( \text{wave} \) [array] Wavelength in Angstrom
- \( \text{flux} \) [array] Calibrated flux vector, same number of elements as wave.
**ebv** [float, optional] Color excess E(B-V). If a negative eb is supplied, then fluxes will be reddened rather than dereddened. The default is 3.1.

**AVGLMC** [boolean] If True, then the default fit parameters c1,c2,c3,c4,\(\gamma\),x0 are set to the average values determined for reddening in the general Large Magellanic Cloud (LMC) field by Misselt et al. (1999, ApJ, 515, 128). The default is False.

**LMC2** [boolean] If True, the fit parameters are set to the values determined for the LMC2 field (including 30 Dor) by Misselt et al. Note that neither AVGLMC nor LMC2 will alter the default value of R\(_V\), which is poorly known for the LMC.

**Returns**

**new_flux** [array] Dereddened flux vector, same units and number of elements as input flux.

**Notes**

*Note:* This function was ported from the IDL Astronomy User’s Library.

---

**IDL - Documentation**

**PURPOSE:** Deredden a flux vector using the Fitzpatrick (1999) parameterization

**EXPLANATION:** The R-dependent Galactic extinction curve is that of Fitzpatrick & Massa (Fitzpatrick, 1999, PASP, 111, 63; astro-ph/9809387). Parameterization is valid from the IR to the far-UV (3.5 microns to 0.1 microns). UV extinction curve is extrapolated down to 912 Angstroms.

**CALLING SEQUENCE:**

```idl```
FM_UNRED, wave, flux, ebv, [ funred, R_V = , /LMC2, /AVGLMC, ExtCurve= gamma =, x0=, c1=, c2=, c3=, c4= ]
```idl```

**INPUT:**

- **WAVE** - wavelength vector (Angstroms)
- **FLUX** - calibrated flux vector, same number of elements as WAVE

If only 3 parameters are supplied, then this vector will updated on output to contain the dereddened flux.

- **EBV** - color excess E(B-V), scalar. **If a negative EBV is supplied,** then fluxes will be reddened rather than dereddened.

**OUTPUT:**

- **FUNRED** - unreddened flux vector, same units and number of elements as FLUX

**OPTIONAL INPUT KEYWORDS**

- **R_V** - scalar specifying the ratio of total to selective extinction \(R(V) = A(V) / E(B - V)\). If not specified, then R = 3.1 Extreme values of R(V) range from 2.3 to 5.3

- **/AVGLMC** - if set, then the default fit parameters c1,c2,c3,c4,\(\gamma\),x0 are set to the average values determined for reddening in the general Large Magellanic Cloud (LMC) field by Misselt et al. (1999, ApJ, 515, 128)

- **/LMC2** - if set, then the fit parameters are set to the values determined
for the LMC2 field (including 30 Dor) by Misselt et al. Note that neither 
/VGLMC or /LMC2 will alter the default value of R_V which is poorly known 
for the LMC.

The following five input keyword parameters allow the user to customize the adopted 
extinction curve. For example, see Clayton et al. (2003, ApJ, 588, 871) for examples 
of these parameters in different interstellar environments.

x0 - Centroid of 2200 A bump in microns (default = 4.596) 
gamma - Width of 2200 A 
bump in microns (default =0.99) 
c3 - Strength of the 2200 A bump (default = 3.23) 
c4 - FUV curvature (default = 0.41) 
c2 - Slope of the linear UV extinction component 
(d = -0.824 + 4.717/R)

c1 - Intercept of the linear UV extinction component  (default = 2.030 - 3.007*c2

Example of usage

```python
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pylab as plt

# Approximate a solar spectrum using a Planck 
# function with a temperature of 5778 K between 
# 3000 A and 8000 A.
wvl = np.arange(3000., 8000., 1.0)
flux = pyasl.planck(T=5778., lam=wvl*1e-10)

# Deredden the spectrum assuming ebv=0.1
fluxUnred = pyasl.unred(wvl, flux, ebv=0.1, R_V=3.1)

# Plot the result
plt.title("Reddened flux (red) and dereddened flux (blue)")
plt.plot(wvl, flux, 'r--')
plt.plot(wvl, fluxUnred, 'b--')
plt.show()
```

Convert damping constant into line width

Convert damping constant into natural line-width.

PyAstronomy.pyasl.convertDampingConstant (gamma, wavelength)

Convert damping constant into line-width in wavelength units.

The inverse lifetime, 1/τ, is related to the damping constant, Γ, and the Einstein coefficients, A, according to

\[
\frac{1}{\tau_u} = \Gamma_{tu} = \sum_{j < u} A_{tj} + \sum_{j < l} A_{uj},
\]

where u and l denote the upper and lower state of the transition (see, e.g., “Der neue Kosmos”, Unsoeld und 
Baschek, 7th edition). The damping coefficient, gamma, determines the FWHM in units of circular frequency. 
The natural line width (FWHM of the Lorentzian profile) in wavelength units is then given by

\[
\frac{\lambda^2}{2\pi c} \Gamma.
\]
**gamma** [float] The damping constant [1/s]. The damping constant may be determined by summing all relevant Einstein coefficients as shown above.

**wavelength** [float] Wavelength of the transition in Å.

**Returns**

**Natural line-width** [float] The Full Width at Half Maximum of the Lorentzian [cm].

**Example**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Einstein coefficient relevant for hydrogen LyA
gLyA = 6.258085e8

print("Width of H LyA line at 1215.67 Å = \$e cm" % 
      pyasl.convertDampingConstant(gLyA, 1215.67))
```

**Flux-photon conversion**

PyAstronomy.pyasl.`flux2photons`(wave, flux)
Convert flux (erg/s) to photons/s.

**Parameters**

- `wave` [float, array] The wavelength in Angstrom.

**Returns**

- `photons` [array] Number of photons/s for the given input flux

PyAstronomy.pyasl.`photons2flux`(wave, photons)
Convert photons/s to flux (erg/s).

**Parameters**

- `wave` [float, array] The wavelength in Angstrom.

**Returns**

- `flux` [array] Flux in erg/s for the given photon flux

**Example**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Wavelength in Angstrom
wvl = 4000.
# Flux in erg/s
flux = 1.5e-14
```
# Convert into photons
photons = pyasl.flux2photons(wvl, flux)

# How many photons is this?
print("%g erg/s at %g A correspond to %g photons/s" \ 
  % (flux, wvl, photons))

# Converting back
flux2 = pyasl.photons2flux(wvl, photons)
print("%g photons/s at %g A correspond to %g erg/s" \ 
  % (photons, wvl, flux2))

Read/write 1d spectrum from/to fits file

The routines \texttt{read1dFitsSpec()} and \texttt{write1dFitsSpec()} provide simple interfaces for reading and writing one-dimensional fits spectra with pyfits (astropy.io.fits).

Reading fits spectrum

\texttt{PyAstronomy.pyasl.read1dFitsSpec(fn, hdu=0, fullout=False, CRPIX1=None, keymap={})} 

Read a simple 1d spectrum from fits file.

Reads a 1d-spectrum from a file and constructs the associated wavelength axis. To this end, the expression: \( \text{wvl} = ((\text{np.arange(N)} + 1.0) - \text{CRPIX1}) \times \text{CDELT1} + \text{CRVAL1} \) will be evaluated, where \( N \) is the number of bins and \( \text{CRPIX1}, \text{CDELT1}, \text{and CRVAL1} \) are header keywords.

Parameters

- \texttt{fn} [string] Filename
- \texttt{hdu} [int, optional] The number of the HDU to be used. The default is 0, i.e., the primary HDU.
- \texttt{fullout} [boolean, optional] If True, the header keywords used to construct the wavelength axis will be returned. The default is False.
- \texttt{CRPIX1} [int, optional] Can be used to circumvent missing CRPIX1 entry.
- \texttt{keymap} [dict, optional] Can be used to map header keywords

Returns

- \texttt{wvl} [array] The wavelength array.
- \texttt{flx} [array] The flux array.

Examples

\begin{verbatim}
from PyAstronomy import pyasl
wvl, flx = pyasl.read1dFitsSpec("mySpectrum.fits")
\end{verbatim}
Writing fits spectrum

PyAstronomy.pyasl.write1dFitsSpec(fn, flux, wvl=None, waveParams=None, fluxErr=None, header=None, clobber=False, refFileName=None, refFileExt=0)

Write a 1d spectrum with equidistant binning to a fits file.

Write a 1d-spectrum to a file. Wavelength axis and header keywords are related through the following expression: \( wvl = ((\text{np.arange}(N) + 1.0) - \text{CRPIX1}) \times \text{CDELT1} + \text{CRVAL1} \), where CRPIX1, CDELT1, and CRVAL1 are the relevant header keywords.

The function allows to specify an existing fits extension, from which the header will be cloned. Alternatively, an arbitrary, user-defined header may be given.

**Parameters**

- **fn** [string] Filename
- **flux** [array] Flux array
- **wvl** [array, optional] Wavelength array. Either the wavelength array or the header keywords have to be provided (see waveParams).
- **waveParams** [dict, optional] Wavelength information required in the fits-header. Required keys are CDELT, CRVAL, CRPIX or (CDELT1, CRVAL1, CRPIX1).
- **fluxErr** [array, optional] Flux errors. If given, the error will be stored in an additional extension.
- **header** [dict, optional] Dictionary with header information to be transfered to the new file. Note that the wavelength information will be overwritten by the information given to this routine. If both, a reference file to clone the header from and the header parameters are given, the cloned header from the reference file will be overwritten and extended by the keywords specified here.
- **refFileName** [string, optional] Clone header keywords from a reference file. Note that the wavelength information will be overwritten by the information given to this routine.
- **refFileExt** [int, optional] Reference-file extension to be used for cloning the header keywords. The default is 0.

**Example of usage**

```python
import numpy as np
from PyAstronomy import pyasl

# Generate a "spectrum"
wvl = np.arange(5000., 5010., 0.01)
flux = np.random.normal(1.0, 0.01, wvl.size)

# Write spectrum providing wavelength array
pyasl.write1dFitsSpec("test1.fits", flux, wvl=wvl, clobber=True)

# Write spectrum specifying wavelength-related header keywords manually
wp = {"CRVAL1": 5000., "CDELT1": 0.01, "CRPIX1": 1}
pyasl.write1dFitsSpec("test2.fits", flux, waveParams=wp, clobber=True)
```
Photometric filter transmission curves

Broad-band photometry is an important tool to study variability and band transmission curves are, e.g., required to derive photometric fluxes from spectra. The `TransmissionCurves()` class provides access to photometric transmission curves.

Further information on photometric bands:

<table>
<thead>
<tr>
<th>Band</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bessel</td>
<td>Bessel 1990, PASP 102, 1181B (<a href="#">helpful link</a>)</td>
</tr>
<tr>
<td>Spitzer IRAC</td>
<td>Transmission curves for Spitzer IRAC instrument Call <code>addSpitzerIRACPassbands</code> to add to passband inventory.</td>
</tr>
<tr>
<td>Kepler</td>
<td>Kepler passband (high resolution) Call <code>addKeplerPassband</code> to add passband to inventory.</td>
</tr>
<tr>
<td>TESS</td>
<td>Transmission curve for TESS mission see Call <code>addTESSPassband</code> to add passband to inventory.</td>
</tr>
</tbody>
</table>

Example of usage

```python
from __future__ import print_function
from PyAstronomy import pyasl
import numpy as np
import matplotlib.pyplot as plt
# Get transmission curve object
tcs = pyasl.TransmissionCurves()

# Add passbands from Spitzer IRAC
tcs.addSpitzerIRACPassbands()
print("Available bands: ", tcs.availableBands())

# Wavelength axis
wvl = np.linspace(3000, 10000, 10000)

# Plot transmission curves for Bessel b, v, and r bands
for (b, c) in six.iteritems({"b": "b", "v": "k", "r": "r"):  
    tc = tcs.getTransCurve("Bessel " + b)
    trans = tc(wvl)
    plt.plot(wvl, trans, c+'-', label="Bessel " + b)

# Plot transmission curves for Johnson U, B, and V bands
for (b, c) in six.iteritems({"U": "m", "B": "b", "V": "k"):  
    tc = tcs.getTransCurve("Johnson " + b)
    trans = tc(wvl)
    plt.plot(wvl, trans, c+'--', label="Johnson " + b)

plt.legend()
plt.xlabel("Wavelength [$\AA$]")
plt.ylabel("Transmission")
```

(continues on next page)
plt.show()

# Create Planck spectrum ...
# ... and convolve with Johnson V band

plt.plot(wvl, spec, 'b-', label='Input spectrum')
plt.plot(wvl, vbs, 'r--', label='Convolution with Johnson V band')
plt.legend()
plt.show()
**addPassband** (*name*, *wvl*, *trans*, *snc=False*)

Add a new passband to the inventory.

**Parameters**

- **name** [string] The name of the passband.
- **wvl** [array] Wavelength in Å.
- **trans** [array] Transmission of the passband.
- **snc** [boolean, optional] A *Skip Name Check* flag. If False (default), an exception is raised if the passband name is already present in the inventory of passbands. Otherwise the old passband is replaced by the new specification.

**addSpitzerIRACPassbands** (*forceDownload=False, verbose=True*)

Adds Spitzer IRAC passbands.

On first call, the passband files are downloaded. The files are downloaded from:

http://irsa.ipac.caltech.edu/data/SPITZER/docs/irac/calibrationfiles/spectralresponse/

**Parameters**

- **forceDownload** [boolean, optional] If True, a re-download of the passband files is triggered. Default is False.
- **verbose** [boolean, optional] If True (default), download process will print information on progress.

**addTESSPassband** (*forceDownload=False, verbose=True*)

Adds TESS mission passband.

TESS passband is downloaded from: https://heasarc.gsfc.nasa.gov/docs/tess/data/tess-response-function-v1.0.csv

and added as ‘TESS’.

**Parameters**

- **forceDownload** [boolean, optional] If True, a re-download of the passband files is triggered. Default is False.
- **verbose** [boolean, optional] If True (default), download process will print information on progress.

**availableBands** ()

Band names of available transmission curves.

**Returns**

- **Available names** [list of strings] All bands for which data are available.

**convolveWith** (*wvl*, *spec*, *bn*, *ik='linear'*)

Convolve spectrum with transmission curve.

**Parameters**

- **wvl**, **spec** [arrays] Wavelength axis [Å] and spectral data.
- **bn** [string] Name of the band.
- **ik** [string, optional] The type of interpolation. Accepts all values also accepted by the *kind* keyword of scipy’s *interp1d* routine. Default is ‘linear’.

**Returns**
**Convolved spectrum** [array] Input spectrum multiplied with transmission curve of the specified band.

`getTransCurve (bn, ik='linear')`
Get a transmission curve.

**Parameters**
- **bn** [string] Name of the band.
- **ik** [string, optional] The type of interpolation. Accepts all values also accepted by the `kind` keyword of scipy’s `interp1d` routine. Default is ‘linear’.

**Returns**
- **Transmission curve** [callable] An object (scipy.interpolate.interp1d) that can be called with wavelength (float or array in [Å]) as argument and returns the transmission.

`getTransCurveData (bn)`
Get data specifying the transmission curve.

**Returns**
- **Transmission table** [2d array] A table (array) with wavelength [Å] in first column and transmission (0-1) in the second column.

**Coordinates, time, and observation tools**

**Julian/calendar date and heliocentric correction**

Below you find the documentation of the following routines:

- `daycnv`
- `bprecess`
- `premat`
- `precess`
- `precess_xyz`
- `xyz`
- `helio_jd`
- `jdcnv`
- `get_juldate`
- `juldate`
- `localTime`
- `weekday`

**Example: Convert from JD into calendar date (daycnv)**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Convert JD to calendar date
```
jd = 2440000.0 + 18614/(24.*3600.)
print("year = %4d, month = %2d, day = %2d, hour = %5.3f \\
       % tuple(pyasl.daycnv(jd))")
print()
print("year = %4d, month = %2d, day = %2d, hour = %2d, minute = %2d, seconds = %2d, \\
       microseconds = %6d" \\
       % tuple(pyasl.daycnv(jd, mode='dtlist')))
print()
dt = pyasl.daycnv(jd, mode='dt')
print("Datetime object: ", dt)

Example: daycnv, helio_jd, jdcnv, get_juldate, and juldate

from __future__ import print_function, division
from PyAstronomy import pyasl
import datetime

# Convert JD to calendar date
jd = 2440000.0
print("year = %4d, month = %2d, day = %2d, hour = %5.3f \\
       % tuple(pyasl.daycnv(jd))")
print()

# Convert calendar date to JD
dt = datetime.datetime(1968, 5, 23, 12)
print("Input date: ", dt)
print("Corresponding Julian date: ", pyasl.jdcnv(dt))
print("Corresponding reduced Julian date: ", pyasl.juldate(dt))
print("Corresponds to weekday: ", pyasl.weekday(dt))
print()

# Get current Julian date and convert to heliocentric date
cjd = pyasl.get_juldate()
print("Current Julian date: ", cjd)
print("Current (reduced) heliocentric Julian date (ra=100 deg, dec=37 deg): ", 
      pyasl.helio_jd(cjd-2.4e6, 100.0, 37.0))

Routines

PyAstronomy.pyasl.asl.astroTimeLegacy.bprecess(ra, dec, mu_radec=None, parallax=0.0, 
                 rad_vel=0.0, epoch=2000.0)

Precess positions from J2000.0 (FK5) to B1950.0 (FK4).

Parameters

ra  [float] Right ascension [deg]
dec [float] Declination [deg]
mu_radec [list] List of two float entries, optional Proper motion [arcsec per tropical CENTURY]
parallax [float] The parallax of the target
rad_vel [float] Radial velocity [km/s]
Returns

**Precessed position** [list] [ra_1950, dec_1950, MU_RADEC, PARALLAX, RAD_VEL] referring to 1950

Notes

**Note:** This function was ported from the IDL Astronomy User’s Library.

**IDL - Documentation**

**NAME:** BPRECESS

**PURPOSE:** Precess positions from J2000.0 (FK5) to B1950.0 (FK4)

**EXPLANATION:** Calculates the mean place of a star at B1950.0 on the FK4 system from the mean place at J2000.0 on the FK5 system.

**CALLING SEQUENCE:**

```plaintext
bprecess, ra, dec, ra_1950, dec_1950, [ MU_RADEC = , PARALLAX = RAD_VEL =, EPOCH = ]
```

**INPUTS:**

RA,DEC - Input J2000 right ascension and declination in degrees. Scalar or N element vector

**OUTPUTS:**

RA_1950, DEC_1950 - The corresponding B1950 right ascension and declination in degrees. Same number of elements as RA,DEC but always double precision.

**OPTIONAL INPUT-OUTPUT KEYWORDS**

- **MU_RADEC** - 2xN element double precision vector containing the proper motion in seconds of arc per tropical century in right ascension and declination.
- **PARALLAX** - N_element vector giving stellar parallax (seconds of arc) RAD_VEL - N_element vector giving radial velocity in km/s

The values of MU_RADEC, PARALLAX, and RADVEL will all be modified upon output to contain the values of these quantities in the B1950 system. The parallax and radial velocity will have a very minor influence on the B1950 position.

**EPOCH** - scalar giving epoch of original observations, default 2000.0d This keyword value is only used if the MU_RADEC keyword is not set.


BPRECESS distinguishes between the following two cases: (1) The proper motion is known and non-zero (2) the proper motion is unknown or known to be exactly zero (i.e. extragalactic radio sources). In this case, the reverse of the algorithm in Appendix 2 of Aoki et al. (1983) is used to ensure that the output proper motion is exactly zero. Better precision can be achieved in this case by inputting the EPOCH of the original observations.

The error in using the IDL procedure PRECESS for converting between B1950 and J1950 can be up to 12”, mainly in right ascension. If better accuracy than this is needed then BPRECESS should be used.

An unsystematic comparison of BPRECESS with the IPAC precession routine (http://nedwww.ipac.caltech.edu/forms/calculator.html) always gives differences less than 0.15”.

---

2.1. Python AstroLib

RA(2000) = 13h 42m 12.740s Dec(2000) = 8d 23’ 17.69” Mu(RA) = -.0257 s/yr Mu(Dec) = -.090 ''/yr

IDL> mu_radec = 100D* [-15D*.0257, -0.090] IDL> ra = ten(13, 42, 12.740)*15.D IDL> dec = ten(8, 23, 17.69) IDL> bprecess, ra, dec, ra1950, dec1950, mu_radec = mu_radec IDL> print, adstring(ra1950, dec1950, 2)

===> 13h 39m 44.526s +08d 38’ 28.63”


W. Landsman February 2000

PyAstronomy.pyasl.asl.astroTimeLegacy.daycnv(xjd, mode='idl')

Converts Julian dates to Gregorian calendar dates.

Handles both individual floats as xjd and iterables such as lists and arrays. In the latter case, the result is returned in the form of a list.

Parameters

xjd [float, list, array] The Julian date

mode [string, [idl, dtlist, dt], optional] Determines format of output. If ‘idl’ is given (default), a list holding [year, month, day, (fractional) hours] is returned; this mimics the behavior of the IDL astrolib function. If ‘dtlist’ is given, a list holding [year, month, day, hours, minutes, seconds, microseconds] is returned. Finally, if ‘dt’ is specified, a Python datetime object will be returned. If the input is an iterable, the mode determines the format of the individual items in the result list.

Returns

Calendar date [list or datetime object] A list holding [year, month, day, (fractional) hours] (default) or [year, month, day, hours, minutes, seconds, microseconds]. Alternatively, a Python datetime object is returned. The format depends on the ‘mode’ specified. If the input is an iterable of Julian dates, the output is a list.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: DAYCNV

PURPOSE: Converts Julian dates to Gregorian calendar dates

CALLING SEQUENCE: DAYCNV, XJD, YR, MN, DAY, HR

INPUTS: XJD = Julian date, positive double precision scalar or vector

OUTPUTS: YR = Year (Integer) MN = Month (Integer) DAY = Day (Integer) HR = Hours and fractional hours (Real). If XJD is a vector,
then YR, MN, DAY and HR will be vectors of the same length.

**EXAMPLE:** IDL> DAYCNV, 2440000.D, yr, mn, day, hr yields yr = 1968, mn = 5, day = 23, hr = 12.

**WARNING:** Be sure that the Julian date is specified as double precision to maintain accuracy at the fractional hour level.

**METHOD:** Uses the algorithm of Fliegel and Van Flandern (1968) as reported in the “Explanatory Supplement to the Astronomical Almanac” (1992), p. 604. Works for all Gregorian calendar dates with XJD > 0, i.e., dates after -4713 November 23.

**REVISION HISTORY:** Converted to IDL from Yeoman’s Comet Ephemeris Generator, B. Pfarr, STX, 6/16/88. Converted to IDL V5.0 W. Landsman September 1997

PyAstronomy.pyasl.asl.astroTimeLegacy.get_juldate()

Return the current Julian Date

**Notes**

**Note:** This function was ported from the IDL Astronomy User’s Library.

---

**IDL - Documentation**

**NAME:** GET_JULDATE

**PURPOSE:** Return the current Julian Date

**EXPLANATION:** In V5.4, GET_JULDATE became completely obsolete with the introduction of the /UTC keyword to SYSTIME(). So GET_JULDATE,jd is equivalent to jd = SYSTIME(/JULIAN,/UTC).

**CALLING SEQUENCE:** GET_JULDATE,jd

**INPUTS:** None

**OUTPUTS:** jd = Current Julian Date, double precision scalar

**EXAMPLE:** Return the current hour, day, month and year as integers

IDL> GET_JULDATE, JD ;Get current Julian date IDL> DAYCNV, JD, YR, MON, DAY, HOURS ;Convert to hour, day month & year

**METHOD:** A call is made to SYSTIME(/JULIAN,/UTC).


PyAstronomy.pyasl.asl.astroTimeLegacy.helio_jd(date, ra, dec, B1950=False, TIME_DIFF=False)

Convert geocentric (reduced) Julian date to heliocentric Julian date

**Parameters**

- **date** [float] (Reduced) Julian date (2.4e6 subtracted)
- **ra, dec** [float] Right ascension and declination in degrees
- **B1950** [boolean] If True, input coordinates are assumed to be given in equinox 1950 coordinates.
TIME_DIFF [boolean] If True, this function returns the time difference (heliocentric JD - geocentric JD) in seconds

Returns

HJD [float] The heliocentric Julian date.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: HELIO_JD

PURPOSE: Convert geocentric (reduced) Julian date to heliocentric Julian date

EXPLANATION: This procedure correct for the extra light travel time between the Earth and the Sun.

An online calculator for this quantity is available at http://www.physics.sfasu.edu/astro/javascript/hjd.html

CALLING SEQUENCE: jdhelio = HELIO_JD( date, ra, dec, /B1950, /TIME_DIFF)

INPUTS

date - reduced Julian date (= JD - 2400000), scalar or vector, MUST be double precision

ra,dec - scalars giving right ascension and declination in DEGREES Equinox is J2000 unless the /B1950 keyword is set

OUTPUTS:

jdhelio - heliocentric reduced Julian date. If /TIME_DIFF is set, then HELIO_JD() instead returns the time difference in seconds between the geocentric and heliocentric Julian date.

OPTIONAL INPUT KEYWORDS

/B1950 - if set, then input coordinates are assumed to be in equinox B1950 coordinates.

/TIME_DIFF - if set, then HELIO_JD() returns the time difference (heliocentric JD - geocentric JD) in seconds

EXAMPLE: What is the heliocentric Julian date of an observation of V402 Cygni (J2000: RA = 20 9 7.8, Dec = 37 09 07) taken June 15, 1973 at 11:40 UT?

IDL> juldate, [1973,6,15,11,40], jd ;Get geocentric Julian date

IDL> hjd = helio_jd( jd, ten(20,9,7.8)*15., ten(37,9,7) )

===> hjd = 41848.9881

Wayne Warren (Raytheon ITSS) has compared the results of HELIO_JD with the FORTRAN subroutines in the STARLINK SLALIB library (see http://star-www.rl.ac.uk/).

Time Diff (sec)

1999-10-29T00:00:00.0 21 08 25. -67 22 00. -59.0 -59.0 1999-10-29T00:00:00.0 02 56 33.4 +00 26 55. 474.1 474.1 1940-12-11T06:55:00.0 07 34 41.9 -00 30 42. 366.3 370.2 1992-02-29T03:15:56.2 12 56 27.4 +42 10 17.
PyAstronomy Documentation, Release 0.15.2

350.8 350.9 2000-03-01T10:26:31.8 14 28 36.7 -20 42 11. 243.7 243.7 2100-02-26T09:18:24.2 08 26 51.7 +85 47 28. 104.0 108.8 PROCEDURES CALLED:

\texttt{bprecess, xyz, zparcheck}


\texttt{PyAstronomy.pyasl.asl.astroTimeLegacy.jdcnv}\(\texttt{(dt)}\)

Converts Gregorian dates to Julian days

**Parameters**

\texttt{dt [DateTime object]} The date. This is interpreted as UTC and the timezone component is not considered.

**Returns**

\texttt{Julian day [float]}

**Notes**

\texttt{Note: This function was ported from the IDL Astronomy User’s Library.}

**IDL - Documentation**

**NAME:** JDCNV

**PURPOSE:** Converts Gregorian dates to Julian days

**EXPLANATION:** For IDL versions V5.1 or greater, this procedure is superceded by JUL\_DAY() function in the standard IDL distribution. Note, however, that prior to V5.1 there was a bug in JUL\_DAY() that gave answers off by 0.5 days.

**CALLING SEQUENCE:** JDCNV, YR, MN, DAY, HR, JULIAN

**INPUTS:** YR = Year, integer scalar or vector MN = Month integer (1-12) scalar or vector DAY = Day integer 1-31) scalar or vector HR = Hours and fractions of hours of universal time (U.T.), scalar or vector

**OUTPUTS:** JULIAN = Julian date (double precision)

**EXAMPLE:** To find the Julian Date at 1978 January 1, 0h (U.T.)

\texttt{IDL> JDCNV, 1978, 1, 1, 0., JULIAN}

will give \texttt{JULIAN = 2443509.5}

**NOTES:**

(1) JDCNV will accept vector arguments

(2) JULDAT\_E is an alternate procedure to perform the same function

**REVISION HISTORY:** Converted to IDL from Don Yeomans Comet Ephemeris Generator, B. Pfarr, STX, 6/15/88 Converted to IDL V5.0 W. Landsman September 1997 Added checks on valid month, day ranges W. Landsman July 2008

2.1. Python AstroLib
PyAstronomy Documentation, Release 0.15.2

PyAstronomy.pyasl.asl.astroTimeLegacy.juldate(date)

Convert from calendar to Reduced Julian Date

This function returns the reduced Julian date, which is obtained by subtracting 2400000 from the Julian date. To convert the output into Modified Julian Date (MJD), another 0.5 days have to be subtracted.

Parameters

date [DateTime object] Calendar date

Returns

RJD [float] The reduced Julian date.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: JULDATE

PURPOSE: Convert from calendar to Reduced Julian Date

EXPLANATION: Julian Day Number is a count of days elapsed since Greenwich mean noon on 1 January 4713 B.C. The Julian Date is the Julian day number followed by the fraction of the day elapsed since the preceding noon.

This procedure duplicates the functionality of the JULDAY() function in the standard IDL distribution, but also allows interactive input and gives output as Reduced Julian date (=JD - 2400000.) (Also note that prior to V5.1 there was a bug in JULDAY() that gave answers offset by 0.5 days.)

CALLING SEQUENCE:

JULDATE, /PROMPT ;Prompt for calendar Date, print Julian Date or
JULDATE, date, jd

INPUT:

DATE - 3 to 6-element vector containing year,month (1-12),day, and optionally hour, minute, and second all specified as numbers (Universal Time). Year should be supplied with all digits. Years B.C should be entered as negative numbers (and note that Year 0 did not exist). If Hour, minute or seconds are not supplied, they will default to 0.

OUTPUT:

JD - Reduced Julian date, double precision scalar. To convert to Julian Date, add 2400000. JULDATE will print the value of JD at the terminal if less than 2 parameters are supplied, or if the /PROMPT keyword is set

OPTIONAL INPUT KEYWORD:

/PROMPT - If this keyword is set and non-zero, then JULDATE will prompt for the calendar date at the terminal.

RESTRICTIONS: The procedure HELIO_JD can be used after JULDATE, if a heliocentric Julian date is required.
**EXAMPLE:** A date of 25-DEC-2006 06:25 UT may be expressed as either

```
IDL> jdate, [2006, 12, 25, 6, 25], jd
IDL> jdate, [2006, 12, 25.2673611d], jd
```

In either case, one should obtain a Reduced Julian date of JD = 54094.7673611

**PROCEDURE USED:** GETOPT()

**REVISION HISTORY** Adapted from IUE RDAF (S. Parsons) 8-31-87 Algorithm from Sky and Telescope April 1981 Added /PROMPT keyword, W. Landsman September 1992 Converted to IDL V5.0 W. Landsman September 1997 Make negative years correspond to B.C. (no year 0), work for year 1582 Disallow 2 digit years. W. Landsman March 2000

PyAstronomy.pyasl.asl.astroTimeLegacy.precess(ra, dec, equinox1, equinox2, FK4=False, radian=False)

Precess coordinates from EQUINOX1 to EQUINOX2.

**Parameters**
- `ra, dec, equinox1, equinox2` [float] Position and equinox
- `FK4` [boolean] Set to True to obtain output in FK4 system.
- `radian` [boolean] If True, `ra` and `dec` must be given in radian (degrees otherwise).

**Returns**
- `Position` [list of ra and dec] A list with [ra, dec] precessed from equinox 1 to equinox 2.

**Notes**

*Note:* This function was ported from the IDL Astronomy User’s Library.

**IDL - Documentation**

**NAME:** PRECESS

**PURPOSE:** Precess coordinates from EQUINOX1 to EQUINOX2.

**EXPLANATION:** For interactive display, one can use the procedure ASTRO which calls PRECESS or use the /PRINT keyword. The default (RA,DEC) system is FK5 based on epoch J2000.0 but FK4 based on B1950.0 is available via the /FK4 keyword.

Use BPRECESS and JPRECESS to convert between FK4 and FK5 systems

**CALLING SEQUENCE:** PRECESS, ra, dec, [ equinox1, equinox2, /PRINT, /FK4, /RADIAN ]

**INPUT - OUTPUT:**
- **RA** - Input right ascension (scalar or vector) in DEGREES, unless the /RADIAN keyword is set
- **DEC** - Input declination in DEGREES (scalar or vector), unless the /RADIAN keyword is set

The input RA and DEC are modified by PRECESS to give the values after precession.

**OPTIONAL INPUTS:**
- **EQUINOX1** - Original equinox of coordinates, numeric scalar. If omitted, then PRECESS will query for EQUINOX1 and EQUINOX2.
- **EQUINOX2** - Equinox of precessed coordinates.
OPTIONAL INPUT KEYWORDS:

/PRINT - If this keyword is set and non-zero, then the precessed coordinates are displayed at the terminal. Cannot be used with the /RADIAN keyword

/FK4
  • If this keyword is set and non-zero, the FK4 (B1950.0) system will be used otherwise FK5 (J2000.0) will be used instead.

/RADIAN - If this keyword is set and non-zero, then the input and output RA and DEC vectors are in radians rather than degrees

RESTRICTIONS: Accuracy of precession decreases for declination values near 90 degrees. PRECESS should not be used more than 2.5 centuries from 2000 on the FK5 system (1950.0 on the FK4 system).

EXAMPLES:

(1) The Pole Star has J2000.0 coordinates (2h, 31m, 46.3s, 89d 15’ 50.6”); compute its coordinates at J1985.0

IDL> precess, ten(2,31,46.3)*15, ten(89,15,50.6), 2000, 1985, /PRINT
    ===> 2h 16m 22.73s, 89d 11’ 47.3”

(2) Precess the B1950 coordinates of Eps Ind (RA = 21h 59m,33.053s, DEC = (-56d, 59’, 33.053”) to equinox B1975.

IDL> ra = ten(21, 59, 33.053)*15 IDL> dec = ten(-56, 59, 33.053) IDL> precess, ra, dec ,1950, 1975, /fk4


PROCEDURE CALLED: Function PREMAT - computes precession matrix


Precess equatorial geocentric rectangular coordinates.

Parameters

x, y, z, equinox1, equinox2 [float]

Returns

Precessed coordinates [list] A list containing the updated x, y, and z values.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.
NAME: PRECESS_XYZ

PURPOSE: Precess equatorial geocentric rectangular coordinates.

CALLING SEQUENCE: precess_xyz, x, y, z, equinox1, equinox2

INPUT/OUTPUT:

x,y,z: scalars or vectors giving heliocentric rectangular coordinates THESE ARE CHANGED UPON RETURNING.

INPUT: EQUINOX1: equinox of input coordinates, numeric scalar EQUINOX2: equinox of output coordinates, numeric scalar

OUTPUT: x,y,z are changed upon return

NOTES:

The equatorial geocentric rectangular coords are converted to RA and Dec, precessed in the normal way, then changed back to x, y and z using unit vectors.

EXAMPLE: Precess 1950 equinox coords x, y and z to 2000. IDL> precess_xyz,x,y,z, 1950, 2000

HISTORY:

Written by P. Plait/ACC March 24 1999 (unit vectors provided by D. Lindler)
Use /Radian call to PRECESS W. Landsman November 2000 Use two parameter call to ATAN W. Landsman June 2001

PyAstronomy.pyasl.asl.astroTimeLegacy. premat (equinox1, equinox2, FK4=False)
Return the precession matrix needed to go from EQUINOX1 to EQUINOX2.

Parameters

equinox1, equinox2 [float] EQUINOX1 and EQUINOX2

FK4 [boolean] Set this to True to obtain output in FK4 system

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: PREMAT

PURPOSE: Return the precession matrix needed to go from EQUINOX1 to EQUINOX2.

EXPLANATION: This matrix is used by the procedures PRECESS and BARYVEL to precess astronomical coordinates

CALLING SEQUENCE: matrix = PREMAT( equinox1, equinox2, [ /FK4 ] )

INPUTS: EQUINOX1 - Original equinox of coordinates, numeric scalar. EQUINOX2 - Equinox of precessed coordinates.

OUTPUT:

matrix - double precision 3 x 3 precession matrix, used to precess equatorial rectangular coordinates

OPTIONAL INPUT KEYWORDS:
**/FK4**

- If this keyword is set, the FK4 (B1950.0) system precession angles are used to compute the precession matrix. The default is to use FK5 (J2000.0) precession angles.

**EXAMPLES:**
Return the precession matrix from 1950.0 to 1975.0 in the FK4 system.

IDL> matrix = PREMAT( 1950.0, 1975.0, /FK4)


**REVISION HISTORY**
Written, Wayne Landsman, HSTX Corporation, June 1994 Converted to IDL V5.0 W. Landsman September 1997

---

**weekday**

`weekday(date)`

Return weekday by name

**Parameters**
- `date` [DATETIME object] Give the date of a day as DATETIME object.

**Returns**
- `name` string “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday”, “Sunday”

---

**xyz**

`xyz(date, velocity=False, equinox=1950.0)`

Calculate geocentric X,Y, and Z and velocity coordinates of the Sun.

**Parameters**
- `date` [float] Julian date
- `equinox` [float] Equinox of output. If None, Equinox will be 1950.
- `velocity` [boolean] If False, the velocity of the Sun will not be calculated

**Returns**
- **Sun position and velocity** [list] A list of the from [X, Y, Z, XVEL, YVEL, ZVEL]. Last three values are None if `velocity` flag is set to False.

---

**Notes**

- This function was ported from the IDL Astronomy User’s Library.

---

**IDL - Documentation**

**NAME:** XYZ

**PURPOSE:** Calculate geocentric X,Y, and Z and velocity coordinates of the Sun

**EXPLANATION:** Calculates geocentric X,Y, and Z vectors and velocity coordinates (dx, dy and dz) of the Sun. (The positive X axis is directed towards the equinox, the y-axis, towards the point on the equator at right ascension 6h, and the z axis toward the north pole of the equator). Typical position accuracy is <1e-4 AU (15000 km).

**CALLING SEQUENCE:** XYZ, date, x, y, z, [ xvel, yvel, zvel, EQUINOX = ]

**INPUT:** date: reduced julian date (=JD - 2400000), scalar or vector
OUTPUT:

\textbf{x,y,z: scalars or vectors giving heliocentric rectangular coordinates} (in A.U) for each date supplied.

Note that \(\sqrt{x^2 + y^2 + z^2}\) gives the Earth-Sun distance for the given date.

\textbf{xvel, yvel, zvel: velocity vectors corresponding to X, Y and Z.}

\textbf{OPTIONAL KEYWORD INPUT:} EQUINOX: equinox of output. Default is 1950.

\textbf{EXAMPLE:} What were the rectangular coordinates and velocities of the Sun on Jan 22, 1999 0h UT (= JD 2451200.5) in J2000 coords? NOTE: Astronomical Almanac (AA) is in TDT, so add 64 seconds to UT to convert.

IDL> xyz,51200.5+64.d/86400.d,x,y,z,xv,yv,zv,equinox = 2000

\textbf{Compare to Astronomical Almanac (1999 page C20)} X (AU) Y (AU) Z (AU)

XYZ: 0.51456871 -0.76963263 -0.33376880 AA: 0.51453130 -0.7697110 -0.3337152 abs(err): 0.00003739 0.00007839 0.00005360 abs(err)

(km): 5609 11759 8040

\textbf{NOTE: Velocities in AA are for Earth/Moon barycenter}

(a very minor offset) see AA 1999 page E3 X VEL (AU/DAY) YVEL (AU/DAY) Z VEL (AU/DAY)

XYZ: -0.014947268 -0.0083148382 -0.0036068577 AA: -0.01494574 -0.00831185 -0.00360365 abs(err): 0.000001583 0.0000029886 0.0000032077 abs(err) (km/sec): 0.00265 0.00519 0.00557

\textbf{PROCEDURE CALLS:} PRECESS_XYZ

\textbf{REVISION HISTORY} Original algorithm from Almanac for Computers, Doggett et al. USNO 1978 Adapted from the book Astronomical Photometry by A. Henden Written W. Landsman STX June 1989 Correct error in X coefficient W. Landsman HSTX January 1995 Added velocities, more terms to positions and EQUINOX keyword,

some minor adjustments to calculations P. Plait/ACC March 24, 1999

\textbf{PyAstronomy.pyasl.asl.\texttt{LocalTime}(utc, lon, diff=True)}

Computes the Local Time for a given UTC at a given geographic longitude.

The local time is computed as UTC + LONGITUDE/15.

\textbf{Parameters}

- \texttt{utc} [float or array] The time in UTC in hours.
- \texttt{lon} [float or array] The geographic (East) longitude in DEGREES for which local time should be calculated.
- \texttt{diff} [boolean, optional] If True (default), returns the difference in HOURS between UTC and local time.

\textbf{Returns}

- \texttt{Time} [float or array] Local time in HOURS (0 - 24) for given geographic longitude and UTC.
- \texttt{Time difference} [float or array] The difference between local and UTC time in hours (only returned if \texttt{diff} is True)
Decimal representation of year

Convert between decimal year representation and gregorian date.

Note: Due to the nature of the gregorian calendar, the decimal year does not represent a uniform measure of time, because some years are longer than others.

PyAstronomy.pyasl.\texttt{decimalYear}(date)
Calculates the decimal representation of a date, e.g., 2013.12.

The code uses Python’s datetime package to determine the fractional year. Thus, leap years are taken into account. There may still be issues with time zones or daylight saving times etc..

Code from: http://stackoverflow.com/questions/6451655/python-how-to-convert-datetime-dates-to-decimal-years

Parameters
- \texttt{date} [python date instance] The input date (and time).

Returns
- Decimal year [float] Decimal representation of the date.

PyAstronomy.pyasl.\texttt{decimalYearGregorianDate}(date, form='datetime')
Convert decimal year into gregorian date.

Formally, the precision of the result is one microsecond.

Parameters
- \texttt{date} [float] The input date (and time).
- \texttt{form} [str, optional] Output format for the date. Either one of the following strings defining a format: “dd-mm-yyyy [hh:mm:ss]”, “yyyy-mm-dd [hh:mm:ss]”, where the term in braces is optional, or “tuple” or “datetime”. If ‘tuple’ is specified, the result will be a tuple holding (year, month, day, hour, minute, second, microseconds). In the case of “datetime” (default), the result will be a datetime object.

Returns
- Gregorian date [str, tuple, or datetime instance] The gregorian representation of the input in the specified format. In case of an invalid format, None is returned.

Example:

```python
from __future__ import print_function, division
import datetime as dt
from PyAstronomy import pyasl

# Convert July 2nd, 1998, 12:30:59 into decimal representation
d = dt.datetime(1998, 7, 2, 12, 30, 59)

decy = pyasl.decimalYear(d)
print("Decimal representation: ", decy)

# Convert back into gregorian date first
```

(continues on next page)
print("The decimal year %10.5f correspond to " % decy +
    pyasl.decimalYearGregorianDate(decy, "yyy-mm-dd hh:mm:ss"))
print(" ... or equivalently (y, m, d, h, m, s, ms): ",
    pyasl.decimalYearGregorianDate(decy, "tuple"))

Aitoff projection

PyAstronomy.pyasl.**aitoff**(l, b)

Carry out Aitoff projection.

**Parameters**

- l, b [float or array] The longitude and latitude [deg]

**Returns**

- x, y [float or array] Aitoff-projected coordinates (x, y)

**Notes**

**Note:** This function was ported from the IDL Astronomy User’s Library.

**IDL - Documentation**

pro aitoff,l,b,x,y +

**NAME:** AITO

**PURPOSE:** Convert longitude, latitude to X,Y using an AITOFF projection.

**EXPLANATION:** This procedure can be used to create an all-sky map in Galactic coordinates with an equal-area Aitoff projection. Output map coordinates are zero longitude centered.

**CALLING SEQUENCE:** AITO, L, B, X, Y

**INPUTS:**
- L - longitude - scalar or vector, in degrees
- B - latitude - same number of elements as L, in degrees

**OUTPUTS:**
- X - X coordinate, same number of elements as L. X is normalized to be between -180 and 180
- Y - Y coordinate, same number of elements as L. Y is normalized to be between -90 and 90.

**NOTES:** See AIPS memo No. 46, page 4, for details of the algorithm. This version of AITO
assumes the projection is centered at b=0 degrees.

**REVISION HISTORY:** Written W.B. Landsman STX December 1989 Modified for Unix:
- J. Bloch LANL SST-9 5/16/91 1.1
- Converted to IDL V5.0 W. Landsman September 1997
PyAstronomy Documentation, Release 0.15.2

PyAstronomy.pyasl.\texttt{inverseAitoff}(x,y)

Carry out an inverse Aitoff projection.

This function reverts to aitoff projection made by the function \texttt{aitoff}. The result is either two floats or arrays (depending on whether float or array was used as input) representing longitude and latitude. Both are given in degrees with \(-180 < \text{longitude} < +180\) and \(-90 < \text{latitude} < 90\).

\textbf{Parameters}

- \texttt{x} [float or array] A value between \(-180.\) and \(+180.\) (see convention in \texttt{aitoff} function).
- \texttt{y} [float or array] A value between \(-90.\) and \(+90.\) (see convention in \texttt{aitoff} function).

\textbf{Returns}

- \texttt{Deprojected coordinates} [float or array] If arrays are used for input, the function returns an array for the longitude, for the latitude, and an index array containing those array indices for which the reprojection could be carried out.

\textbf{Example: Aitoff projection and its inverse}

Carry out Aitoff projection and its inverse.

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Define longitude and latitude in degrees
l = 130.
b = -35.
print("Input - Longitude: \%4d deg, Latitude: \%4d deg" % (l, b))

print("Aitoff project them and ...")
x, y = pyasl.aitoff(l, b)
print(x, y)

print("... get them back.")
l2, b2 = pyasl.inverseAitoff(x, y)
print(l2, b2)
```

\textbf{Position of the Sun}

PyAstronomy.pyasl.\texttt{sunpos}(jd, end_jd=None, jd_steps=None, outfile=None, radian=False, plot=False, full_output=False)

Compute right ascension and declination of the Sun at a given time.

\textbf{Parameters}

- \texttt{jd} [float] The Julian date
- \texttt{end_jd} [float, optional] The end of the time period as Julian date. If given, \texttt{sunpos} computes RA and DEC at \texttt{jd_steps} time points between \texttt{jd} and ending at \texttt{end_jd}.
- \texttt{jd_steps} [integer, optional] The number of steps between \texttt{jd} and \texttt{end_jd} for which RA and DEC are to be calculated.
- \texttt{outfile} [string, optional] If given, the output will be written to a file named according to \texttt{outfile}.
radian [boolean, optional] Results are returned in radian instead of in degrees. Default is False.

plot [boolean, optional] If True, the result is plotted.

full_output: boolean, optional If True, sunpos, additionally, returns the elongation and obliquity of the Sun.

Returns

Time [array] The JDs for which calculations where carried out.

Ra [array] Right ascension of the Sun.


Elongation [array, optional] Elongation of the Sun (only of full_output is set to True).

Obliquity [array, optional] Obliquity of the Sun (only of full_output is set to True).

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: SUNPOS

PURPOSE: To compute the RA and Dec of the Sun at a given date.

CALLING SEQUENCE: SUNPOS, jd, ra, dec, [elong, obliquity, /RADIAN ]

INPUTS:

jd - The Julian date of the day (and time), scalar or vector usually double precision

OUTPUTS:

ra - The right ascension of the sun at that date in DEGREES double precision, same number of elements as jd

dec - The declination of the sun at that date in DEGREES

OPTIONAL OUTPUTS: elong - Ecliptic longitude of the sun at that date in DEGREES, obliquity - the obliquity of the ecliptic, in DEGREES

OPTIONAL INPUT KEYWORD:

/RADIAN - If this keyword is set and non-zero, then all output variables are given in Radians rather than Degrees

NOTES: Patrick Wallace (Rutherford Appleton Laboratory, UK) has tested the accuracy of a C adaptation of the sunpos.pro code and found the following results. From 1900-2100 SUNPOS gave 7.3 arcsec maximum error, 2.6 arcsec RMS. Over the shorter interval 1950-2050 the figures were 6.4 arcsec max, 2.2 arcsec RMS.

The returned RA and Dec are in the given date’s equinox.

Procedure was extensively revised in May 1996, and the new calling sequence is incompatible with the old one.
**METHOD:** Uses a truncated version of Newcomb's Sun. Adapted from the IDL routine SUN_POS by CD Pike, which was adapted from a FORTRAN routine by B. Emerson (RGO).

**EXAMPLE:**

(1) Find the apparent RA and Dec of the Sun on May 1, 1982

```idl
IDL> jdcnv, 1982, 5, 1.0, jd ;Find Julian date jd = 2445090.5
IDL> sunpos, jd, ra, dec
IDL> print,adstring(ra,dec,2)
```

```
02 31 32.61 +14 54 34.9
```

The Astronomical Almanac gives 02 31 32.58 +14 54 34.9 so the error in SUNPOS for this case is < 0.5".

(2) Find the apparent RA and Dec of the Sun for every day in 1997

```idl
IDL> jdcnv, 1997, 1, 1, 0, jd ;Julian date on Jan 1, 1997
IDL> sunpos, jd+dindgen(365), ra, dec ;RA and Dec for each day
```


**Example**

```python
from __future__ import print_function, division
import numpy as np
from PyAstronomy import pyasl
import datetime

# Convert calendar date into JD
# use the datetime package
jd = datetime.datetime(2013, 4, 16)
jd = pyasl.jdcnv(jd)
print("JD = " + str(jd))
pos = pyasl.sunpos(jd, full_output=True)
print("Coordinates of the Sun (ra, dec): $g, $g" % (pos[1], pos[2]))
print("Solar elongation = $g and obliquity = $g" % (pos[3], pos[4]))

# Get the Sun's RA and DEC values for a period of time.
startjd = datetime.datetime(2013, 4, 16)
endjd = datetime.datetime(2013, 6, 16)
# Convert into Julian dates
startjd = pyasl.jdcnv(startjd)
endjd = pyasl.jdcnv(endjd)
print()
pos = pyasl.sunpos(startjd, end_jd=endjd, jd_steps=10, plot=False, full_output=True)

for i in range(len(pos[0])):
    print("At JD = $g: ra = $g, dec = $g" % (pos[0][i], pos[1][i], pos[2][i]))
```

118 Chapter 2. PyAstronomy's sub-packages
Nutation

PyAstronomy.pyasl.nutate (jd, radian=False, plot=False)

Computes the Earth’s nutation in longitude and obliquity for a given (array) of Julian date.

**Warning:** The output of the IDL routine is in units of arcseconds, whereas the default if this routine returns degrees.

**Parameters**
- **jd** [float] The Julian date
- **radian** [boolean, optional] Results are returned in radian instead of in degrees. The default is False.
- **plot** [boolean, optional] Results are plotted. The default is False.

**Returns**
- **Longitude** [float] The nutation in longitude (in deg by default).
- **Obliquity** [float] The nutation in latitude (in deg by default).

**Notes**

**Note:** This function was ported from the IDL Astronomy User’s Library.

**IDL - Documentation**

**NAME:** NUTATE

**PURPOSE:** Return the nutation in longitude and obliquity for a given Julian date

**CALLING SEQUENCE:** NUTATE, jd, Nut_long, Nut_obliq

**INPUT:** jd - Julian ephemeris date, scalar or vector, double precision

**OUTPUT:** Nut_long - the nutation in longitude, same # of elements as jd Nut_obliq - nutation in latitude, same # of elements as jd

**EXAMPLE:**

1. **Find the nutation in longitude and obliquity 1987 on Apr 10 at Oh.** This is example 22.a from Meeus

   IDL> jdcnv,1987,4,10,0,jul IDL> nutate, jul, nut_long, nut_obliq
   ==> nut_long = -3.788 nut_obliq = 9.443

2. **Plot the large-scale variation of the nutation in longitude** during the 20th century

   IDL> yr = 1900 + indgen(100) ;Compute once a year IDL> jdcnv,yr,1,1,0,jul ;Find Julian date of first day of year IDL> nutate,jul, nut_long ;Nutation in longitude IDL> plot, yr, nut_long

   This plot will reveal the dominant (18.6 year) period, but a finer grid is needed to display the shorter periods in the nutation.
METHOD: Uses the formula in Chapter 22 of “Astronomical Algorithms” by Jean Meeus (1998, 2nd ed.) which is based on the 1980 IAU Theory of Nutation and includes all terms larger than 0.0003”.

PROCEDURES CALLED: POLY() (from IDL User’s Library) CIRRANGE, ISARRAY() (from IDL Astronomy Library)


Example: nutate

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import datetime
import numpy as np

# Convert calendar date into JD
# use the datetime package
jd = datetime.datetime(2013, 4, 16)
jd = pyasl.jdcnv(jd)
print("Nutation for the date.")
res = pyasl.nutate(jd)
print("JD = " + str(jd) + ", Longitude = " + str(res[0]) +
"," Obliquity = " + str(res[1]))

# Get nutation for an array of JDs.
startjd = datetime.datetime(2013, 4, 16)
endjd = datetime.datetime(2013, 6, 16)
startjd = pyasl.jdcnv(startjd)
endjd = pyasl.jdcnv(endjd)
jds = np.arange(startjd, endjd, .5)
print()
print("Plot the results")
res = pyasl.nutate(jds, plot=True)
print("Longitude: ", res[0])
print("Obliquity: ", res[1])
```

PyAstronomy.pyasl.co_nutate (jd, ra, dec, radian=False, plot=False, full_output=False)
Compute the changes in RA and DEC due to the Earth’s nutation.

Parameters

- **jd** [float or array] The Julian date. If given as array, its size must match that of ra and dec.
- **ra** [float or array] The right ascension in degrees. If array, it must be same size as dec.
- **dec** [float or array] The declination in degrees. If array, it must be same size as ra.
- **radian** [boolean, optional] Results are returned in radian instead of in degrees. The default is False.
- **plot** [boolean, optional] If True, the results are plotted. For single value jd, the change in ra and dec is plotted versus ra and dec. For an array of JDs, ra and dec is plotted versus JD. The default is False
full_output [boolean, optional] If True, the result will also contain the obliquity of the ecliptic, the nutation in the longitude and the nutation in the obliquity of the ecliptic. The default is False.

Returns

dRa [float or array] The change in right ascension [by default in deg].

dDec [float or array] The change in declination [by default in deg].

True obliquity [float, optional] The true obliquity of the ecliptic [by default in deg]. Only if full_output is True.

dLong [float or array, optional] The nutation in longitude [by default in deg]. Only if full_output is True.

dObliquity [float or array, optional] The nutation in the obliquity of the ecliptic [by default in deg]. Only if full_output is True.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

---

IDL - Documentation

NAME: CO_NUTATE

PURPOSE: Calculate changes in RA and Dec due to nutation of the Earth’s rotation

EXPLANATION: Calculates necessary changes to ra and dec due to the nutation of the Earth’s rotation axis, as described in Meeus, Chap 23. Uses formulae from Astronomical Almanac, 1984, and does the calculations in equatorial rectangular coordinates to avoid singularities at the celestial poles.

CALLING SEQUENCE: CO_NUTATE, jd, ra, dec, d_ra, d_dec, [EPS=, D_PSI =, D_EPS = ]

INPUTS

JD: Julian Date [scalar or vector] RA, DEC : Arrays (or scalars) of the ra and dec’s of interest

Note: if jd is a vector, ra and dec MUST be vectors of the same length.

OUTPUTS:

_d_ra, d_dec: the corrections to ra and dec due to nutation (must then be added to ra and dec to get corrected values).

OPTIONAL OUTPUT KEYWORDS:

EPS: set this to a named variable that will contain the obliquity of the ecliptic.

D_PSI: set this to a named variable that will contain the nutation in the longitude of the ecliptic

D_EPS: set this to a named variable that will contain the nutation in the obliquity of the ecliptic

EXAMPLE:

(1) Example 23a in Meeus: On 2028 Nov 13.19 TD the mean position of Theta Persei is 2h 46m 11.331s 49d 20’ 54.54”. Determine the shift in position due to the Earth’s nutation.

IDL> jd = JULDAY(11,13,2028,.19*24) ;Get Julian date IDL> CO_NUTATE, jd,ten(2,46,11.331)*15.,ten(49,20,54.54),d_ra,d_dec
Example: co_nutate

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import datetime
import numpy as np

# Convert calendar date into JD
# use the datetime package
jd = datetime.datetime(2013, 4, 16)
jd = pyasl.jdcnv(jd)
# Specify RA and DEC (degrees)
ra = 10.
dec = 30.
print("Get change in RA and DEC due to Earth's nutation for JD = "+str(jd))
print(pyasl.co_nutate(jd, ra, dec))

print()
print("Get change for several RAs and DECs for the same JD")
ra = np.arange(0., 160., 20.)
dec = np.arange(-80., 80., 20.)
res = pyasl.co_nutate(np.repeat(jd, ra.size), ra, dec)
print(res[0], res[1])

print()
print("Get change for several RAs and DECs for different JDs")
jds = np.arange(jd, jd+ra.size, 1)
res = pyasl.co_nutate(jds, ra, dec)
print("JD delta(RA) delta(DEC")
for i in range(len(ra)):
    print("%12.5f %8.5f %8.5f" % (jds[i], res[0][i], res[1][i]))
```

### Aberration

**PyAstronomy.pyasl.co_aberration** *(jd, ra, dec, radian=False)*

Computes the changes in RA and DEC due to annual aberration.

**Parameters**

- **jd** [float or array] The Julian date(s). If array, must be the same size as `ra` and `dec`.
- **ra** [float or array] The right ascension in degrees. If array, it must be the same size as `dec`.
- **dec** [float or array] The declination in degrees. If array, it must be the same size as `ra`.
- **radian** [boolean, optional] Results are returned in radian instead of degrees. The default is False.

**Returns**

- **dRA** [float or array] The change in right ascension [by default in deg].
dDec  [float or arrays] The change in declination [by default in deg].

Notes

Note: This function was ported from the IDL Astronomy User's Library.

IDL - Documentation

NAME: CO_ABERRATION

PURPOSE: Calculate changes to Ra and Dec due to the effect of annual aberration

EXPLANATION: as described in Meeus, Chap 23.

CALLING SEQUENCE: co_aberration, jd, ra, dec, d_ra, d_dec, [EPS = ]

INPUTS

jd : Julian Date [scalar or vector] ra, dec : Arrays (or scalars) of the ra and dec's in degrees

Note: if jd is a vector, then ra and dec must either be scalars, or vectors of the same length.

OUTPUTS

d_ra, d_dec: the corrections to ra and dec due to aberration in arcseconds. (These values can be added to the true RA and dec to get the apparent position). Note that d_ra is not multiplied by cos(dec), so that apparent_ra = ra + d_ra/3600.

OPTIONAL INPUT KEYWORD:

eps [set this to the true obliquity of the ecliptic (in radians), or]

it will be set for you if you don't know it (in that case, set it to an empty variable).

EXAMPLE: Compute the change in RA and Dec of Theta Persei (RA = 2h46m,11.331s, Dec = 49d20',54.54") due to aberration on 2028 Nov 13.19 TD

IDL> jdcnv,2028,11,13,.19*24,jd ;Get Julian date IDL>
co_aberration,jd,ten(2,46,11.331)*15,ten(49,20,54.54),d_ra,d_dec

===> d_ra = 30.045" (=2.003s) d_dec = 6.697"

NOTES:

These formula are from Meeus, Chapters 23. Accuracy is much better than 1 arcsecond.

The maximum deviation due to annual aberration is 20.49" and occurs when the Earth velocity is perpendicular to the direction of the star.

REVISION HISTORY: Written, June 2002, Chris O’Dell, U. of Wisconsin Fix error with vector input W. Landsman June 2009 June 2009 update fixed case where JD was scalar but RA,Dec were vectors, but broke the case when both JD and RA,Dec were vectors Aug 2012 W. Landsman

Example:
from __future__ import print_function, division
from PyAstronomy import pyasl
import datetime
import numpy as np

# Convert calendar date to JD
# use the datetime package
jd = datetime.datetime(2013, 4, 16)
jd = pyasl.jdcnv(jd)

# Specify RA and DEC
ra = 10.
dec = 30.

print("Get change in RA and DEC due to annual aberration")
print(" for JD = " + str(jd) + ":",
    np.ravel(pyasl.co_aberration(jd, ra, dec)))

print()

print("Get change for several RAs and DECs for the same JD")
ra = np.arange(10., 50., 10.)
dec = np.arange(30., 70., 10.)
res = pyasl.co_aberration(np.repeat(jd, ra.size), ra, dec)
print(res[0], res[1])

print()

print("Get change for several RAs and DECs for different JDs")
jds = np.arange(jd, jd+ra.size, 1)
res = pyasl.co_aberration(jds, ra, dec)
print("JD    delta(RA)    delta(DEC)")
for i in range(ra.size):
    print("%12.5f %8.5f %8.5f" % (jds[i], res[0][i], res[1][i]))

---

**Observed and apparent altitude**

PyAstronomy.pyasl.co_refract_forward(alt, pressure=1010.0, temperature=10.0)

Converts the observed into the apparent (real) altitude.

The *observed altitude* is the altitude that a star is seen to be with a telescope. This is where it appears in the sky. The observed altitude is always greater than the the *apparent altitude*, which is the altitude that a star would be at, if there were no atmosphere (sometimes called “true” altitude).

**Parameters**

- **alt** [float or array] Observed altitude of an object in DEGREES.
- **pressure** [float or array, optional] Atmospheric pressure in MILLIBAR. Default pressure is 1010 mbar. If a single value is given, it will be used for all given altitudes.
- **temperature** [float or array, optional] Ground temperature in degrees Celsius. Default temperature is 10 Celsius. If a single value is given, it will be used for all given altitudes.

**Returns**

- **Altitude correction** [array] An array holding the altitude correction [deg]. To convert observed altitude into apparent (real) altitude, the correction needs to be subtracted from the observed altitude.
**Notes**

**Note:** This function was ported from the IDL Astronomy User’s Library.

**IDL - Documentation**

function co_refract_forward, a, P=P, T=T

**INPUTS**

a = The observed (apparent) altitude, in DEGREES. May be scalar or vector.

**INPUT KEYWORDS**

P: Pressure [in millibars]. Default is 1010 millibars. [scalar or vector] T: Ground Temp [in Celsius]. Default is 0 Celsius. [scalar or vector]

PyAstronomy.pyasl.co_refract(alt, observer_alt=0.0, pressure=None, temperature=None, epsilon=0.25, convert_to_observed=False, full_output=True)

Convert between apparent (real) altitude and observed altitude.

This routine converts between the apparent (real) altitude of the object, which does not include the influence of the atmosphere, and the observed altitude, which is the altitude as seen through the atmosphere.

The convert_to_observed flag determines the direction of the conversion. By default, observed altitude is converted into apparent altitude.

**Parameters**

alt [float or array] Altitude of an object in DEGREES. Whether the value is interpreted as apparent or observed altitude depends on the convert_to_observed flag. By default, it refers to the apparent (real) altitude.

observer_alt [float or array, optional] Altitude of the observer in METER. Default is 0 (sea level).

pressure [float or array, optional] Atmospheric pressure in MILLIBAR. Default pressure is 1010 mbar. If observer_alt is given, an estimate for the real atmospheric pressure is calculated and used.

temperature [float or array, optional] Atmospheric temperature at the observing location in Celsius. If not specified, the temperature will be calculated assuming a ground temperature of 10 degrees Celsius.

epsilon [float, optional] If convert_to_observed is TRUE, it specifies the accuracy of the calculated altitude in ARCSECONDS that should be reached by the iteration process.

convert_to_observed [boolean, optional] If set True, an iterative method is used to calculate the observed altitude of an object, which includes atmospheric refraction. If False (default), the given altitude will be interpreted as the observed altitude and the apparent (real) altitude will be calculated using co_refract_forward().

full_output [boolean, optional] If True (default), pressure and temperature used in the calculation will be returned as well.

**Returns**

Altitude [array] By default, this will be the observed altitude of the object in degrees. If convert_to_observed was set to False, the number refers to the apparent (real) altitude.

Pressure [array] The pressure [mbar] used in the calculations (only returned if full_output is True).
Temperature [array] The temperature used in the calculations [K] (only returned if full_output is True).

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: CO_REFRACT()

PURPOSE: Calculate correction to altitude due to atmospheric refraction.

DESCRIPTION: CO_REFRACT can calculate both apparent altitude from observed altitude and vice-versa.

CALLING SEQUENCE:

\[
\text{new}_\text{alt} = \text{CO}\_\text{REFRACT}(\text{old}_\text{alt}, [\text{ALTITUDE}=, \text{PRESSURE}=, $\text{TEMPERATURE}=, /\text{TO}\_\text{OBSERVED}, \text{EPSILON}=])
\]

INPUT:

old_alt - Observed (apparent) altitude, in DEGREES. (apparent if keyword /TO_OBSERVED set). May be scalar or vector.

OUTPUT:

Function returns apparent (observed) altitude, in DEGREES. (observed if keyword /TO_OBSERVED set). Will be of same type as input altitude(s).

OPTIONAL KEYWORD INPUTS:

ALTITUDE [The height of the observing location, in meters. This is only used to determine an approximate temperature and pressure, if these are not specified separately. [default=0, i.e. sea level]

PRESSURE : The pressure at the observing location, in millibars. TEMPERATURE: The temperature at the observing location, in Kelvin. EPSILON: When keyword /TO_OBSERVED has been set, this is the accuracy to obtain via the iteration, in arcseconds [default = 0.25 arcseconds].

/TO_OBSERVED: Set this keyword to go from Apparent->Observed altitude,

using the iterative technique.

Note, if altitude is set, but temperature or pressure are not, the program will make an intelligent guess for the temperature and pressure.

DESCRIPTION:

Because the index of refraction of air is not precisely 1.0, the atmosphere bends all incoming light, making a star or other celestial object appear at a slightly different altitude (or elevation) than it really is. It is important to understand the following definitions:

Observed Altitude: The altitude that a star is SEEN to BE, with a telescope. This is where it appears in the sky. This is always GREATER than the apparent altitude.

Apparent Altitude: The altitude that a star would be at, if *there were no atmosphere* (sometimes called “true” altitude). This is usually calculated from an object’s celestial coordinates. Apparent altitude is always LOWER than the observed altitude.
Thus, for example, the Sun’s apparent altitude when you see it right on the horizon is actually -34 arcminutes.

This program uses couple simple formulae to estimate the effect for most optical and radio wavelengths. Typically, you know your observed altitude (from an observation), and want the apparent altitude. To go the other way, this program uses an iterative approach.

EXAMPLE: The lower limb of the Sun is observed to have altitude of 0d 30’. Calculate the the true (=apparent) altitude of the Sun’s lower limb using mean conditions of air pressure and temperature

IDL> print, co_refract(0.5) ===> 0.025degrees (1.55’)

WAVELENGTH DEPENDENCE: This correction is 0 at zenith, about 1 arcminute at 45 degrees, and 34 arcminutes at the horizon FOR OPTICAL WAVELENGTHS. The correction is NON-NEGLIGIBLE at all wavelengths, but is not very easily calculable. These formulae assume a wavelength of 550 nm, and will be accurate to about 4 arcseconds for all visible wavelengths, for elevations of 10 degrees and higher. Amazingly, they are also ACCURATE FOR RADIO FREQUENCIES LESS THAN ~ 100 GHz.

It is important to understand that these formulae really can’t do better than about 30 arcseconds of accuracy very close to the horizon, as variable atmospheric effects become very important.

REFERENCES:


DEPENDENCIES: CO_REFRACT_FORWARD (contained in this file and automatically compiled).

AUTHOR:

Chris O’Dell  Univ. of Wisconsin-Madison
Observational Cosmology Laboratory  Email: odell@cmb.physics.wisc.edu


Fixed problem with vector input when /TO_OBSERVED set W. Landsman Dec 2005 Allow arrays with more than 32767 elements W.Landsman/C.Dickinson Feb 2010

Example: co_refract_forward and co_refract

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import datetime
import numpy as np

# Assume, a star is observed at an altitude of 50 degrees
alt = 50.
# Now one wants to know the real altitude of the star, i.e.,
# the altitude corrected for atmospheric refraction.
print()
print("Get apparent (real) altitude of a star with observed altitude of " +
      str(alt) + " degrees")
```

(continues on next page)
print(" -> Apparent altitude = ", alt - pyasl.co_refract_forward(alt))

print()
print("You are not observing from sea level, but from an altitude of 5000 meter.")
print("Apparent altitude = %.9f, estimated pressure [mbar] = %.9f, " +
      "estimated temperature [K] = %.9f") %
      pyasl.co_refract(alt, observer_alt=5000, convert_to_observed=False))

print()
print("Convert apparent (real) altitude into observed altitude.")
print("Apparent altitude = " + str(alt) + " degrees", end=' '
print(" -> Observed altitude = " + str(pyasl.co_refract(alt, full_output=False,
      convert_to_observed=True)[0]))

print()
print("The same object observed from different observer altitudes")
apparentAltitudes = np.repeat(30.0, 10)
obsalts = np.linspace(0., 5000., len(apparentAltitudes))
r = pyasl.co_refract(apparentAltitudes, observer_alt=obsalts, convert_to_˓
->observed=True)
for i in range(len(r[0])):
  print("Observed altitude [deg] = %g, pressure [mbar] = %g, temperature [K] = %g" %
      (r[0][i], r[1][i], r[2][i]))

Horizontal coordinates

Convert hour angle and declination into horizontal coordinates

PyAstronomy.pyasl.hadec2altaz(ha, dec, lat, ws=False, radian=False)

Convert hour angle and declination into horizon (alt/az) coordinates.

Parameters

ha [float or array] Local apparent hour angle in DEGREES.
dec [float or array] Local apparent declination in DEGREES.
lat [float or array] Local latitude in DEGREES.
radian [boolean, optional] If True, the result is returned in radian instead of in degrees (default is False).
ws [boolean, optional] Set this to True, if the azimuth shall be measured West from South. Default is to measure azimuth East from North.

Returns

Altitude [list] A list holding the Local Apparent Altitude [deg].
Apparent Azimuth [list] The Local Apparent Azimuth [deg].

Notes

Note: This function was ported from the IDL Astronomy User’s Library.
**NAME:** HADEC2ALTAZ

**PURPOSE:** Converts Hour Angle and Declination to Horizon (alt-az) coordinates.

**EXPLANATION:** Can deal with NCP/SCP singularity. Intended mainly to be used by program EQ2HOR

**CALLING SEQUENCE:** HADEC2ALTAZ, ha, dec, lat ,alt, az [ /WS ]

**INPUTS**
- **ha** - the local apparent hour angle, in DEGREES, scalar or vector
- **dec** - the local apparent declination, in DEGREES, scalar or vector
- **lat** - the local latitude, in DEGREES, scalar or vector

**OUTPUTS**
- **alt** - the local apparent altitude, in DEGREES.
- **az** - the local apparent azimuth, in DEGREES, all results in double precision

**OPTIONAL KEYWORD INPUT:**
- **/WS** - Set this keyword for the output azimuth to be measured West from South. The default is to measure azimuth East from North.

**EXAMPLE:**
What were the apparent altitude and azimuth of the sun when it transited the local meridian at Pine Bluff Observatory (Lat=+43.07833 degrees) on April 21, 2002? An object transits the local meridian at 0 hour angle. Assume this will happen at roughly 1 PM local time (18:00 UTC).

```
IDL> jdcnv, 2002, 4, 21, 18., jd ; get rough Julian date to determine Sun ra, dec.
IDL> sunpos, jd, ra, dec
IDL> hadec2altaz, 0., dec, 43.078333, alt, az
===> Altitude alt = 58.90 Azimuth az = 180.0
```

**REVISION HISTORY:** Written Chris O’Dell Univ. of Wisconsin-Madison May 2002

---

Example:

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import datetime
import numpy as np

# Hour angle 0. means transiting the local meridian.
ha = 0.
# Declination of object
dec = 30.
# Latitude of the observer (here Hamburger Sternwarte)
lat = +53.48
print("Get altitude and azimuth of object in DEGREES")
print(pyasl.hadec2altaz(ha, dec, lat))

# List of coordinates
ha = np.arange(0., 20., 5.)
dec = np.arange(30., 50., 5.)
lat = np.zeros(dec.size)+53.48
print()
print("Get altitude and azimuth for a list of objects from same observer latitude")
altaz = pyasl.hadec2altaz(ha, dec, lat)
print("alt: ", altaz[0])
print("az: ", altaz[1])
```
Convert celestial coordinates (RA/DEC) into local horizon coordinates (ALT/AZ)

PyAstronomy.pyasl.eq2hor(jd, ra, dec, observatory=None, lon=None, lat=None, alt=None, B1950=False, precess=True, nutate=True, aberration=True, refract=True)

Convert celestial coordinates (RA/DEC) to local horizon coordinates (ALT/AZ).

This routine is typically accurate to about 1 arcsec. It considers Earth’s precession, nutation, aberration, and refraction (if keywords are True).

**Parameters**

- **jd** [float or array] The Julian date(s)
- **ra** [float or array] The right ascension in DEGREES.
- **dec** [float or array] The declination in DEGREES.
- **observatory** [string, {HS}, optional] A string identifying the observatory. If given, the observer’s longitude, latitude, and altitude are set automatically (and must not be given separately then).
- **lon** [float] East longitude of the observer in DEGREES. Specify West longitude with a negative sign. Default is the longitude of the Hamburger Sternwarte.
- **lat** [float] Latitude of the observer in DEGREES. Default is the latitude of the Hamburger Sternwarte.
- **alt** [float] Altitude of the observer in METER. Default is the altitude of the Hamburger Sternwarte.
- **B1950** [boolean, optional] If True, your RA and DEC coordinates are given for epoch B1950 FK4. If False, RA and DEC are given for epoch J2000 FK5. Default is FALSE.
- **precess** [boolean, optional] If True (default), Earth’s precess motion is considered in the calculations.
- **nutate** [boolean, optional] If True (default), Earth’s nutation is considered in the calculations.
- **aberration** [boolean, optional] If True (default), the annual aberration is considered in the calculations.
- **refraction** [boolean, optional] If True, the atmospheric refraction is considered in the calculations.

**Returns**

- **Altitude** [float or array] The altitude in degrees.
- **Azimuth** [float or array] The azimuth in degrees (measured East from North).
- **Hour angle** [float or array] The hour angle in degrees.

**Notes**

- **Note:** This function was ported from the IDL Astronomy User’s Library.

**IDL - Documentation**

**NAME:** EQ2HOR
**PURPOSE:** Convert celestial (ra-dec) coords to local horizon coords (alt-az).

**CALLING SEQUENCE:**
```python
eq2hor, ra, dec, jd, alt, az, [ha, LAT= , LON= , /WS, OBSNAME= , $ /B1950 , PRECESS_= 0, 
NUTATE_= 0, REFRACT_= 0, $ ABERRATION_= 0, ALTITUDE= , /VERBOSE, _EXTRA= ]
```

**DESCRIPTION:** This code calculates horizon (alt,az) coordinates from equatorial (ra,dec) coords. It is typically accurate to about 1 arcsecond or better (I have checked the output against the publicly available XEPHEM software). It performs precession, nutation, aberration, and refraction corrections. The perhaps best thing about it is that it can take arrays as inputs, in all variables and keywords EXCEPT Lat, lon, and Altitude (the code assumes these aren’t changing), and uses vector arithmetic in every calculation except when calculating the precession matrices.

**INPUT VARIABLES:**
- **RA** [Right Ascension of object (J2000) in degrees (FK5); scalar or] vector.
- **Dec** : Declination of object (J2000) in degrees (FK5), scalar or vector.
- **JD** : Julian Date [scalar or vector]

**Note:** if RA and DEC are arrays, then alt and az will also be arrays. If RA and DEC are arrays, JD may be a scalar OR an array of the same dimensionality.

**OPTIONAL INPUT KEYWORDS:**
- **lat**: north geodetic latitude of location in degrees lon : EAST longitude of location in degrees (Specify west longitude with a negative sign.)

- **WS** [Set this to get the azimuth measured westward from south (not] East of North).

- **obsname**: Set this to a valid observatory name to be used by the astrolib OBSERVATORY procedure, which will return the latitude and longitude to be used by this program.

- **B1950** [Set this if your ra and dec are specified in B1950, FK4] coordinates (instead of J2000, FK5)

- **precess** [Set this to 1 to force precession [default], 0 for no] precession correction

- **nutate** : Set this to 1 to force nutation [default], 0 for no nutation. aberration : Set this to 1 to force aberration correction [default],

  0 for no correction.

- **refrac**t [Set to 1 to force refraction correction [default], 0 for no] correction.

- **altitude**: The altitude of the observing location, in meters. [default=0].

- **verbose**: Set this for verbose output. The default is verbose=0.

- **extra**: This is for setting TEMPERATURE or PRESSURE explicitly, which are used by CO_REFRACT to calculate the refraction effect of the atmosphere. If you don’t set these, the program will make an intelligent guess as to what they are (taking into account your altitude). See CO_REFRACT for more details.

**OUTPUT VARIABLES:** (all double precision)
- **alt** : altitude (in degrees) az : azimuth angle (in degrees, measured EAST from NORTH, but see keyword WS above.)

- **ha** : hour angle (in degrees) (optional)

**DEPENDENCIES:** NUTATE, PRECESS, OBSERVATORY, SUNPOS, ADSTRING() CO_NUTATE, CO_ABERRATION, CO_REFRACT, ALTAZ2HADEC, SETDEFAULTVALUE
BASIC STEPS  Apply refraction correction to find apparent Alt. Calculate Local Mean Sidereal Time Calculate Local Apparent Sidereal Time Do Spherical Trig to find apparent hour angle, declination. Calculate Right Ascension from hour angle and local sidereal time. Nutation Correction to Ra-Dec Aberration correction to Ra-Dec

Precess Ra-Dec to current equinox.

ORRECTIONS I DO NOT MAKE:

• Deflection of Light by the sun due to GR. (typically milliarcseconds, can be arcseconds within one degree of the sun)
• The Effect of Annual Parallax (typically < 1 arcsecond)
• and more (see below)

TO DO

• Better Refraction Correction. Need to put in wavelength dependence, and integrate through the atmosphere.

• Topocentric Parallax Correction (will take into account elevation of the observatory)

• Proper Motion (but this will require crazy lookup tables or something).
  – Difference between UTC and UT1 in determining LAST – is this important?
  – Effect of Annual Parallax (is this the same as topocentric Parallax?)

• Polar Motion
  – Better connection to Julian Date Calculator.

EXAMPLE

Find the position of the open cluster NGC 2264 at the Effelsburg Radio Telescope in Germany, on June 11, 2023, at local time 22:00 (METDST). The inputs will then be:

Julian Date = 2460107.250 Latitude = 50d 31m 36s Longitude = 06h 51m 18s Altitude = 369 meters RA (J2000) = 06h 40m 58.2s Dec(J2000) = 09d 53m 44.0s

IDL> eq2hor, ten(6,40,58.2)*15., ten(9,53,44), 2460107.250d, alt, az, $
lat=ten(50,31,36), lon=ten(6,51,18), altitude=369.0, /verb, $
 pres=980.0, temp=283.0

The program produces this output (because the VERBOSE keyword was set)

Latitude = +50 31 36.0 Longitude = +06 51 18.0 Julian Date = 2460107.250000 Ra, Dec: 06 40 58.2 +09 53 44.0 (J2000) Ra, Dec: 06 42 15.7 +09 52 19.2 (J2023.4422) Ra, Dec: 06 42 13.8 +09 52 26.9 (fully corrected) LMST = +11 46 42.0 LAST = +11 46 41.4 Hour Angle = +05 04 27.6 (hh:mm:ss) Az, El = 17 42 25.6 +16 28 22.8 (Observer Coords) Az, El = 17 42 25.6 +16 28 22.8 (Observer Coords)

Compare this with the result from XEPHEM: Az, El = 17h 42m 25.6s +16d 28m 21s

This 1.8 arcsecond discrepancy in elevation arises primarily from slight differences in the way I calculate the refraction correction from XEPHEM, and is pretty typical.

AUTHOR:

Chris O'Dell  Univ. of Wisconsin-Madison

Observational Cosmology Laboratory Email: odell@cmb.physics.wisc.edu
Example:

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import matplotlib.pylab as plt
import datetime
import numpy as np

# Convert calendar date to JD
# use the datetime package
jd = datetime.datetime(2013, 4, 16)
jd = pyasl.jdcnv(jd)
# Specific RA and DEC
ra = 10.
dec = 30.
print()
print("Get horizontal coordinates (alt, az, ha) from JD, RA,")
print(" and DEC for the Hamburger Sternwarte")
print(pyasl.eq2hor(jd, ra, dec, observatory="HS"))

print()
print("From a list of Julian dates ...")
jds = np.arange(jd, jd+1, .2)
ras = np.zeros(jds.size) + ra
decs = np.zeros(jds.size) + dec
alt, az, ha = pyasl.eq2hor(jds, ras, decs, lon=-70.4042, lat=-24.6272, alt=2635.)
for i in range(alt.size):
    print("JD = %g: alt = %g, az = %g, ha = %g" % (jds[i], alt[i], az[i], ha[i]))

print()
print("For one object and different times at the VLT...")
jds = np.arange(jd-.25, jd+.25, .01)
ras = np.zeros(jds.size) + 130.
decs = np.zeros(jds.size) - 30.
res = pyasl.eq2hor(jds, ras, decs, lon=-70.4042, lat=-24.6272, alt=2635.)
plt.plot(jds, res[0])
plt.xlabel("Julian date")
plt.ylabel("Altitude [deg]")
plt.show()
```

Lunar phase and position

PyAstronomy.pyasl.moonpos (jd, radian=False)

Computes RA and DEC of the Moon at given Julian date(s).

Parameters

jd [float or array] The Julian date.
PyAstronomy Documentation, Release 0.15.2

radian  [boolean, optional] If True, results are returned in RADIANS instead of DEGREES. Default is False.

Returns

RA  [float or array] Right ascension of the moon for given JD(s) in DEGREES.
DEC  [float or array] Declination of the moon for given JD(s) in DEGREES.
DISTANCE  [float or array] Distance of the moon from the Earth for given JD(s) in KILOMETERS.
GEOLONGITUDE  [float or array] Apparent longitude of the moon for given JD(s) in DEGREES.
GEOLATITUDE  [float or array] Apparent latitude of the moon for given JD(s) in DEGREES.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

PRO MOONPOS, jd, ra, dec, dis, geolong, geolat, RADIANS = radian
NAME: MOONPOS
PURPOSE: To compute the RA and Dec of the Moon at specified Julian date(s).
CALLING SEQUENCE: MOONPOS, jd, ra, dec, dis, geolong, geolat, [/RADIANS ]
INPUTS: JD - Julian ephemeris date, scalar or vector, double precision suggested
OUTPUTS:

Ra - Apparent right ascension of the moon in DEGREES, referred to the true equator of the specified date(s)
Dec - The declination of the moon in DEGREES Dis - The Earth-moon distance in kilometers (between the center of the Earth and the center of the Moon).

Geolong - Apparent longitude of the moon in DEGREES, referred to the ecliptic of the specified date(s)
Geolat - Apparent latitude of the moon in DEGREES, referred to the ecliptic of the specified date(s)

The output variables will all have the same number of elements as the input Julian date vector, JD. If JD is a scalar then the output variables will be also.

OPTIONAL INPUT KEYWORD:

/RADIANS - If this keyword is set and non-zero, then all output variables are given in Radians rather than Degrees

EXAMPLES:

(1) Find the position of the moon on April 12, 1992
IDL> jdcnv,1992,4,12,0,jd ;Get Julian date
IDL> moonpos, jd, ra, dec ;Get RA and Dec of moon
IDL> print,adstring(ra,dec,1)

=> 08 58 45.23 +13 46 6.1

This is within 1" from the position given in the Astronomical Almanac

(2) Plot the Earth-moon distance for every day at 0 TD in July, 1996

IDL> jdcnv,1996,7,1,0,jd ;Get Julian date of July 1
IDL> moonpos,jd+dindgen(31), ra, dec, dis ;Position
at all 31 days
IDL> plot,indgen(31),dis, /YNOZ

METHOD: Derived from the Chapront ELP2000/82 Lunar Theory (Chapront-Touze’ and Chapront, 1983, 124, 50), as described by Jean Meeus in Chapter 47 of “Astronomical Algorithms” (Willmann-Bell, Richmond), 2nd edition, 1998. Meeus quotes an approximate accuracy of 10” in longitude and 4” in latitude, but he does not give the time range for this accuracy.

Comparison of this IDL procedure with the example in “Astronomical Algorithms” reveals a very small discrepancy (~1 km) in the distance computation, but no difference in the position calculation.

This procedure underwent a major rewrite in June 1996, and the new calling sequence is incompatible with the old (e.g. angles now returned in degrees instead of radians).

PROCEDURES CALLED: CIRRANGE, ISARRAY(), NUTATE, TEN() - from IDL Astronomy Library
POLY() - from IDL User's Library


accuracy, W. Landsman Hughes STX June 1996

Added /RADIANS keyword W. Landsman August 1997 Converted to IDL V5.0 W. Landsman September 1997 Use improved expressions for L’,D,M,M’, and F given in 2nd edition of

Meeus (very slight change), W. Landsman November 2000

Avoid 32767 overflow W. Landsman January 2005

Example: Finding the position of the Moon

```python
from __future__ import print_function, division
import datetime
from PyAstronomy import pyasl
import numpy as np

# Convert calendar date to JD
# using the datetime package
jd = datetime.datetime(2013, 4, 16)
jd = pyasl.jdcnv(jd)
jd = np.arange(jd, jd + 20, 1)
# Calculate Moon positions
res = pyasl.moonpos(jd)

print("%15s %8s %8s %11s %8s %8s"
("
JD", "RA", "DEC", "DIST", "GEOLON", "GEOLAT")
print("%15s %8s %8s %11s %8s %8s"
("
[d]", "[deg]", "[deg]", "[km]", "[deg]", "[deg]"
))
for i in range(jd.size):
    print("%15.4f %8.4f %8.4f %11.4f %8.4f %8.4f"
(jd[i], res[0][i], res[1][i], res[2][i], res[3][i], res[4][i]))
```
PyAstronomy Documentation, Release 0.15.2

PyAstronomy.pyasl.moonphase \( (jd) \)

Computes the illuminated fraction of the Moon at given Julian date(s).

**Parameters**

\( jd \) [float or array] The Julian date.

**Returns**

**Fraction** [float or array] The illuminated fraction \([0 - 1]\) of the Moon. Has the same size as \( jd \).

**Notes**

Note: This function was ported from the IDL Astronomy User’s Library.

**IDL - Documentation**

**NAME:** MPHASE  
**PURPOSE:** Return the illuminated fraction of the Moon at given Julian date(s)  
**CALLING SEQUENCE:** MPHASE, jd, k  
**INPUT:** JD - Julian date, scalar or vector, double precision recommended  
**OUTPUT:**  
\( k \) - illuminated fraction of Moon’s disk \((0.0 < k < 1.0)\), same number of elements as \( jd \). \( k = 0 \) indicates a new moon, while \( k = 1 \) for a full moon.

**EXAMPLE:** Plot the illuminated fraction of the moon for every day in July 1996 at 0 TD (~Greenwich noon).

IDL> jdcnv, 1996, 7, 1, 0, jd ;Get Julian date of July 1  
IDL> mphase, jd+dindgen(31), k ;Moon phase for all 31 days  
IDL> plot, indgen(31),k ;Plot phase vs. July day number

**Example: Find the Moon’s illuminated fraction**

```python
from __future__ import print_function, division  
import datetime  
from PyAstronomy import pyasl  
import numpy as np

# Convert calendar date to JD
# using the datetime package
jd = datetime.datetime(2013, 4, 16)  
jd = pyasl.jdcnv(jd)  
jd = np.arange(jd, jd+20, 1)  
mp = pyasl.moonphase(jd)

print("%15s %3s \n%15.4f %3d%% \n")
for i in range(jd.size):
    print("%15.4f %3d%% \n\n")
```
Convert azimuth into cardinal point

The cardinal points or cardinal directions are North, East, South, and West.

**PyAstronomy.pyasl.getCardinalPoint**(azimuth)

Get the cardinal point (North, East, South, or West) for a given azimuth.

Here, the azimuth is measured from North to East. N = (315, 45] deg E = (45, 135] deg S = (135, 225] deg W = (225, 315] deg

**Parameters**

azimuth [float] Azimuth of an object in deg (0-360).

**Returns**

Cardinal point [string, {N,E,S,W}] Returns the cardinal point (N, E, S, W) corresponding to the given azimuth of the object.

**Example**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np

# Get the cardinal point for 10 azimuth angles
azimuths = np.random.random(10) * 360.
for azimuth in azimuths:
    cp = pyasl.getCardinalPoint(azimuth)
    print("Azimuth: {0:6.2f} deg, Cardinal point: {1:1s}".format(azimuth, cp))
```

Name twilight given solar altitude

**PyAstronomy.pyasl.twilightName**(sunAlt)

Determine the type of twilight for a given altitude of the Sun.

**Definitions of twilight:**

- Solar altitude > 0 : day
- Solar altitude <= 0 : civil twilight
- Solar altitude <= -6 : nautical twilight
- Solar altitude <= -12 : astronomical twilight
- Solar altitude <= -18 : night

**Parameters**

sunAlt [float] Altitude of the Sun in degrees.

**Returns**

Twilight flag [string, {“day”, “civil twilight”, “nautical twilight”, “astron. twilight”, “night”}] Specifies the type of twilight corresponding to the given solar altitude.

The twilight flag is either “day”, “civil twilight”, “nautical twilight”, “astron. twilight”, or “night”.

2.1. Python AstroLib
Example

```
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np

for alt in np.linspace(-20., 5., 15):
    print("Altitude = {0:6.2f}, Twilight is called: ",
          pyasl.twilightName(alt))
```

Get angular distance from RA and DEC

PyAstronomy.pyasl.**getAngDist**(ra1, dec1, ra2, dec2)

Calculate the angular distance between two coordinates.

**Parameters**

- **ra1** [float, array] Right ascension of the first object in degrees.
- **dec1** [float, array] Declination of the first object in degrees.
- **ra2** [float, array] Right ascension of the second object in degrees.
- **dec2** [float, array] Declination of the second object in degrees.

**Returns**

- **Angle** [float, array] The angular distance in DEGREES between the first and second coordinate in the sky.

Example

```
from __future__ import print_function, division
from PyAstronomy import pyasl

print("Angular distance between the poles (deg):")
print(pyasl.getAngDist(98.0, -90.0, 100., +90.0))

print("Angular distance between Vega and Altair (deg)")
print(pyasl.getAngDist(279.23473479, +38.78368896,297.69582730, +08.86832120))
```

Calculate the position angle

PyAstronomy.pyasl.**positionAngle**(ra1, dec1, ra2, dec2, positive=True)

Compute the position angle.

The position angle is measured from the first position from North through East. If the **positive** flag is set True (default) the result will be given as an angle from 0 to 360 degrees. If the flag is set False, the scale is -180 to 180 degrees with negative number increasing from North through West; note that this is the behavior of the posAng IDL routine.

**Parameters**

- **ra1** [float] Right ascension of first object [deg].
- **dec1** [float] Declination of first object [deg].
ra2  [float] Right ascension of second object [deg].

dec2  [float] Declination of second object [deg].

positive  [boolean, optional] If True (default), the output will be given as an angle between 0 and 360 degrees. Otherwise, the angle ranges from -180 to +180 degrees.

Returns

Position angle  [float] The position angle in degrees.

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: POSANG

PURPOSE: Computes rigorous position angle of source 2 relative to source 1

EXPLANATION: Computes the rigorous position angle of source 2 (with given RA, Dec) using source 1 (with given RA, Dec) as the center.

CALLING SEQUENCE: POSANG, U, RA1, DC1, RA2, DC2, ANGLE

INPUTS:

U – Describes units of inputs and output: 0: everything radians 1: RAx in decimal hours, DCx in decimal degrees, ANGLE in degrees

RA1 – Right ascension of point 1
DC1 – Declination of point 1
RA2 – Right ascension of point 2
DC2 – Declination of point 2

OUTPUTS:

ANGLE– Angle of the great circle containing [ra2, dc2] from the meridian containing [ra1, dc1], in the sense north through east rotating about [ra1, dc1]. See U above for units.

PROCEDURE: The “four-parts formula” from spherical trig (p. 12 of Smart’s Spherical Astronomy or p. 12 of Green’ Spherical Astronomy).

EXAMPLE: For the star 56 Per, the Hipparcos catalog gives a position of RA = 66.15593384, Dec = 33.94988843 for component A, and RA = 66.15646079, Dec = 33.96100069 for component B. What is the position angle of B relative to A?

IDL> RA1 = 66.15593384/15.d & DC1 = 33.95988843 IDL> RA2 = 66.15646079/15.d & DC2 = 33.96100069 IDL> posang,1,ra1,dc1,ra2,dc2, ang

will give the answer of ang = 21.4 degrees

NOTES: (1) If RA1,DC1 are scalars, and RA2,DC2 are vectors, then ANGLE is a vector giving the position angle between each element of RA2,DC2 and RA1,DC1. Similarly, if RA1,DC1 are vectors, and RA2, DC2 are scalars, then DIS is a vector giving the position angle of each element of RA1, DC1 and RA2, DC2. If both RA1,DC1 and RA2,DC2 are vectors then ANGLE is a vector giving the position angle between
each element of RA1,DC1 and the corresponding element of RA2,DC2. If then vectors are not the same length, then excess elements of the longer one will be ignored.

(2) Note that POSANG is not commutative – the position angle between A and B is theta, then the position angle between B and A is 180+theta

**PROCEDURE CALLS**: ISARRAY()

**HISTORY**: Modified from GCIRC, R. S. Hill, RSTX, 1 Apr. 1998 Use V6.0 notation
W.L. Mar 2011

**Example: Position angle of Alcor and Mizar**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Position of Mizar: 200.98141867 +54.92535197
# Position of Alcor: 201.30640764 +54.98795966

# Calculate position angle
r = pyasl.positionAngle(200.98141867, +54.92535197, 201.30640764,+54.98795966)

print("Position angle of Alcor (from Mizar): %4.2f deg" % r)
```

**Calculate the airmass**

**Plane-parallel atmosphere**

```python
PyAstronomy.pyasl.airmassPP(zangle)
```

Calculate airmass for plane parallel atmosphere.

**Parameters**

- `zangle` [float or array] The zenith angle in degrees.

**Returns**

- `Airmass` [float or array] The airmass assuming a plane parallel atmosphere.

**Example: Airmass with plane-parallel atmosphere**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

print("Airmass for plane-parallel atmosphere")
for za in range(0,70,10):
    print("Zenith angle: %2d deg, airmass = %7.2f" % \
    (za, pyasl.airmassPP(za)))
```
Airmass for spherical, homogeneous atmosphere

PyAstronomy.pyasl.airmassSpherical(zangle, obsAltitude, rearth=6371.0, yatm=10.0)

Calculate the airmass for a given zenith angle and observer altitude.

This routine uses a geometric formula for a homogeneous, spherical atmosphere with an elevated observer.

Note: In this model, the airmass is not necessarily one toward the zenith.

Parameters

- **zangle** [float, array] Zenith angle of an object in deg.
- **obsAltitude** [float] Elevation of the observer in meter.
- **rearth** [float, optional] Earth’s radius in km.
- **yatm** [float, optional] Height of the atmosphere in km.

Returns

- **Airmass** [float or array] The airmass.

Example: Airmass with homogeneous, spherical atmosphere

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
obsAlt = 2400.0
for za in range(0, 90, 10):
    print("Zenith angle: %2d, airmass = %7.2f"
          % (za, pyasl.airmassSpherical(za, obsAlt)))
```

Observatory locations

PyAstronomy.pyasl.listObservatories(show=True)

Get a list of available observatories.

Parameters

- **show** [boolean, optional] If True (default), the observatory data will be written to screen.

Returns

- **Observatory data** [dictionary] For every observatory code (key), the dictionary holds another dictionary with “longitude”, “latitude”, “altitude”, “name”, and “tz”. Longitude and latitude are given in degrees and the observatory’s altitude in meter. Finally, tz defines the time zone (hours West of Greenwich) and “name” gives the full observatory name.

PyAstronomy.pyasl.observatory(obsName)

Get the location of an observatory.

This function emulates the behavior of the IDL Astrolib’s observatory.pro procedure. Use showObservatories to get a list of known observatories.
Parameters

**obsName** [string] The observatory code. Use `listObservatories` to get a list of available observatories and their codes.

Returns

**Observatory location** [dictionary] The keys of this dictionary are: “longitude”, “latitude”, “altitude”, and “tz”. Longitude and latitude are given in degrees and the observatory’s altitude in meter. Finally, tz defines the time zone (hours West of Greenwich).

### Example of usage

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# List all available observatory data
pyasl.listObservatories()

print()
print("Data for Kitt Peak National Observatory")
print(pyasl.observatory("kpno"))
print("(longitude and latitude in degrees, altitude in meters, and")
print("time zone in hours West of Greenwich")
```

### Conversion between sexagesimal and decimal coordinate representation

The routines below convert between sexagesimal and decimal coordinate representations.

**Warning:** Until (and including) version 0.11.0, the sign of declinations starting with ‘-00’ was not handled correctly.

### Example: Convert between decimal and sexagesimal representation

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Coordinates of HD 1 from SIMBAD
hd1 = "00 05 08.83239 +67 50 24.0135"

print("Coordinates of HD 1 (SIMBAD): ", hd1)

# Obtain decimal representation
ra, dec = pyasl.coordsSexaToDeg(hd1)
print("Coordinates of HD 1 [deg]: \$010.6f \$+09.6f" % (ra, dec))

# Convert back into sexagesimal representation
sexa = pyasl.coordsDegToSexa(ra, dec)
print("Coordinates of HD 1 [sexa]: ", sexa)
```
### Routines

**PyAstronomy.pyasl.coordsSexaToDeg** (*c, fullOut=False*)

Convert sexagesimal coordinate string into degrees.

**Parameters**

- **c** [string] The coordinate string. Valid formats are, e.g., “00 05 08.83239 +67 50 24.0135” or “00:05:08.83239 -67:50:24.0135”. Spaces or colons are allowed as separators for the individual components of the coordinates.

- **fullOut** [boolean, optional] If True, two additional tuples holding the individual components of the right ascension and declination specification will be returned. The default is False.

**Returns**

- **ra, dec** [float] Right ascension and declination in degrees.
- **hms, dms** [tuples of three floats] If fullOut is True, two tuples of three numbers each will be returned, which hold the individual constituents making up the right ascension (hms) and declination (dms) specifiers in sexagesimal representation.

**PyAstronomy.pyasl.coordsDegToSexa** (*ra, dec, asString=True, fmt=('%02d %02d %06.3f ', '%s%02d %02d %06.3f'))

Convert right ascension and declination from degrees into sexagesimal representation.

**Parameters**

- **ra** [float] Right ascension in degrees.
- **dec** [float] Declination in degrees.
- **asString** [boolean, optional] If True (default), the result will be a string formatted according to the rules specified by fmt.
- **fmt** [tuple of strings, optional] The output format used to create the output string (first ra, second dec). Only used if asString is True (default).

**Returns**

- **Coordinates** [string or tuple] If asString is True (default), a string holding the coordinates is returned, which is formatted according to the rules specified by the fmt parameter. If False, a tuple of two tuples is returned, of which the first holds three numbers representing the right ascension (hms) and the second three numbers representing the declination (dms, sign).

**PyAstronomy.pyasl.hmsToDeg** (*h, m, s*)

Convert hour-minute-second specification into degrees.

**Parameters**

- **h** [float] Hours (0-24)
- **m** [float] Minutes (time, 0-60)
- **s** [float] Seconds (time, 0-60)

**Returns**

- **Angle** [float] The corresponding angle in degrees.

**PyAstronomy.pyasl.degToHMS** (*d*)

Convert degrees into time units (hours-minutes-seconds)

**Parameters**

---

2.1. Python AstroLib
d  [float] Degrees (0-360)

Returns

h, m, s: float  Hours, minutes, and seconds

PyAstronomy.pyasl.dmsToDeg(d, m, s, esign=0)
Convert degree-arcminute-arcsecond specification into degrees.

Parameters

d  [float] Degrees.
m  [float] Arcminutes (0-60)
s  [float] Arcseconds (0-60)
esign  [int, optional, {-1,0,1}] Explicit sign with -1 representing negative sign, +1 representing positive sign, and 0 indicating no explicit sign specification. The explicit sign is necessary if negative southern coordinates are specified but d is 0 and, thus, cannot carry the sign.

Returns

Angle  [float] The corresponding angle in degrees.

PyAstronomy.pyasl.degToDMS(g)
Convert degrees into arc units (degrees, (arc)minutes, (arc)seconds)

Parameters

g  [float] Value in degrees.

Returns

d, m, s, sign  [float, int] Degrees, (arc)minutes, and (arc)seconds. Note that only the degree number is signed. The sign (+1/-1) is also returned to yield a complete result if the value of degree (d) is zero.

Apply proper motion

PyAstronomy.pyasl.applyProperMotion(ra, dec, pmra, pmdec, dt, fixes=1)
Apply proper motion to coordinates

Parameters

ra, dec  [float or array] Coordinates at reference epoch
pmra, pmdec  [float or array] Proper motion in mas/yr.
dt  [float] Elapsed time [yr]
fixes  [int, optional] Number of steps in which the proper motion is applied. Default is one.

Example: Proper motion of GJ 1215

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Year 2018 coordinates and proper motion (mas/yr)
# of GJ 1215
ra, dec = 259.4319131981014, +11.6678903453170
```
pmra, pmdec = -352.897, -445.558

# Calculate 2050 position
ra5, dec5 = pyasl.applyProperMotion(ra, dec, pmra, pmdec, dt=2050-2018, fixes=1)

print("Position 2018: %10.6f % 10.6f" % (ra, dec))
print("Position 2050: %10.6f % 10.6f" % (ra5, dec5))

Galactic space velocity (UVW)

PyAstronomy.pyasl.gal_uvw (ra, dec, pmra, pmdec, distance, vrad, lsr=None)
Calculate the Galactic space velocity (U,V,W)

Note: In contrast to the IDL implementation, U is taken positive toward the galactic center.

Parameters

ra, dec [float or array] Position in degrees
pmra, pmdec [float or array] Proper motion in arcsec/yr
distance [float or array] Distance in pc
vrad [float or array] Radial velocity in km/s
lsr [three-float tuple, optional] If given, the output UVW velocities are corrected for the solar motion (U,V,W)_Sun to the local standard of rest (LSR). Use, e.g., lsr = (8.5, 13.38, 6.49) from Coskunoglu et al. 2011, MNRAS 412, 1237-1245. Be aware that the value of the solar motion through the LSR remains poorly determined.

Returns

U, V, W [float or arrays] U: Velocity (km/s) positive toward the Galactic center V: Velocity (km/s) positive in the direction of Galactic rotation W: Velocity (km/s) positive toward the North Galactic Pole

Notes

Note: This function was ported from the IDL Astronomy User’s Library.

IDL - Documentation

NAME: GAL_UVW
PURPOSE: Calculate the Galactic space velocity (U,V,W) of star
EXPLANATION: Calculates the Galactic space velocity U, V, W of star given its (1) coordinates, (2) proper motion, (3) distance (or parallax), and (4) radial velocity.
CALLING SEQUENCE:

GAL_UVW [ /LSR, RA=, DEC=, PMRA=, PMDEC=, VRAD=, DISTANCE=, PLX= ]
OUTPUT PARAMETERS: U - Velocity (km/s) positive toward the Galactic anti-center
V - Velocity (km/s) positive in the direction of Galactic rotation
W - Velocity (km/s) positive toward the North Galactic Pole

REQUIRED INPUT KEYWORDS:
User must supply a position, proper motion, radial velocity and distance (or parallax). Either scalars or vectors can be supplied.

(1) Position: RA - Right Ascension in Degrees
Dec - Declination in Degrees
(2) Proper Motion: PMRA = Proper motion in RA in arc units (typically milli-arcseconds/yr)
PMDEC = Proper motion in Declination (typically mas/yr)
(3) Radial Velocity: VRAD = radial velocity in km/s
(4) Distance or Parallax: DISTANCE - distance in parsecs
or
PLX - parallax with same distance units as proper motion measurements typically milliarcseconds (mas)

OPTIONAL INPUT KEYWORD:
/LSR - If this keyword is set, then the output velocities will be corrected for the solar motion (U,V,W)_Sun = (-8.5, 13.38, 6.49) (Coskunoglu et al. 2011 MNRAS) to the local standard of rest. Note that the value of the solar motion through the LSR remains poorly determined.

EXAMPLE:
(1) Compute the U,V,W coordinates for the halo star HD 6755. Use values from Hipparcos catalog, and correct to the LSR
ra = ten(1,9,42.3)*15. & dec = ten(61,32,49.5) pmra = 627.89 & pmdec = 77.84 ;mas/yr dis = 144
& vrad = -321.4 gal_uvw,u,v,w,ra=ra,dec=dec,pmra=pmra,pmdec=pmdec,vrad=vrad,dis=dis,lsr
=====> u=154 v = -493 w = 97 ;km/s
(2) Use the Hipparcos Input and Output Catalog IDL databases (see http://idlastro.gsfc.nasa.gov/ftp/zdbase/) to obtain space velocities for all stars within 10 pc with radial velocities > 10 km/s
dbopen,'hipparcos,hic' ;Need Hipparcos output and input catalogs list = dbfind('plx>100,vrad>10') ;Plx > 100 mas, Vrad > 10 km/s
dbext,list,'pmra,pmdec,vrad,ra,dec,plx',pmra,pmdec,vrad,ra,dec,plx ra = ra*15. ;Need right ascension in degrees
GAL_UVW,u,v,w,ra=ra,dec=dec,pmra=pmra,pmdec=pmdec,vrad=vrad,plx=plx forcprint,u,v,w ;Display results

METHOD: Follows the general outline of Johnson & Soderblom (1987, AJ, 93,864) except that U is positive outward toward the Galactic anti-center, and the J2000 transformation matrix to Galactic coordinates is taken from the introduction to the Hipparcos catalog.

REVISION HISTORY: Written, W. Landsman December 2000 fix the bug occurring if the input arrays are longer than 32767 and update the Sun velocity Sergey Koposov June 2008 vectorization of the loop – performance on large arrays is now 10 times higher Sergey Koposov December 2008
Example

```python
from __future__ import print_function
from PyAstronomy import pyasl

ra, dec = 017.4294147291650, +61.5473037149441
pmra, pmdec = 627.89, 77.84
d = 144
rv = -321.4

print("U, V, W \[kms/s\]: %5.3f, %5.3f, %5.3f \"
      print(pyasl.gal_uvw(ra, dec, pmra, pmdec, d, rv))
print("U, V, W (LSR corrected) \[kms/s\]: %5.3f, %5.3f, %5.3f\"
      print(pyasl.gal_uvw(ra, dec, pmra, pmdec, d, rv, lsr=(8.5, 13.38, 6.49)))
```

Transit and planet routines

Transit duration and contact point times

Contents

- Transit duration and contact point times
  - Transit and in- and egress duration for circular orbit
    - Example: Estimate duration of Earth’s transit and ingress (solar system units)
    - Example: Estimate transit and ingress duration of HD 189733 b (stellar units)
    - Function documentation
  - Estimate times of contact (T1-T4) for eccentric orbit
    - Example
    - Function documentation

Transit and in- and egress duration for circular orbit

Functions to estimate the duration of the transit and the in- and egress for circular orbits based on input in solar system units (AU, solar and Jovian radius)

- Transit duration (solar system units)
- Ingress duration (solar system units)

or parameters expressed in terms of the stellar radius

- Transit duration (stellar units)
- Ingress duration (stellar units)
Example: Estimate duration of Earth’s transit and ingress (solar system units)

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
from PyAstronomy import constants as pc

# Earth radius expressed in Jovian radii
reJ = pc.REarth / pc.RJ
print("Earth radius in Jovian units: ", reJ)

# Estimate the duration of Earth’s transit
td = pyasl.transitDuration(1.0, reJ, 1.0, 90.0, 365.0)
print("The transit of Earth lasts about: \$%.3f\$ days" % td)

# Estimate the duration of in- and egress
ti = pyasl.ingressDuration(1.0, reJ, 1.0, 90.0, 365.0)
print("The in- and egress last: \$%.4f\$ days or \$%.2f\$ hours" % (ti, ti*24))
```

Example: Estimate transit and ingress duration of HD 189733 b (stellar units)

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Semi-major axis in units of stellar radius
sma = 8.8
# Radius ratio (Rp/Rs)
rprs = 0.16

# Estimate the duration of Earth's transit
td = pyasl.transitDuration_Rs(sma, rprs, 85.7, 2.21858)
print("The transit of HD 189733 b lasts about: \$%.2f\$ hours" % (td*24.))

# Estimate the duration of in- and egress
ti = pyasl.ingressDuration_Rs(sma, rprs, 85.7, 2.21858)
print("The in- and egress of HD 189733 b lasts: \$%.2f\$ hours" % (ti*24.))
```

Function documentation

**PyAstronomy.pyasl.transitDuration** (sma, rp, rs, inc, period, exact=False)

Calculate the transit duration.

The routine calculates the transit duration assuming a spherical star and planet and a circular planetary orbit. It evaluates the expression

\[
T_D = \frac{P}{\pi} \arcsin \left( \frac{\sqrt{(R_s + R_p)^2 - b^2}}{a} \right)
\]

where \(P\) is the orbital period, \(b\) the impact parameter \((b = a \cos(i))\), and \(a\) the semi-major axis.

If the `exact` flag is set True, the slightly more accurate expression

\[
T_D = \frac{P}{\pi} \arcsin \left( \frac{\sqrt{(R_s + R_p)^2 - b^2}}{a \sin(i)} \right)
\]
is evaluated, where the additional sine term accounts for orbit curvature.

**Note:** The units of the transit duration are the same as the units of the input orbital period.

### Parameters

- `sma` [float] The semi-major axis in AU.
- `rp` [float] The planetary radius in Jovian radii.
- `rs` [float] The stellar radius in solar radii.
- `inc` [float] The orbital inclination in degrees.
- `period` [float] The orbital period.
- `exact` [boolean, optional] If True, a slightly more accurate expression is evaluated. The default is False.

### Returns

**Transit duration** [float] The duration of the transit (same units as `period`).

PyAstronomy.pyasl.**transitDuration_Rs**(sma, rprs, inc, period, exact=False)

Calculate transit duration

Invokes `transitDuration()` after parameter transformation.

### Parameters

- `sma` [float] Semi-major axis [stellar radii]
- `rprs` [float] Planet-to-star radius ratio (Rp/Rs)
- `inc` [float] Orbital inclination [deg]
- `period` [float] Orbital period
- `exact` [boolean, optional] If True, a slightly more accurate expression is evaluated. The default is False.

### Returns

**Transit duration** [float] The duration of the transit (same units as `period`).

PyAstronomy.pyasl.**ingressDuration**(sma, rp, rs, inc, period)

Calculate the ingress (and egress) duration of the transit.

The routine calculates the transit in- and egress duration assuming a spherical star and planet and a circular planetary orbit.

It evaluates the expression

\[
T_I = \frac{P}{2\pi} \left( \arcsin \left( \frac{\sqrt{(R_s + R_p)^2 - b^2}}{a \sin(i)} \right) - \arcsin \left( \frac{\sqrt{(R_s - R_p)^2 - b^2}}{a \sin(i)} \right) \right)
\]

is evaluated, where b is the impact parameter \((b = a \cos(i))\).

**Note:** The units of the ingress duration are the same as the units of the input orbital period.

### Parameters
sma [float] The semi-major axis in AU.

rp [float] The planetary radius in Jovian radii.

rs [float] The stellar radius in solar radii.

inc [float] The orbital inclination in degrees.

period [float] The orbital period.

Returns

Ingress duration [float] The duration of in- and egress (same units as period).

PyAstronomy.pyasl.\texttt{ingressDuration\_Rs}(sma, rprs, inc, period)

Calculate transit duration

Invokes \texttt{transitDuration()} after parameter transformation.

Parameters

sma [float] Semi-major axis [stellar radii]

rprs [float] Planet-to-star radius ratio (Rp/Rs)

inc [float] Orbital inclination [deg]

period [float] Orbital period

Returns

Transit duration [float] The duration of the transit (same units as period).

Estimate times of contact (T1-T4) for eccentric orbit

Numerically estimate contact points for primary and secondary eclipse.

Example

\begin{verbatim}
from __future__ import print_function
from PyAstronomy import pyasl

# SMA in stellar radii
sma = 5.67
# Rp/Rs
rprs = 0.15
# Orbital inclination
inc = 89.2
# Orbital period (time units are arbitrary but must be consistent)
p = 2.0
# Eccentricity
e = 0.63
# Argument of periastron (planetary orbit)
w = 155.
# Time of periastron passage
tau = 2412345.346

# Contact times for primary transit
pts = pyasl.transit_T1_T4_ell(sma, rprs, inc, p, tau, e, w, transit="p")
# Contact times for secondary transit
\end{verbatim}

(continues on next page)
sts = pyasl.transit_T1_T4_ell(sma, rprs, inc, p, tau, e, w, transit="s")

print("Transit times at arbitrary epoch (N*period may added)")
print("Primary transit T1-4: ", pts)
print("Secondary transit T1-4: ", sts)
print()  
print("Duration of primary and secondary transit: %5.3f, %5.3f")
→(pts[3]-pts[0], st[3]-st[0])

Function documentation

PyAstronomy.pyasl.transit_T1_T4_ell (sma, rprs, inc, period, tau, e, w, transit='p')
Calculate first to fourth contact times for elliptical orbit.

Parameters

    sma [float] Semi-major axis [stellar radii]
    rprs [float] Planet-to-star radius ratio (Rp/Rs)
    inc [float] Orbital inclination [deg]
    period [float] Orbital period
    tau [float] Time of periastron passage
    e [float] Eccentricity
    w [float] Argument of periastron

    transit [string, {p, s}, optional] Determines whether contact point times for primary (p, default) or secondary (s) transit are to be calculated.

Returns

    T1-T4 [tuple of floats] Contact times (arbitrary epoch). Note that contact time may not exist in the case of grazing transits or no transit at all. None is returned for undefined contact times.

Check whether time falls in transit window

PyAstronomy.pyasl.isInTransit (time, T0, period, halfDuration, boolOutput=False, secin=False)
Check whether time is inclosed by transit interval.

This function uses the given ephemerides (T0, period, and halfDuration) to check whether the time point(s) specified by time are within a transit window or not. The edges of the window are counted as transit times.

Parameters

    time [float or array like] The time(s) which are to be checked.
    T0 [float] The time reference point (center of transit).
    period [float] The orbital period.
    halfDuration [float] The half-duration of the event. Must have same units as time.
**boolOutput** [boolean, optional] If set True and *time* is an array, the function will return a boolean array holding True for time points in- and False for time points out-of-transit.

**secin** [boolean, optional] If True, also points associated with the secondary transit (around phase 0.5) will be counted as falling into the transit window. Default is False.

**Returns**

**inTransit** [boolean or array of int] If *time* was a float, the return value is a boolean, which is True if the give time falls into a transit interval and False otherwise. If *time* was given as an array, the return value is an array holding the indices of those time points, which fall into a transit window. The **boolOutput** option may be used to obtain a boolean array holding True for in-transit points.

**Example: Check individual point in time**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

time = 2476357.756234
T0 = 2475123.01245
period = 3.4789112
duration = 2.2/24.0

if not pyasl.isInTransit(time, T0, period, duration/2.0):
    print("No")
else:
    print("Yes")
```

**Example: Checking a series of times**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import numpy as np

times = 2476357.756234 + np.linspace(0.0, 5.0, 300)
T0 = 2475123.01245
period = 3.4789112
duration = 2.2/24.0

print(pyasl.isInTransit(times, T0, period, duration/2.0))
print(pyasl.isInTransit(times, T0, period, duration/2.0, boolOutput=True))
```
Transit visibility

The function `transitTimes()` provides a table containing, among others, mid-transit times, altitude above horizon, and Moon distance and phase; i.e., the most relevant information to evaluate whether a transit is observable at a particular point.

Using `transitVisibilityPlot()`, the data can be plotted to obtain a visual impression of the visibility.

Example of usage

The following example demonstrates the use of the functions. Both code and output is shown.

Code

```python
from PyAstronomy import pyasl
import datetime as dt

# Get the data for WASP-7 from NEXA data base
nexa = pyasl.NasaExoplanetArchive()
dat = nexa.selectByPlanetName("Wasp-7 b")

# Convert July 14th, 2018, 10pm to a Julian date
d = dt.datetime(2018, 7, 14, 22)
jd = pyasl.jdcnv(d)

# Calculate transit data for transits within 100 days starting from July 14th, 2018.
dat = pyasl.transitTimes(jd, jd+100., dat, nexaInput=True, observatory="esoparanal", obsOffset=1./24., minAltitude=10.0)

# Plot the result
pyasl.transitVisibilityPlot(dat, markTransit=True)
```

Text output

Specified time span
Start date (DDDD-MM-YY and fractional hours): 2018-07-14 21.962
End date (DDDD-MM-YY and fractional hours): 2018-10-22 22.129

Transit duration: 226.512 minutes
Off-transit time before and after transit: 60.0 minutes

<table>
<thead>
<tr>
<th>No. Tmid [HJD]</th>
<th>Obs. start [UT] [ALT, DIR(AZI)]</th>
<th>Transit mid [UT] [ALT, DIR(AZI)]</th>
<th>twink</th>
<th>moon distance</th>
<th>moon phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2458394.66052 3.10.0:57 [75°,S(193°)]</td>
<td>3.10.3:51 [47°,W(233°)]</td>
<td>night (-30°, -59°, -46°)</td>
<td>(153°, 154°, 155°) 42%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2458399.61518 7.10.23:52 [74°,S(159°)]</td>
<td>8.10.2:45 [56°,W(235°)]</td>
<td>astron. twilight (-15°, -50°, -55°)</td>
<td>(122°, 121°, 119°) 1%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2458404.56984 12.10.22:47 [68°,S(137°)]</td>
<td>13.10.1:40 [64°,W(229°)]</td>
<td>civil twilight (0°, -37°, -57°)</td>
<td>(60°, 58°, 57°) 18%</td>
<td></td>
</tr>
</tbody>
</table>
Graphical output

The figure shows the product of the `transitVisibilityPlot` function. In this case, four transits are shown in the form of altitude (or airmass) as a function of UT and local time. The numbers indicate to which transit the individual lines refer. The color of the thin lines indicates the light conditions (e.g., twilight or astronomical night). The green overlay indicates the time of transit; here, we have one hour of additional time before and after the transit, which was specified on the call to `transitTimes`. On the right, you see the dates at which the observation (not necessarily the transit) starts.

| Target coordinates: (311.0426°, -39.2253°) |
| Obs coord.: (289.5988°, -24.6259°, 2635 m) |

![Transit visibility of WASP-7 b](image)
API

**PyAstronomy.pyasl.transitTimes** *(tmin, tmax, planetData, obsOffset=0.0, observatory=None, lon=None, lat=None, alt=None, minAltitude=None, showTwilight='all', moonDist=None, nexaInput=False, fileOutput=None)*

Calculate transit times for a given planet and a given period of time.

The *planetData* dictionary must contain the following information:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>Right ascension of object [deg]</td>
</tr>
<tr>
<td>dec</td>
<td>Declination of object [deg]</td>
</tr>
<tr>
<td>T0</td>
<td>Time reference point (HJD)</td>
</tr>
<tr>
<td>orbPer</td>
<td>Orbital period [d]</td>
</tr>
<tr>
<td>orbInc</td>
<td>Orbital inclination [deg]</td>
</tr>
<tr>
<td>SMA</td>
<td>Semi-major axis [AU]</td>
</tr>
<tr>
<td>RpJ</td>
<td>Planetary radius [Jovian radii]</td>
</tr>
<tr>
<td>RsSun</td>
<td>Stellar Radius [solar]</td>
</tr>
<tr>
<td>Tdur</td>
<td>OPTIONAL, Transit duration [d]</td>
</tr>
</tbody>
</table>

If the transit duration (Tdur) is not given, the duration will be estimated using pyasl's *transitDuration* function.

**Note:** The input times *(tmin and tmax)* are expected in JD (UT). Input time will be calculated to HJD. Time output is in HJD.

**Parameters**

- **tmin** [float] Start of time interval in Julian days (UT).
- **tmax** [float] End of time interval in Julian days (UT).
- **planetData**: dictionary A dictionary containing the parameters of the exoplanet for which the transit times should be calculated. The required keys are specified above.
- **obs_offset** [float, optional] Specifies additional time before AND after the transit in DAYS. This is useful if the observation should start and end some time before and after the actual transit.
- **hjd** [boolean, optional] If True (default), the given Julian dates specifying the time interval *(tmin and tmax)* are automatically converted into the heliocentric frame (HJD).
- **observatory** [string, optional] If given, pyasl’s *observatory* function will be used to automatically resolve the name and obtain longitude, latitude, and altitude of the observatory. If *observatory* is given, *lon*, *lat*, and *alt* must not be specified.
- **lon** [float, optional] East longitude of the observatory given in DEGREES. Longitude is positive in EASTWARD direction. If LON is not given, transitTimes will only return beginning and end of the observation and the time of mid transit.
- **lat** [float, optional] Latitude of the observatory given in DEGREES (positive in NORTHWARD direction).
- **alt** [float, optional] Altitude of the observatory given in METER.
**minAltitude** [float, optional] Minimum altitude of the object in DEGREES.

If a minimum altitude is given, only transits for which the object is above the given altitude during the ENTIRE observation are included in the list created by `transitTimes`. Note that `minAltitude` can only be used if the observer’s location has been specified either via `observatory` or via `lon`, `lat`, and `alt`.

**showTwilight** [string, optional, {“all”, “civil”, “nautical”, “astronomical”, “night”}] Specifies the twilight acceptable during the observation. By default all twilight conditions are acceptable.

Only the transits for which the ENTIRE observation occurs during the specified or darker twilight conditions are listed.

The choices are:

- “all”: all transits are shown (even during day)
- “civil”: only transits during civil twilight and better are shown
- “nautical”: only transits during nautical twilight and better are shown
- “astronomical”: only transits during astronomical twilight and better are shown
- “night”: only transits during night are shown

Note that this can only have an effect, if the observer’s location is specified.

**moonDist** [float] Minimum distance between the Moon and the target in DEGREES. By default all Moon distances are acceptable (moonDist=0.0).

Only observations are listed for which the angular distance between the Moon and the target is larger than `moonDist` during the ENTIRE observation.

Note that this can only have an effect, if the observer’s location is specified.

**fileOutput** [string or file, optional] If a string is given, a file with the name will be created and the output will be written to that file. If a (writable) file object is given, the output will be written to that file. In both cases, no output will be given on screen.

**Returns**

**Transit times** [dictionary] Returns a dictionary containing the transit details. The dictionary key is a running number (starting with one), which is equivalent to that listed in the first column of the table.

For each transit, the function returns a dictionary with the transit details.

If the observer’s location was not specified, the dictionary has the following keys:
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planet name</td>
<td>Name of the planet</td>
</tr>
<tr>
<td>Tmid</td>
<td>HJD of transit center</td>
</tr>
<tr>
<td>Transit jd</td>
<td>Array giving JD of start, mid-time, and end of transit.</td>
</tr>
<tr>
<td>Obs jd</td>
<td>Array specifying the HJD of the start, center and end of the observation.</td>
</tr>
<tr>
<td>Obs cal</td>
<td>Equivalent to ‘Obs jd’, but in the form of the calendar date. In particular, for each date, a list containing [Year, month, day, fractional hours] is given. <strong>Below follows optional output only present if the observer’s location is known</strong></td>
</tr>
<tr>
<td>Obs coord</td>
<td>East longitude [deg], latitude [deg], and altitude [m] of the observatory.</td>
</tr>
<tr>
<td>Sun ra</td>
<td>Right ascension of the Sun at center of observation.</td>
</tr>
<tr>
<td>Sun dec</td>
<td>Declination of the Sun at center of observation.</td>
</tr>
<tr>
<td>Sun alt</td>
<td>Altitude of the Sun [deg] at begin, center, and end of the observation.</td>
</tr>
<tr>
<td>Sun az</td>
<td>Azimuth if the Sun [deg] at begin, center, and end of the observation.</td>
</tr>
<tr>
<td>Moon phase</td>
<td>Array giving lunar phase (in percent) at start, center, and end of the observation.</td>
</tr>
<tr>
<td>Moon AD</td>
<td>Angular distance between the target and the Moon at begin, center, and end of the observation [deg].</td>
</tr>
<tr>
<td>Moon ra</td>
<td>Right ascension of the Moon at begin, center, and end of the observation [deg].</td>
</tr>
<tr>
<td>Moon dec</td>
<td>Declination of the Moon at begin, center, and end of the observation [deg].</td>
</tr>
<tr>
<td>Star ra</td>
<td>Right ascension of the star [deg].</td>
</tr>
<tr>
<td>Star dec</td>
<td>Declination of the star [deg].</td>
</tr>
<tr>
<td>Star CP</td>
<td>Cardinal point of the star at begin, center, and end of the observation.</td>
</tr>
<tr>
<td>Star alt</td>
<td>Altitude of the star [deg] at begin, center, and end of the observation.</td>
</tr>
<tr>
<td>Star az</td>
<td>Azimuth of the star [deg] at begin, center, and end of the observation.</td>
</tr>
<tr>
<td>Twilight</td>
<td>The worst, i.e., brightest type of twilight encountered during the observation.</td>
</tr>
</tbody>
</table>

PyAstronomy.pyasl.transitVisibilityPlot(*allData*, *markTransit=False*, *plotLegend=True*, *showMoonDist=True*, *print2file=False*)

Plot the visibility of transits.

This function can conveniently be used with the output of the transitTimes function.

**Parameters**
**allData** [dictionary] Essentially the output of `transitTimes`. A dictionary mapping consecutive numbers (one per transit) to another dictionary providing the following keys:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planet name</td>
<td>Name of the planet</td>
</tr>
<tr>
<td>Transit jd</td>
<td>(Only if ‘markTransit is True) Array giving JD of start, mid-time, and end of transit.</td>
</tr>
<tr>
<td>Obs jd</td>
<td>Array specifying the HJD of the start, center and end of the observation.</td>
</tr>
<tr>
<td>Obs cal</td>
<td>Equivalent to ‘Obs jd’, but in the form of the calendar date. In particular, for each date, a list containing [Year, month, day, fractional hours] is given.</td>
</tr>
<tr>
<td>Obs coord</td>
<td>East longitude [deg], latitude [deg], and altitude [m] of the observatory.</td>
</tr>
<tr>
<td>Star ra</td>
<td>Right ascension of the star [deg].</td>
</tr>
<tr>
<td>Star dec</td>
<td>Declination of the star [deg].</td>
</tr>
</tbody>
</table>

**Note:** To use the list created by `transitTimes`, the LONGITUDE and LATITUDE of the observatory location must have been specified.

**markTransit** [boolean, optional] If True (default is False), the in-transit times will be clearly indicated in the plot. Note that this would not be the case otherwise, which is particularly important if extra off-transit time before and after the transit has been requested.

**showMoonDist** [boolean, optional] If True (default), the Moon distance will be shown.

**print2file** [boolean or string, optional] If True, the plot will be dumped to a png-file named: “transitVis-“[planetName].png. The default is False. If a string is given, it specifies the name of the output file.

### Phases a planet

**Phase function for the Lambert sphere**

PyAstronomy.pyasl.\_lambertPhaseFunction(\_alpha)  
Calculate phase function for a Lambert sphere.

The phase function of the Lambert sphere is given by:

\[
\Phi(\alpha) = \frac{\sin(\alpha) + (\pi - \alpha) \cos(\alpha)}{\pi}.
\]

Here, \( \alpha \) is the phase angle, which is defined as the angle between the star and the Earth as seen from the planet. Hence, at phase angle zero, the planet is found in opposition. Formally, the phase angle can be between 0 and 180 degrees. This function accounts for cases in which the given phase angle violates these limits by projecting it back into the valid range.

**Parameters**
alpha  [float or array] The phase angle(s) [deg] at which the phase function is to be calculated.

Returns

Phase function  [float or array] The values of the phase function at the input phase angles.

Statistics

F-test

PyAstronomy.pyasl.ftest(chi1, chi2, dof1, dof2, compute_ratio_of_variance=False)
Performs an F-test.

Parameters

chi1  [float] The chi-squared statistic or variance for model 1.
chi2  [float] The chi-squared statistic or variance for model 2. Should be better (lower) than chi1.
dof1  [integer] The number of degrees of freedom of model 1
dof2  [integer] The number of degrees of freedom of model 2. Should be lower than dof1.
compute_ratio_of_variance  [boolean, optional] Determines how the F-statistics is computed. If False (default), the “ANOVA F-test in regression analysis of nested nonlinear models” is applied. If True, “Fisher’s classical test for the equality of two variances” is used.

Returns

Test results  [dictionary]

• “F statistic” - Value of F statistic.
• “p-value” - Probability to obtain an F-value as large as or larger than observed assuming H_0 to be true
• “Gaussian sigma level” - Same probability converted to Gaussian standard deviations (“sigmas”)

Notes

The “method of least squares” is widely used in parameter estimation. Much of its appeal lies in its simplicity: Minimizing the sum of squared differences between the model and the data.

The backbone of least squares is the classical multiple regression analysis using linear models to relate several independent variables to a dependent variable. However, many physical models are nonlinear; this is the realm of nonlinear least squares regression methods.

The aim of model fitting is to explain as much of the variation in the dependent variable as possible from information contained in the independent variables. The contributions of the independent variables to the model are measured by partitions of the total sum of squares of the difference of data and model (“analysis of variance”, ANOVA).

ANOVA F-test in regression analysis for nested nonlinear models

The sum of squares for any hypothesis can be determined from the difference between the residual sums of squares (RSS) of two models: The so-called “full” model, in the context of which the null hypothesis is to be tested, and the “reduced” model, which is derived from the full model by imposing additional constraints specified by the null hypothesis; setting one parameter to zero is a common example. The reduced model is a
special case of the full model and, hence, its residual sum of squares must not be lower than the residual sum of squares for the full model.

Accounting for \( k \) independent constraints on the full model and \( N \) data points, we compute

\[
Q = \text{RSS}(\text{reduced}) - \text{RSS}(\text{full})
\]

which has \( k \) degrees of freedom. Here, \( k \) corresponds to the difference in the number of parameters—or equivalently, the difference in the degrees of freedoms—between the full and reduced model. The \( F \)-test of the null hypothesis can then be computed as

\[
\hat{F} = \left( \frac{Q}{k} \right) / s^2
\]

where \( s^2 \) is an unbiased estimate of \( \sigma^2 \), e.g., derived from the full model. In the case of error-weighted data, the \( F \) statistic reads

\[
\hat{F} = \frac{\left( \chi^2_{\text{reduced}} - \chi^2_{\text{full}} \right)/\left( \nu_{\text{reduced}} - \nu_{\text{full}} \right)}{\chi^2_{\text{full}}/\nu_{\text{full}}}
\]

where \( \nu \) denotes the number of degrees of freedom, which may be calculated as \( N - p - 1 \) given a model with \( p \) parameters and an additional constant. The expectation value of \( \hat{F} \) is 1, thus, if \( \hat{F} = 1 \) there is no significant difference between the RSS of both models. If, however, \( \hat{F} \) deviates from 1, we can compute the probability for \( \hat{F} \) to equal or exceed the obtained value \( \hat{F} \) by

\[
\text{Prob}(F \geq \hat{F}) = CDF(F(\nu_{\text{reduced}} - \nu_{\text{full}}, \nu_{\text{full}}))
\]

For details on the principles of the formalism, see Sect. 4.5.4 in Rawlings’ “Applied Regression Analysis”.

In case of nonlinear models, the distribution of the least-square parameter estimates only approximately follows the normal distribution. Then, the \( F \) statistic (in this case called “Wald statistic”) also holds approximately. The validity of the method depends on how well the model is represented by a linear approximation in the parameters (“parameter effects curvature”). For details, we refer to Sect. 15.3 in Rawlings’ “Applied Regression Analysis” and references therein.

**Fisher’s classical test for the equality of two variances**

If the two competing models cannot be treated as nested, in the sense that the full model encompasses the reduced model, we can directly compare the model variances with an F-test.

Let \( s_1 \) and \( s_2 \) denote unbiased estimates of the variances of two independent, normally distributed populations of size \( N_1 \) and \( N_2 \), respectively. Under the null hypothesis both variances are equal:

\[
H_0 : s_1^2 = s_2^2, \quad H_1 : s_1^2 > s_2^2
\]

Then the quantity

\[
\hat{F} = \frac{s_1^2}{s_2^2}
\]

is \( F \)-distributed with \( N_1 - 1 \) and \( N_2 - 1 \) degrees of freedom. In the case of weighted errors we have

\[
\hat{F} = \frac{\chi^2_1/\nu_1}{\chi^2_2/\nu_2}
\]

and \( \hat{F} \) is distributed according to an \( F \)-distribution with \( \nu_1 \) and \( \nu_2 \) degrees of freedom.

**References**

Exponentially correlated Gaussian random numbers

\texttt{PyAstronomy.pyasl.expCorrRN} \texttt{(n, \ tau, \ mean=0.0, \ std=1.0, \ rnos=\text{None}, \ fullOut=False)}

Generate exponentially correlated random numbers.

This procedure implements the prescription given by Deserno 2002 (“How to generate exponentially correlated Gaussian random numbers”). The autocorrelation function of the resulting numbers decays with the predefined “decay time”, \(\tau\). The correlation coefficient of the resulting numbers is given by \(\exp(-1/\tau)\).

**Parameters**

- \texttt{n} [\text{int}] Number of numbers to be generated.
- \texttt{tau} [\text{float}] Decay time
- \texttt{mean} [\text{float, optional}] Mean of the numbers to be generated. Default is 0.0.
- \texttt{std} [\text{float, optional}] Standard deviation of the generated numbers. Default is 1.0.
- \texttt{rnos} [\text{array, optional}] Uncorrelated Gaussian random numbers with mean 0.0 and standard deviation 1.0 to be used to generate correlated random numbers. If not given, Gaussian random numbers will be obtained using \texttt{numpy.random.normal}.
- \texttt{fullOut} [\text{boolean, optional}] If False (default), only the correlated random numbers will be returned.

**Returns**

- \texttt{Correlated RNs} [\text{array}] Correlated Gaussian random numbers.
- \texttt{Uncorrelated RNs} [\text{array, optional}] The uncorrelated random numbers used to generate the correlated numbers (only of \texttt{fullOut} is True).
- \texttt{Correlation coefficient} [\text{float}] The correlation coefficient (\(\exp(-1/\tau)\), only of \texttt{fullOut} is True).

**Example: Generating correlated numbers**

```python
import matplotlib.pyplot as plt
from PyAstronomy import pyasl

# Generate 200 exponentially correlated Gaussian random numbers with a decay time of 5
c1 = pyasl.expCorrRN(200, 5)

# Generate 200 exponentially correlated Gaussian random numbers with decay time 10, mean 4, and standard deviation of 2.3.
# The results are: The correlated random numbers, the uncorrelated numbers used as input, and the correlated coefficient (\exp(-1/\tau)).
c2, g, f = pyasl.expCorrRN(200, 10, mean=4.0, std=2.3, fullOut=True)

plt.subplot(2, 1, 1)
plt.plot(range(200), c1, 'bp-')
plt.subplot(2, 1, 2)
plt.plot(range(200), c2, 'bp-')
plt.plot(range(200), g, 'g.')
plt.show()
```
Example: Exploring the autocorrelation function

```python
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyasl

# Generate n exponentially correlated Gaussian random numbers with a decay time, tau
n = 500
tau = 5.
c1 = pyasl.expCorrRN(n, tau)

# Obtain autocorrelation function
ac = np.correlate(c1, c1, mode="full")[n-1:]

# Plot correlated random numbers and autocorrelation function along with exponential model.
x = np.arange(float(n))
plt.subplot(2, 1, 1)
plt.plot(x, c1, 'bp-')
plt.subplot(2, 1, 2)
plt.plot(x, ac, 'b.')
plt.plot(x, np.exp(-x/tau)*ac.max(), 'r--')
plt.show()
```

Resource based helpers

Evolutionary tracks (Baraffe et al. 98)

```python
class PyAstronomy.pyasl.Baraffe98Tracks
    Provide access to the evolutionary tracks of Baraffe et al. 98.
    Downloads the table (tab1-3.dat) pertaining to the paper Baraffe et al. 98, A&A 337, 403 and offers some helper method to access the individual tracks.

Methods

```
findModels([Met, Y, Lmix, Mass])  Find models with given parameters.
getModelData(model[, columns])  Find data for a specific model.
getUniqueValues(colName)  Find unique values in a column.
```

```
findModels (Met=None, Y=None, Lmix=None, Mass=None)  Find models with given parameters.
```

Parameters

- **Met** [float, optional] The metallicity.
- **Y** [float, optional] The initial helium mass fraction.
- **Lmix** [float, optional] The initial mixing length parameter.
- **Mass** [float, optional] The mass [solar masses]

Returns
Models  [list of tuples] A list of models specified as tuples defining: metallicity, Y, Lmix, and mass.

getModelData  (model, columns=None)
Find data for a specific model.

Parameters

model  [tuple of float] Specifies the model as a tuple containing four entries, which define: metallicity, initial helium mass fraction (Y), initial mixing length parameter (Lmix), and mass.

columns  [list of strings, optional] A list of column names, which shall be included in the result. The default is including all columns.

Returns

Model  [recarray] The data pertaining to that particular model as a numpy recarray.

getUniqueValues  (colName)
Find unique values in a column.

Parameters

colName  [string, {"Met","Y","Lmix","Mass","Age","Teff","logg","Mbol","MV","MR","MI","MJ","MH","MK"}]
The name of the column.

Returns

Unique values  [array] All unique values in that column.

Example of usage

```python
from __future__ import print_function, division
from PyAstronomy.pyasl import resBased as rb
import matplotlib.pyplot as plt
import numpy as np

bt = rb.BaBaffe98Tracks()

print("Unique metallicity values: ", bt.getUniqueValues("Met"))
print("Unique Y values: ", bt.getUniqueValues("Y"))
print("Unique Lmix values: ", bt.getUniqueValues("Lmix"))
print("Unique mass values: ", bt.getUniqueValues("Mass"))

# Get model data and plot log10(age) versus effective temperature
m = bt.getModelData((0.0, 0.275, 1.0, 0.3))
plt.plot(np.log10(m.Age*1e9), m.Teff, 'b.-')

# Find all models with metallicity 0.0, Y 0.275, Lmix 1.0, and any mass.
models = bt.findModels(Met=0.0, Mass=None, Y=0.275, Lmix=1.0)
# Print out the model parameters.
print()
print("Number of models found: ", len(models))
for i, model in enumerate(models):
    print("Model no. %3d: Met = %3.1f, Y = %5.3f, Lmix = %3.1f, Mass = %4.2f" % ((i+1,) + model))
```
Access the “NASA Exoplanet Archive”

```python
# Finally, show the plot
plt.show()
```

**class PyAstronomy.pyasl.NasaExoplanetArchive**

Easy access to NASA’s exoplanet archive.

This class downloads a table of exoplanet data from the “NASA Exoplanet Archive” (http://exoplanetarchive.ipac.caltech.edu/index.html) and provides access to these data. By default, the data will be re-downloaded every seven days.

The following data are provided:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>pl_hostname</td>
<td>Name of host star</td>
<td></td>
</tr>
<tr>
<td>pl_name</td>
<td>Name of the planet</td>
<td></td>
</tr>
<tr>
<td>pl_letter</td>
<td>Planet letter (e.g., b, c, d, etc.)</td>
<td></td>
</tr>
<tr>
<td>ra</td>
<td>Right ascension</td>
<td>deg</td>
</tr>
<tr>
<td>dec</td>
<td>Declination</td>
<td>deg</td>
</tr>
<tr>
<td>pl_orbper</td>
<td>Planetary orbital period</td>
<td>d</td>
</tr>
<tr>
<td>pl_massj</td>
<td>Planetary mass</td>
<td>MJ</td>
</tr>
<tr>
<td>pl_radj</td>
<td>Planetary radius</td>
<td>RJ</td>
</tr>
<tr>
<td>pl_trandep</td>
<td>Central depth of transit</td>
<td>%</td>
</tr>
<tr>
<td>pl_trandur</td>
<td>Transit duration</td>
<td>d</td>
</tr>
<tr>
<td>pl_tranmid</td>
<td>Transit midpoint</td>
<td>BJD</td>
</tr>
<tr>
<td>pl_orbsmax</td>
<td>Semi-major-axis</td>
<td>AU</td>
</tr>
<tr>
<td>pl_orbincl</td>
<td>Orbital inclination of planet</td>
<td>deg</td>
</tr>
<tr>
<td>st_rad</td>
<td>Stellar radii</td>
<td>Solar</td>
</tr>
<tr>
<td>st_dist</td>
<td>Distance to star</td>
<td>pc</td>
</tr>
<tr>
<td>st_mass</td>
<td>Stellar mass</td>
<td>Solar</td>
</tr>
<tr>
<td>st_teff</td>
<td>Effective temperature of star</td>
<td>K</td>
</tr>
<tr>
<td>st_vsin</td>
<td>Stellar vsin(i)</td>
<td>km/s</td>
</tr>
<tr>
<td>st_logg</td>
<td>Stellar surface gravity</td>
<td>cm/s**2</td>
</tr>
<tr>
<td>st_acts</td>
<td>Stellar S-Index</td>
<td></td>
</tr>
<tr>
<td>st_vj</td>
<td>Stellar V-band brightness</td>
<td>mag</td>
</tr>
</tbody>
</table>

**Methods**

- `availableColumns([verbose])`: Shows a list of available data columns.
- `changeDownloadCycle(c)`: Change the time after which the data are updated.
- `dataAge()`: Determine the “age” of the data.
- `downloadData()`: Trigger download of data.
- `getAllData()`: Get all available data.
- `needsUpdate()`: Determine whether data need to be updated.
- `selectByPlanetName(planetName[, caseSensitive])`: Get entry by planet name.
availableColumns (verbose=True)
  Shows a list of available data columns.

  Parameters
  verbose [boolean, optional] If True (default), prints information to screen.

  Returns
  columns [list of strings] The names of the available data columns.

changeDownloadCycle (c)
  Change the time after which the data are updated.

  By default, the data will be updated if they are older than the given update cycle. This method allows you
  to change that cycle.

  Parameters
  c [float or None] The new update cycle in days. If None is provided, updating is switched
  off.

dataAge ()
  Determine the “age” of the data.

  Returns
  age [float] The time since last data update in days. None, if no age can be determined,
  e.g., if data have never been downloaded.

downloadData ()
  Trigger download of data.

egetAllData ()
  Get all available data.

  Returns
  Data [numpy recarray] All data stored in the table as a numpy recarray.

needsUpdate ()
  Determine whether data need to be updated.

  Returns
  Update flag [boolean] True if data need update and False otherwise.

selectByPlanetName (planetName, caseSensitive=False)
  Get entry by planet name.

  Parameters
  planetName [string] The name of the planet (includes planet letter, e.g., “corot-2 b”
  caseSensitive [boolean, optional] If False (default), the search will be case-insensitive.

  Returns
  Data entry [dictionary] A dictionary with a key for every data column holding the asso-
  ciated value from the data table.
Example of usage

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import matplotlib.pyplot as plt

nexa = pyasl.NasaExoplanetArchive()

# See what information is available
cols = nexa.availableColumns()
print()

# Get all information for planet 'wasp-12 b'
# By default, the search is case-insensitive
print("Entry of Wasp-12 b")
print(nexa.selectByPlanetName("Wasp-12 b"))

print()
# Get all data and plot ra vs. dec
dat = nexa.getAllData()
plt.plot(dat.ra, dat.dec, 'b. ')
plt.show()
```

Access the exoplanet.eu data base

PyA provides the **ExoplanetEU2()** to access the data provided by exoplanet.eu. The class download the data as a Virtual Observatory (VO) table to provide access to it.

**Note:** PyA also provides the **ExoplanetEU()** (note the missing ‘2’), which is the predecessor of the above implementation. Although functional, this implementation should be considered deprecated.

ExoplanetEU2

**Example: Usage of ExoplanetEU2**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import matplotlib.pyplot as plt

# Instantiate exoplanetEU2 object
v = pyasl.ExoplanetEU2()

# Show the available data
v.showAvailableData()
print()

# Get a list of all available column names
acs = v.getColnames()
print("Available column names: " + , ".join(acs)
print()
```

(continues on next page)
# Select data by planet name (returns a dictionary)
print(v.selectByPlanetName("CoRoT-2 b"))
print()

# Get all data as an astropy table
at = v.getAllDataAPT()

# Export all data as a pandas DataFrame
pd = v.getAllDataPandas()

# Plot mass vs. SMA
plt.title("Mass vs. SMA")
plt.xlabel("[" + v.getUnitOf("mass") + "]")
plt.ylabel("[" + v.getUnitOf("semi_major_axis") + "])")
plt.loglog(at["mass"], at["semi_major_axis"], 'b.'
plt.show()

## API documentation (ExoplanetEU2)

**class PyAstronomy.pyasl.ExoplanetEU2** *(skipUpdate=False, forceUpdate=False)*

Provides access to exoplanet.eu data base.

This class downloads the data base as a VO table. By default, the data are re-downloaded every 7 days.

The class provides access to the entire data base offered by exoplanet.eu as shown below. Use *showAvailableData* to view the table of available data, including potential updates.

<table>
<thead>
<tr>
<th>name</th>
<th>dtype</th>
<th>unit</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>object</td>
<td></td>
<td>Name of a planet</td>
</tr>
<tr>
<td>mass</td>
<td>float64</td>
<td>jovMass</td>
<td>Planetary Mass</td>
</tr>
<tr>
<td>mass_err_min</td>
<td>float64</td>
<td>jovMass</td>
<td>Planetary Mass error</td>
</tr>
<tr>
<td>mass_err_max</td>
<td>float64</td>
<td>jovMass</td>
<td>Planetary Mass error</td>
</tr>
<tr>
<td>mass_sini</td>
<td>float64</td>
<td>jovMass</td>
<td>Planetary Mass*sin(i)</td>
</tr>
<tr>
<td>mass_sini_error_min</td>
<td>float64</td>
<td>jovMass</td>
<td>Planetary Mass*sin(i) error</td>
</tr>
<tr>
<td>mass_sini_error_max</td>
<td>float64</td>
<td>jovMass</td>
<td>Planetary Mass*sin(i) error</td>
</tr>
<tr>
<td>radius</td>
<td>float64</td>
<td>Rjup</td>
<td>Planetary Radius</td>
</tr>
<tr>
<td>radius_error_min</td>
<td>float64</td>
<td>Rjup</td>
<td>Planetary Radius error</td>
</tr>
<tr>
<td>radius_error_max</td>
<td>float64</td>
<td>Rjup</td>
<td>Planetary Radius error</td>
</tr>
<tr>
<td>orbital_period</td>
<td>float64</td>
<td>d</td>
<td>Orbital Period</td>
</tr>
<tr>
<td>orbital_period_err_min</td>
<td>float64</td>
<td>d</td>
<td>Orbital Period error</td>
</tr>
<tr>
<td>orbital_period_err_max</td>
<td>float64</td>
<td>d</td>
<td>Orbital Period error</td>
</tr>
<tr>
<td>semi_major_axis</td>
<td>float64</td>
<td>AU</td>
<td>Semi-Major Axis</td>
</tr>
<tr>
<td>semi_major_axis_error_min</td>
<td>float64</td>
<td>AU</td>
<td>Semi-Major Axis error</td>
</tr>
<tr>
<td>semi_major_axis_error_max</td>
<td>float64</td>
<td>AU</td>
<td>Semi-Major Axis error</td>
</tr>
<tr>
<td>eccentricity</td>
<td>float64</td>
<td></td>
<td>Orbital Eccentricity</td>
</tr>
<tr>
<td>eccentricity_error_min</td>
<td>float64</td>
<td></td>
<td>Orbital Eccentricity error</td>
</tr>
<tr>
<td>eccentricity_error_max</td>
<td>float64</td>
<td></td>
<td>Orbital Eccentricity error</td>
</tr>
<tr>
<td>inclination</td>
<td>float64</td>
<td>deg</td>
<td>Orbital Inclination</td>
</tr>
<tr>
<td>inclination_error_min</td>
<td>float64</td>
<td>deg</td>
<td>Orbital Inclination error</td>
</tr>
<tr>
<td>inclination_error_max</td>
<td>float64</td>
<td>deg</td>
<td>Orbital Inclination error</td>
</tr>
<tr>
<td>angular_distance</td>
<td>float64</td>
<td>arcs</td>
<td>Angular Distance</td>
</tr>
<tr>
<td>name</td>
<td>dtype</td>
<td>unit</td>
<td>description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>discovered</td>
<td>int32</td>
<td>yr</td>
<td>Year of Discovery</td>
</tr>
<tr>
<td>updated</td>
<td>object</td>
<td></td>
<td>Last Update</td>
</tr>
<tr>
<td>omega</td>
<td>float64</td>
<td>deg</td>
<td>Argument of Periastron</td>
</tr>
<tr>
<td>omega_error_min</td>
<td>float64</td>
<td>deg</td>
<td>Argument of Periastron error</td>
</tr>
<tr>
<td>omega_error_max</td>
<td>float64</td>
<td>deg</td>
<td>Argument of Periastron error</td>
</tr>
<tr>
<td>tperi</td>
<td>float64</td>
<td>d</td>
<td>Epoch of Periastron</td>
</tr>
<tr>
<td>tperi_error_min</td>
<td>float64</td>
<td>d</td>
<td>Epoch of Periastron error</td>
</tr>
<tr>
<td>tperi_error_max</td>
<td>float64</td>
<td>d</td>
<td>Epoch of Periastron error</td>
</tr>
<tr>
<td>tconj</td>
<td>float64</td>
<td>d</td>
<td>Conjonction Date</td>
</tr>
<tr>
<td>tconj_error_min</td>
<td>float64</td>
<td>d</td>
<td>Conjonction Date error</td>
</tr>
<tr>
<td>tconj_error_max</td>
<td>float64</td>
<td>d</td>
<td>Conjonction Date error</td>
</tr>
<tr>
<td>tzero_tr</td>
<td>float64</td>
<td>d</td>
<td>Primary Transit</td>
</tr>
<tr>
<td>tzero_tr_error_min</td>
<td>float64</td>
<td>d</td>
<td>Primary Transit error</td>
</tr>
<tr>
<td>tzero_tr_error_max</td>
<td>float64</td>
<td>d</td>
<td>Primary Transit error</td>
</tr>
<tr>
<td>tzero_tr_sec</td>
<td>float64</td>
<td>d</td>
<td>Secondary Transit</td>
</tr>
<tr>
<td>tzero_tr_sec_error_min</td>
<td>float64</td>
<td>d</td>
<td>Secondary transit error</td>
</tr>
<tr>
<td>tzero_tr_sec_error_max</td>
<td>float64</td>
<td>d</td>
<td>Secondary transit error</td>
</tr>
<tr>
<td>lambda_angle</td>
<td>float64</td>
<td>deg</td>
<td>Sky-projected angle between the planetary orbital spin and the stellar rotational spin</td>
</tr>
<tr>
<td>lambda_angle_error_min</td>
<td>float64</td>
<td>deg</td>
<td>Sky-projected angle between the planetary orbital spin and the stellar rotational spin error</td>
</tr>
<tr>
<td>lambda_angle_error_max</td>
<td>float64</td>
<td>deg</td>
<td>Sky-projected angle between the planetary orbital spin and the stellar rotational spin error</td>
</tr>
<tr>
<td>impact_parameter</td>
<td>float64</td>
<td></td>
<td>Impact Parameter b</td>
</tr>
<tr>
<td>impact_parameter_error_min</td>
<td>float64</td>
<td></td>
<td>Impact Parameter b error</td>
</tr>
<tr>
<td>impact_parameter_error_max</td>
<td>float64</td>
<td></td>
<td>Impact Parameter b error</td>
</tr>
<tr>
<td>tzero_vr</td>
<td>float64</td>
<td>d</td>
<td>Zero Radial Speed time</td>
</tr>
<tr>
<td>tzero_vr_error_min</td>
<td>float64</td>
<td>d</td>
<td>Zero Radial Speed time error</td>
</tr>
<tr>
<td>tzero_vr_error_max</td>
<td>float64</td>
<td>d</td>
<td>Zero Radial Speed time error</td>
</tr>
<tr>
<td>k</td>
<td>float64</td>
<td>m / s</td>
<td>Velocity Semiamplitude K</td>
</tr>
<tr>
<td>k_error_min</td>
<td>float64</td>
<td>m / s</td>
<td>Velocity Semiamplitude K error</td>
</tr>
<tr>
<td>k_error_max</td>
<td>float64</td>
<td>m / s</td>
<td>Velocity Semiamplitude K error</td>
</tr>
<tr>
<td>temp_calculated</td>
<td>float64</td>
<td>K</td>
<td>Calculated temperature</td>
</tr>
<tr>
<td>temp_measured</td>
<td>float64</td>
<td>K</td>
<td>Measured temperature</td>
</tr>
<tr>
<td>hot_point_lon</td>
<td>float64</td>
<td>deg</td>
<td>Hottest point longitude</td>
</tr>
<tr>
<td>geometric_albedo</td>
<td>float64</td>
<td></td>
<td>Geometric albedo</td>
</tr>
<tr>
<td>geometric_albedo_error_min</td>
<td>float64</td>
<td></td>
<td>Geometric albedo error</td>
</tr>
<tr>
<td>geometric_albedo_error_max</td>
<td>float64</td>
<td></td>
<td>Geometric albedo error</td>
</tr>
<tr>
<td>log_g</td>
<td>float64</td>
<td>log g</td>
<td>log(g)</td>
</tr>
<tr>
<td>publication_status</td>
<td>object</td>
<td></td>
<td>Publication Status</td>
</tr>
<tr>
<td>detection_type</td>
<td>object</td>
<td></td>
<td>Detection type</td>
</tr>
<tr>
<td>mass_detection_type</td>
<td>object</td>
<td></td>
<td>Mass Measurement Method</td>
</tr>
<tr>
<td>radius_detection_type</td>
<td>object</td>
<td></td>
<td>Radius Measurement Method</td>
</tr>
<tr>
<td>alternate_names</td>
<td>object</td>
<td></td>
<td>List of planet alternative names</td>
</tr>
<tr>
<td>molecules</td>
<td>object</td>
<td></td>
<td>List of detected molecules</td>
</tr>
<tr>
<td>star_name</td>
<td>object</td>
<td></td>
<td>Name of a host star</td>
</tr>
<tr>
<td>ra</td>
<td>float64</td>
<td>deg</td>
<td>RA (J2000) of a star</td>
</tr>
<tr>
<td>dec</td>
<td>float64</td>
<td>deg</td>
<td>Dec (J2000) of a star</td>
</tr>
<tr>
<td>mag_v</td>
<td>float64</td>
<td>mag</td>
<td>V magnitude of a host star</td>
</tr>
<tr>
<td>mag_i</td>
<td>float64</td>
<td>mag</td>
<td>I magnitude of a host star</td>
</tr>
<tr>
<td>mag_j</td>
<td>float64</td>
<td>mag</td>
<td>J magnitude of a host star</td>
</tr>
<tr>
<td>mag_h</td>
<td>float64</td>
<td>mag</td>
<td>H magnitude of a host star</td>
</tr>
</tbody>
</table>
### Table 17 – continued from previous page

<table>
<thead>
<tr>
<th>name</th>
<th>dtype</th>
<th>unit</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mag_k</td>
<td>float64</td>
<td>mag</td>
<td>K magnitude of a host star</td>
</tr>
<tr>
<td>star_distance</td>
<td>float64</td>
<td>pc</td>
<td>Distance to a host star</td>
</tr>
<tr>
<td>star_distance_error_min</td>
<td>float64</td>
<td>pc</td>
<td>Distance to a host star error</td>
</tr>
<tr>
<td>star_distance_error_max</td>
<td>float64</td>
<td>pc</td>
<td>Distance to a host star error</td>
</tr>
<tr>
<td>star_metallicity</td>
<td>float64</td>
<td></td>
<td>Metallicity of a host star</td>
</tr>
<tr>
<td>star_metallicity_error_min</td>
<td>float64</td>
<td></td>
<td>Metallicity of a host star error</td>
</tr>
<tr>
<td>star_metallicity_error_max</td>
<td>float64</td>
<td></td>
<td>Metallicity of a host star error</td>
</tr>
<tr>
<td>star_mass</td>
<td>float64</td>
<td>Msun</td>
<td>Mass of a host star</td>
</tr>
<tr>
<td>star_mass_error_min</td>
<td>float64</td>
<td>Msun</td>
<td>Mass of a host star error</td>
</tr>
<tr>
<td>star_mass_error_max</td>
<td>float64</td>
<td>Msun</td>
<td>Mass of a host star error</td>
</tr>
<tr>
<td>star_radius</td>
<td>float64</td>
<td>Rsun</td>
<td>Radius of a host star</td>
</tr>
<tr>
<td>star_radius_error_min</td>
<td>float64</td>
<td>Rsun</td>
<td>Radius of a host star error</td>
</tr>
<tr>
<td>star_radius_error_max</td>
<td>float64</td>
<td>Rsun</td>
<td>Radius of a host star error</td>
</tr>
<tr>
<td>star_sp_type</td>
<td>object</td>
<td></td>
<td>Spectral type of a host star</td>
</tr>
<tr>
<td>star_age</td>
<td>float64</td>
<td>Gyr</td>
<td>Age of a host star</td>
</tr>
<tr>
<td>star_age_error_min</td>
<td>float64</td>
<td>Gyr</td>
<td>Age of a host star error</td>
</tr>
<tr>
<td>star_age_error_max</td>
<td>float64</td>
<td>Gyr</td>
<td>Age of a host star error</td>
</tr>
<tr>
<td>star_teff</td>
<td>float64</td>
<td>K</td>
<td>Effective temperature of a host star</td>
</tr>
<tr>
<td>star_teff_error_min</td>
<td>float64</td>
<td>K</td>
<td>Effective temperature of a host star error</td>
</tr>
<tr>
<td>star_teff_error_max</td>
<td>float64</td>
<td>K</td>
<td>Effective temperature of a host star error</td>
</tr>
<tr>
<td>star_detected_disc</td>
<td>object</td>
<td></td>
<td>Star Detected Disc</td>
</tr>
<tr>
<td>star_magnetic_field</td>
<td>bool</td>
<td></td>
<td>Star magnetic field</td>
</tr>
<tr>
<td>star_alternate_names</td>
<td>object</td>
<td></td>
<td>List of star alternative names</td>
</tr>
</tbody>
</table>

### Parameters

- **skipUpdate** [Boolean, optional] If True, update of data will be skipped (default is False).
- **forceUpdate** [Boolean, optional] If True, update of data will be forced (independent of value if skipUpdate, default is False)

### Methods

- `changeDownloadCycle(c)` Change the time after which the data are updated.
- `dataAge()` Determine the “age” of the data.
- `forceUpdate()` Force a fresh download of the data and read the data.
- `getAllDataAPIT()` Get all data as astropy table object.
- `getAllDataPandas()` Get all data as pandas DataFrame.
- `getColnames()` Get all column names.
- `getUnitOf(col)` Get unit of column.
- `needsUpdate()` Determine whether data need to be updated.
- `selectByPlanetName(planetName[, toScreen, ...])` Get entry by planet name.
- `showAvailableData()` Show available data.

### changeDownloadCycle(c)

Change the time after which the data are updated.

By default, the data will be updated if they are older than the given update cycle. This method allows you to change that cycle.

---

2.1. Python AstroLib 169
Parameters

c  [float or None] The new update cycle in days. If None is provided, updating is switched off.

dataAge()
  Determine the “age” of the data.

  Returns
  age  [float] The time since last data update in days. None, if no age can be determined, e.g., if data have never been downloaded.

forceUpdate()
  Force a fresh download of the data and read the data.
  By default, the data will be updated every 7 days.

getAllDataAPT()
  Get all data as astropy table object.

  Returns
  votable  [astropy.table.Table] All tabulated data as an astropy table

ggetAllDataPandas()
  Get all data as pandas DataFrame.

  Returns
  table  [DataFrame] All available data in pandas format.

ggetColnames()
  Get all column names.

  Returns
  Columns names  [list] All column names.

ggetUnitOf(col)
  Get unit of column.

  Parameters
  col  [string] The name of column.

  Returns
  unit  [string] The unit used for this column.

needsUpdate()
  Determine whether data need to be updated.

  Returns
  Update flag  [boolean] True if data need update and False otherwise.

selectByPlanetName(planetName, toScreen=True, caseSensitive=False)
  Get entry by planet name.

  Parameters
  planetName  [string] The name of the planet (includes planet letter, e.g., “corot-2 b”
  caseSensitive  [boolean, optional] If False (default), the search will be case-insensitive.
  toScreen  [boolean, optional] If True (default), the information on the system is printed to screen in human-readable format.
Returns

**Data entry** [dictionary] A dictionary with a key for every data column holding the associated value from the data table.

```python
showAvailableData()
```

Show available data.

Displays the available column names and associated data types, units, and descriptions.

**ExoplanetEU (deprecated)**

**Example: Using ExoplanetEU**

```python
from __future__ import print_function, division
from PyAstronomy import pyasl
import matplotlib.pyplot as plt

eu = pyasl.ExoplanetEU()

# See what information is available
cols = eu.availableColumns()
print(cols)

# Get all data and plot planet Mass vs.
# semi-major axis in log-log plot
dat = eu.getAllData()
plt.xlabel("Planet Mass [RJ]"")
plt.ylabel("Semi-major axis [AU]"")
plt.loglog(dat.plMass, dat.sma, 'b.')
plt.show()
```

**API documentation (ExoplanetEU)**

```python
class PyAstronomy.pyasl.ExoplanetEU(skipUpdate=False)
```

Provides access to exoplanet.eu data base.

This class downloads the data base as a csv file and converts it into a numpy recarray. By default, the data are re-downloaded every 7 days.

The available columns are:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>plName</td>
<td>Name of planet</td>
<td></td>
</tr>
<tr>
<td>plMass</td>
<td>Mass of planet</td>
<td>MJ</td>
</tr>
<tr>
<td>plRadius</td>
<td>Radius of planet</td>
<td>RJ</td>
</tr>
<tr>
<td>period</td>
<td>Orbital period</td>
<td>d</td>
</tr>
<tr>
<td>sma</td>
<td>Semi-major axis</td>
<td>AU</td>
</tr>
<tr>
<td>eccentricity</td>
<td>Orbital eccentricity</td>
<td></td>
</tr>
<tr>
<td>inclination</td>
<td>Orbital inclination</td>
<td>deg</td>
</tr>
<tr>
<td>angDistance</td>
<td>Angular Distance</td>
<td>arcsec</td>
</tr>
<tr>
<td>pubStatus</td>
<td>Publication status</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 19 – continued from previous page

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>discovered</td>
<td>Year of discovery</td>
</tr>
<tr>
<td>updated</td>
<td>Date of data update</td>
</tr>
<tr>
<td>omega</td>
<td>Argument of Periastron</td>
</tr>
<tr>
<td>tperi</td>
<td>Epoch of Periastron</td>
</tr>
<tr>
<td>detType</td>
<td>Detection type</td>
</tr>
<tr>
<td>molecules</td>
<td>List of detected molecules</td>
</tr>
<tr>
<td>stName</td>
<td>Name of star</td>
</tr>
<tr>
<td>ra</td>
<td>Right ascension (J2000)</td>
</tr>
<tr>
<td>dec</td>
<td>Declination (J2000)</td>
</tr>
<tr>
<td>mag_v</td>
<td>V magnitude of a host star</td>
</tr>
<tr>
<td>mag_i</td>
<td>I magnitude of a host star</td>
</tr>
<tr>
<td>mag_j</td>
<td>J magnitude of a host star</td>
</tr>
<tr>
<td>mag_h</td>
<td>H magnitude of a host star</td>
</tr>
<tr>
<td>mag_k</td>
<td>K magnitude of a host star</td>
</tr>
<tr>
<td>dist</td>
<td>Distance to host star</td>
</tr>
<tr>
<td>mh</td>
<td>Metallicity of host star</td>
</tr>
<tr>
<td>stMass</td>
<td>Stellar mass</td>
</tr>
<tr>
<td>stRadius</td>
<td>Radius of star</td>
</tr>
<tr>
<td>SpT</td>
<td>Spectral type of host star</td>
</tr>
<tr>
<td>stAge</td>
<td>Stellar age</td>
</tr>
<tr>
<td>stTeff</td>
<td>Stellar effective temperature</td>
</tr>
<tr>
<td>plRadMM</td>
<td>Measuring method of Rpl</td>
</tr>
</tbody>
</table>

Parameters

**skipUpdate** [boolean, optional] If True, no re-download of the data will be initiated no matter how old they are.

Methods

```
availableColumns()  # Show a summary of the available columns.
changeDownloadCycle(c)  # Change the time after which the data are updated.
dataAge()  # Determine the “age” of the data.
forceUpdate()  # Force a fresh download of the data.
getAllData()  # Provides all data as a numpy recarray.
needsUpdate()  # Determine whether data need to be updated.
```

availableColumns()

Show a summary of the available columns.

Returns

**Column names** [list of strings] The names of the columns.

changeDownloadCycle(c)

Change the time after which the data are updated.

By default, the data will be updated if they are older than the given update cycle. This method allows you to change that cycle.

Parameters

**c** [float or None] The new update cycle in days. If None is provided, updating is switched off.
**dataAge ()**

Determine the “age” of the data.

**Returns**

*age* [float] The time since last data update in days. None, if no age can be determined, e.g., if data have never been downloaded.

**forceUpdate ()**

Force a fresh download of the data.

By default, the data will be updated every 7 days.

**getAllData ()**

Provides all data as a numpy recarray.

**Returns**

*Data* [numpy recarray] All available data. Use, e.g., *availableColumns* to get an overview of the available information.

**needsUpdate ()**

Determine whether data need to be updated.

**Returns**

*Update flag* [boolean] True if data need update and False otherwise.

---

**Access the exoplanets.org data base**

**class PyAstronomy.pyasl.ExoplanetsOrg (skipUpdate=False, verbose=False)**

Easy access to exoplanets.org exoplanet archive.

This class downloads a table of exoplanet data from the exoplanets.org (http://exoplanets.org/csv-files/) and provides access to these data. By default, the data will be re-downloaded every seven days.

The following data are provided:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>pl_name</td>
<td>Name of the planet</td>
<td></td>
</tr>
<tr>
<td>pl_orbper</td>
<td>Planetary orbital period</td>
<td>d</td>
</tr>
<tr>
<td>pl_massj</td>
<td>Planetary mass</td>
<td>MJ</td>
</tr>
<tr>
<td>pl_msini</td>
<td>Minimum planetary mass</td>
<td>MJ</td>
</tr>
<tr>
<td>pl_radj</td>
<td>Planetary radius</td>
<td>RJ</td>
</tr>
<tr>
<td>pl_trandep</td>
<td>Central depth of transit</td>
<td>%</td>
</tr>
<tr>
<td>pl_impact</td>
<td>Impact Parameter</td>
<td>Stellar Radii</td>
</tr>
<tr>
<td>pl_trandur</td>
<td>Transit duration</td>
<td>d</td>
</tr>
<tr>
<td>pl_tranmid</td>
<td>Transit midpoint</td>
<td>BJD</td>
</tr>
<tr>
<td>pl_tperi</td>
<td>Time of Periastron passage</td>
<td>BJD</td>
</tr>
<tr>
<td>pl_orbsmax</td>
<td>Semi-major-axis</td>
<td>AU</td>
</tr>
<tr>
<td>pl_orbsmaxr</td>
<td>Semi-major-axis / R_star</td>
<td>Stellar Radii</td>
</tr>
<tr>
<td>pl_orbinc</td>
<td>Orbital inclination of planet</td>
<td>deg</td>
</tr>
<tr>
<td>pl_missal</td>
<td>Orbital misalignment of planet</td>
<td>deg</td>
</tr>
<tr>
<td>pl_omega</td>
<td>Argument of Periastron</td>
<td>deg</td>
</tr>
<tr>
<td>pl_ecc</td>
<td>Planetary orbital eccentricity</td>
<td>%</td>
</tr>
<tr>
<td>pl_grav</td>
<td>Planetary surface gravity</td>
<td>log10(cm/s^2)</td>
</tr>
<tr>
<td>pl_dens</td>
<td>Planetary Density</td>
<td>g/cm^3</td>
</tr>
</tbody>
</table>

Continued on next page
Table 21 – continued from previous page

<table>
<thead>
<tr>
<th>pl_dtype</th>
<th>Detection type</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>KOI</td>
<td>Kepler ID (if available)</td>
<td></td>
</tr>
<tr>
<td>pl_hostname</td>
<td>Name of host star</td>
<td></td>
</tr>
<tr>
<td>st_binary</td>
<td>Binary Flag</td>
<td></td>
</tr>
<tr>
<td>st_rad</td>
<td>Stellar radii</td>
<td>Solar</td>
</tr>
<tr>
<td>st_dist</td>
<td>Distance to star</td>
<td>pc</td>
</tr>
<tr>
<td>st_par</td>
<td>Stellar parallax</td>
<td>mas</td>
</tr>
<tr>
<td>st_mass</td>
<td>Stellar mass</td>
<td>Solar</td>
</tr>
<tr>
<td>st_teff</td>
<td>Effective temperature of star</td>
<td>K</td>
</tr>
<tr>
<td>st_vsini</td>
<td>Stellar vsin(i)</td>
<td>km/s</td>
</tr>
<tr>
<td>st_logg</td>
<td>Stellar surface gravity</td>
<td>cm/s**2</td>
</tr>
<tr>
<td>stActs</td>
<td>Stellar S-Index</td>
<td></td>
</tr>
<tr>
<td>st_actc</td>
<td>Stellar chromospheric activity</td>
<td></td>
</tr>
<tr>
<td>st_vj</td>
<td>Stellar V-band brightness</td>
<td>mag</td>
</tr>
<tr>
<td>st_fe</td>
<td>Stellar metallicity</td>
<td></td>
</tr>
<tr>
<td>st_radv</td>
<td>System radial velocity</td>
<td>km/s</td>
</tr>
<tr>
<td>st_dens</td>
<td>Density of star</td>
<td>g/cm**3</td>
</tr>
<tr>
<td>K</td>
<td>Velocity Semi-amplitude</td>
<td>m/s</td>
</tr>
<tr>
<td>dec</td>
<td>Declination (J2000)</td>
<td>dms</td>
</tr>
<tr>
<td>ra</td>
<td>Right ascension (J2000)</td>
<td>hms</td>
</tr>
</tbody>
</table>

Methods

**availableColumns**

(share availableColumns)

Shows a list of available data columns.

**changeDownloadCycle**

(c)

Change the time after which the data are updated.

**dataAge**

() Determine the "age" of the data.

**downloadData**

() Trigger download of data.

**getAllData**

() Get all available data.

**needsUpdate**

() Determine whether data need to be updated.

**selectByPlanetName**

(planetName[, caseSensitive])

Get entry by planet name.

**availableColumns** (**verbose=**True)

Shows a list of available data columns.

**Parameters**

**verbose** [boolean, optional] If True (default), prints information to screen.

**Returns**

**columns** [list of strings] The names of the available data columns.

**changeDownloadCycle** (**c**)

Change the time after which the data are updated.

By default, the data will be updated if they are older than the given update cycle. This method allows you to change that cycle.

**Parameters**

**c** [float or None] The new update cycle in days. If **None** is provided, updating is switched off.
**dataAge()**
Determine the “age” of the data.

**Returns**
- **age** [float] The time since last data update in days. None, if no age can be determined, e.g., if data have never been downloaded.

**downloadData()**
Trigger download of data.

**getAllData()**
Get all available data.

**Returns**
- **Data** [numpy recarray] All data stored in the table as a numpy recarray.

**needsUpdate()**
Determine whether data need to be updated.

**Returns**
- **Update flag** [boolean] True if data need update and False otherwise.

**selectByPlanetName(planetName, caseSensitive=False)**
Get entry by planet name.

**Parameters**
- **planetName** [string] The name of the planet (includes planet letter, e.g., “corot-2 b”
- **caseSensitive** [boolean, optional] If False (default), the search will be case-insensitive.

**Returns**
- **Data entry** [dictionary] A dictionary with a key for every data column holding the associated value from the data table.

### Example of usage

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

# Instantiate the access class
epl = pyasl.ExoplanetsOrg()

# Show the available columns
epl.availableColumns()

# Get information in Kepler-5 b
d = epl.selectByPlanetName("kepler-5 b")

# Print whatever information has been received
print()
print("Information on Kepler-5 b")
print()
for k, v in list(d.items()):
    print("%12s %12s" % (k, str(v)))
```
The SWEET-Cat catalog

class PyAstronomy.pyasl.SWEETCat(skipUpdate=False)

Access the SWEET-Cat catalog.

The SWEET-Cat catalog provides parameters for planet host stars.

The following data are provided

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>star</td>
<td>Name of the star</td>
<td></td>
</tr>
<tr>
<td>hd</td>
<td>The HD number (if available)</td>
<td></td>
</tr>
<tr>
<td>ra</td>
<td>The right ascension</td>
<td>hms</td>
</tr>
<tr>
<td>dec</td>
<td>The declination</td>
<td>dms</td>
</tr>
<tr>
<td>vmag</td>
<td>V magnitude</td>
<td>mag</td>
</tr>
<tr>
<td>ervmag</td>
<td>Error on V magnitude</td>
<td>mag</td>
</tr>
<tr>
<td>par</td>
<td>Parallax</td>
<td>mas</td>
</tr>
<tr>
<td>erpar</td>
<td>Error on parallax</td>
<td>mas</td>
</tr>
<tr>
<td>parsource</td>
<td>If par is from Simbad or calculated</td>
<td></td>
</tr>
<tr>
<td>teff</td>
<td>Effective temperature</td>
<td>K</td>
</tr>
<tr>
<td>erteff</td>
<td>Error on effective temperature</td>
<td>K</td>
</tr>
<tr>
<td>logg</td>
<td>Surface gravity</td>
<td>cgs</td>
</tr>
<tr>
<td>erlogg</td>
<td>Error on surface gravity</td>
<td>cgs</td>
</tr>
<tr>
<td>logglc</td>
<td>Surface gravity from LC</td>
<td>cgs</td>
</tr>
<tr>
<td>erlogglc</td>
<td>Error on surface gravity from LC</td>
<td>cgs</td>
</tr>
<tr>
<td>vt</td>
<td>Micro turbulence</td>
<td>km/s</td>
</tr>
<tr>
<td>ertv</td>
<td>Error on micro turbulence</td>
<td>km/s</td>
</tr>
<tr>
<td>metal</td>
<td>Metallicity ([Fe/H])</td>
<td></td>
</tr>
<tr>
<td>ermetal</td>
<td>Error on metallicity</td>
<td></td>
</tr>
<tr>
<td>mass</td>
<td>Mass calculated from Torres et al.</td>
<td>Solar</td>
</tr>
<tr>
<td>ermass</td>
<td>Error on calculated mass</td>
<td>Solar</td>
</tr>
<tr>
<td>author</td>
<td>Author of source</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td>Link to paper in ADS</td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>1 means CAUP’s method. 0 otherwise</td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>When the parameters were updated</td>
<td></td>
</tr>
<tr>
<td>comment1</td>
<td>Special comment</td>
<td></td>
</tr>
<tr>
<td>comment2</td>
<td>Blank</td>
<td></td>
</tr>
</tbody>
</table>

Detailed information can be found here: https://www.astro.up.pt/resources/sweet-cat/ and in the associated publications.

**Attributes**

data [pandas data frame] The catalog data

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changeDownloadCycle(c)</td>
<td>Change the time after which the data are updated.</td>
</tr>
<tr>
<td>dataAge()</td>
<td>Determine the “age” of the data.</td>
</tr>
<tr>
<td>downloadData()</td>
<td>Trigger download of data.</td>
</tr>
<tr>
<td>needsUpdate()</td>
<td>Determine whether data need to be updated.</td>
</tr>
</tbody>
</table>
downloadData()

Trigger download of data.

Example: Access catalog via pandas data frame

```python
from __future__ import print_function
from PyAstronomy import pyasl

sc = pyasl.SWEETCat()

# Access via pandas data frame
# Show first three rows
print(sc.data[0:3])
print()

# Which stars are available?
print("Available stars: ", sc.data["star"].values)
print()

# Check whether star is "available"
star = "XO-5"
starInTable = (star in sc.data.star.values)
pold = "in the table? "
print("Is " + star + " in the table? ", starInTable)

if starInTable:
    # Get information on that star
    print("Data for star: ", star)
    info = sc.data[sc.data["star"] == star]
    print(info)
    print()
    print("Get the effective temperature")
    teff, erteff = info["teff"], info["erteff"]
    print("Effective temperature of " + star + " = \$6.1f +/- \$6.1f$ K" % (teff, erteff))
```

Access to Kurucz atmospheric models

The classes and functions available here provide access to the model grids made available by Robert L. Kurucz on: http://kurucz.harvard.edu/grids.html

**Note:** Model grids are downloaded on first request and stored in PyA's data path.

Example: Get access to the models

```python
from __future__ import print_function, division
from PyAstronomy import pyasl

km = pyasl.KuruczModels()
# See what model grids are available
print(km.availableGrids())

# See whether model grid for log(metallicity) = 0.0
(continues on next page)
```
# is available
print(km.gridAvailable(0.0))

# Obtain the model grid for solar metallicity
mg = km.requestModelGrid(0.0)

# See what Teffs and loggs are available
print("Teffs: ", mg.availableTeffs())
print("Loggs: ", mg.availableLoggs())

print()

print()

# Use simple access method to obtain a model.
# The input is: Teff, logg, and log10(metallicity)
model = pyasl.getKuruczModel(4250, 4.5, 0.1)

---

**Purge data**

If required, e.g., to initiate a re-download, the Kurucz data stored in PyA's data directory can be deleted.

```python
from PyAstronomy import pyasl
pyasl.purgeKuruczData()
```

---

**Classes and functions**

PyAstronomy.pyasl.getKuruczModel (teff, logg, met, nameadd="")

Obtain a Kurucz model

- **Parameters**
  - `teff` [float] Effective temperature [K]
  - `logg` [float] Logarithmic surface gravity [cgs]
  - `met` [float] Logarithmic metallicity, e.g., +0.1.
  - `nameadd` [string, optional] Name extension of the model grid; for instance, “NOVER”.

- **Returns**
  - `Model` [list of strings] The requested model.

PyAstronomy.pyasl.purgeKuruczData ()

Remove downloaded Kurucz data from PyA's data directory

**class** PyAstronomy.pyasl.KuruczModels

Provides access to the Kurucz models.

---

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>availableGrids()</td>
<td>All available model grids.</td>
</tr>
<tr>
<td>getListOfGridsFN()</td>
<td>Get filename for list of grids</td>
</tr>
</tbody>
</table>

---

Continued on next page
Table 24 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridAvailable(met[, add])</td>
<td>Check whether model grid is available.</td>
</tr>
<tr>
<td>listGridLinks()</td>
<td>List links to available grids</td>
</tr>
<tr>
<td>requestModelGrid(met[, add])</td>
<td>Get a model grid.</td>
</tr>
</tbody>
</table>

availableGrids()
All available model grids.

Returns
Available grids [list of strings] The names of all available model grids.

getListOfGridsFN()
Get filename for list of grids

gridAvailable (met, add=’’)
Check whether model grid is available.

Parameters
met [float] Log10 of metallicity.

Returns
Availability flag [boolen] True, if model grid is available.

listGridLinks()
List links to available grids

requestModelGrid (met, add=’’)
Get a model grid.

Parameters
met [float] Log10 of the metallicity. For instance use: 0.0, +0.1, +0.5, or -0.5.

Returns
Model grid [KuruczMT] The model grid enclosed in a class instance, which allows easy access to the models.

class PyAstronomy.pyasl.KuruczMT(fn)
Provides access to individual models in a model grid.

Parameters
fn [string] Name of the Kurucz model file.

Attributes
loggs [array] Available loggs in ascending order [cgs].
models [dictionary] The key is a tuple of the form: (teff, logg, met) and the value is a list of strings representing the model.
met [float] The logarithmic metallicity of the model grid.

Methods

availableLoggs() Get available logg values.
availableTeffs()  Get available effective temperatures.

getModel(teff, logg[, met])  Get a model.

metallicity()  Get metallicity of model grid.

modelAvailable(teff, logg[, met])  Determine whether model is available.

availableLoggs()  Get available logg values.

Note that not all models are available for all logg values.

Returns

Loggs  [array] Array of available loggs sorted in ascending order.

availableTeffs()  Get available effective temperatures.

Returns

Teffs  [array] Array of available effective temperatures sorted in ascending order.

g Model(teff, logg, met=None)  Get a model.

Parameters

  teff  [float] Effective temperatures [K]
  logg  [float] Logg [cgs]
  met  [float, optional] Logarithmic metallicity. If not given, the metallicity of the model grid is used.

Returns

  Model  [list of strings] The model as found on the file.

metallicity()  Get metallicity of model grid.

Returns

  met  [float] Log10 of the metallicity.

modelAvailable(teff, logg, met=None)  Determine whether model is available.

Parameters

  teff  [float] Effective temperatures [K]
  logg  [float] Logg [cgs]
  met  [float, optional] Logarithmic metallicity. If not given, the metallicity of the model grid is used.

Returns

  Availability flag  [boolean] True, if the model is available.
First ionization potential

class PyAstronomy.pyasl.FirstIonizationPot
    First ionization potentials of individual elements.
    The data have been obtained from NIST: http://physics.nist.gov/PhysRefData/ASD/ionEnergy.html

Methods

getFIP(atom) Get the first ionization energy.

Parameters

    atom [string or integer] Either a string specifying the elemental symbol (e.g., ‘H’) or an integer specifying the atomic number.

Returns

    FIP [float] First ionization potential [eV].
    FIP error [float] Error of the FIP [eV]. May also be None if no error is available.

PyAstronomy.pyasl.plotFIP()
    Show a plot of first ionization energy vs. atomic number.

Example

    from __future__ import print_function, division
    from PyAstronomy import pyasl

    fip = pyasl.FirstIonizationPot()

    print("First ionization energy of Li = $4.2e +/- $4.2e eV" % fip.getFIP(3))
    print("First ionization energy of Protactinium = $4.2e +/- $4.2e eV" % fip.getFIP(91))

    # And the same using elemental symbols
    print()
    print("First ionization energy of Li = $4.2e +/- $4.2e eV" % fip.getFIP("Li"))
    print("First ionization energy of Protactinium = $4.2e +/- $4.2e eV" % fip.getFIP("Pa"))

    # Plot the first ionization energy as a function of atomic number
    pyasl.plotFIP()
class `PyAstronomy.pyasl.MMSCETSTable` *(skipUpdate=False, forceUpdate=False)*

Access to the “Modern Mean Stellar Color and Effective Temperature Sequence” table.

A compilation of data in the form of a table entitled “A Modern Mean Stellar Color and Effective Temperature Sequence for O9V-Y0V Dwarf Stars” is made available by E. Mamajek.

In particular, the following columns are provided: SpT, Teff, logT, BCv, Mv, logL, B-V, Bt-Vt, U-B, V-Rc, V-Ic, V-Ks, J-H, H-K, Ks-W1, Msun, logAge, b-y, #SpT, M_J, M_Ks, Mbol”.

Much of the material (but not all) is published in Pecaut & Mamajek (2013, ApJS, 208, 9), which should be cited when using the data, and another reference for a fraction of the data is Pecaut, Mamajek, & Bubar (2012, ApJ 756, 154). More information can be found on the E. Mamajek’s website and within the data file itself (use the `getContent` method to get the entire file).

Entries without a valid entry are represented by NaN in this class.

**Note:** The author of the table, E. Mamajek, states “Please email me if you use the table in your research and/or have any questions. - EEM”.

### Methods

- **availableColumns()**
  Returns a list of available column names.

- **changeDownloadCycle(c)**
  Change the time after which the data are updated.

- **dataAge()**
  Determine the “age” of the data.

- **getColumn(col[, asarray])**
  Get data of specific column.

- **getContent()**
  Get content of data file.

- **getTable()**
  Get table as astropy table object.

- **needsUpdate()**
  Determine whether data need to be updated.

**availableColumns()**

Returns a list of available column names.

**getColumn (col, asarray=True)**

Get data of specific column.

**Returns**

- **column data** [array or column object] Data for the specified column.

**getContent ()**

Get content of data file.

The file contains a lot more information about the sources of the data, a change log etc..

**Returns**

- **content** [list of strings] The content of the file

**getTable ()**

Get table as astropy table object.

**Example**
2.1.2 License note

The “IDL Astronomy User’s Library” is released in “public domain”. That means that nobody holds the copyright and, thus, nobody has the right to sublicense it (not even the authors). As we remain as close as possible in code and ship the documentation mainly as is, it would be quite cumbersome to actually assign copyright to parts written by us. Therefore, code which was directly ported from the IDL Astronomy User’s Library is released in public domain and is marked by a note in the respective documentation saying:

This function was ported from the IDL Astronomy User's Library.

All other code not carrying this note (may include fresh implementations of functionality already provided by the IDL Astronomy User’s Library, newly introduced functions, etc.) is released under the MIT license.

2.2 PyA GUI

2.2.1 PyA Picker — Select points from a plot

The “Picker” is a simple tool to select points in a plot interactively.

Example use of Picker

from __future__ import print_function, division
from PyAstronomy import pyaGui
import numpy as np

# Data for the plot
x = np.linspace(0., 10., 100)
y = np.exp(-x/10.)

(continues on next page)
2.2.2 Interactive model explorer

The “FuncFit model explorer” is a GUI tool, which can be used to manipulate and plot `funcFit` models interactively. Individual model parameters can be selected and their values can be manipulated via the mouse wheel or changed directly.

See the following example to explore its possibilities.

Example—Manipulating a Gaussian

This example demonstrates the basic usage of the model explorer.

```python
import numpy as np
from PyAstronomy import pyaGui
from PyAstronomy import funcFit as fuf

# Create a Gaussian fitting instance
# and set some parameters
gg = fuf.GaussFit1d()
gg["A"] = 1.0
gg["sig"] = 0.5

# Let A and mu be free during a fit
gg.thaw(["A", "mu"])

# Create some artificial data
x = np.linspace(-2., 2., 100)
yerr = np.ones(len(x))*0.01
y = gg.evaluate(x) + np.random.normal(0., 0.01, len(x))

# In order to use the interactive explorer, you
# need a class, which plots the model. The default
# class for this purpose is "FFModelPlotFit", which
# needs the x and y values. Optionally, you can specify
# errors via yerr'. Depending on the setting for
# "withResiduals", the residuals will be shown or not.
mp = pyaGui.FFModelPlotFit(x, y, yerr=yerr, withResiduals=True)

# Use the function ffmodelExplorer (note the lowercase letters)
```
# to create an instance of the FFModelExplorer class, which needs to be given the model (gg in this case) and the plotter (and fitter), which we created above.

g = pyaGui.ffmodelExplorer(gg, mp)
g.show()

Implementation

2.2.3 Interactive data normalization and continuum finder

The ContinuumInteractive class provides a GUI, which helps to interactively normalize data, e.g., for continuum normalization.

Example of usage

```python
from PyAstronomy import pyaGui as pg
import numpy as np
import matplotlib.pylab as plt

# Create some artificial data/spectrum
x = np.arange(1000)
y = np.random.random(1000) + 1000.0 + np.sin(x/80.)
y += np.random.random(len(x))

# Construct an instance of the interactive normalization GUI
cf = pg.ContinuumInteractive(x, y)

# The 'plot' command can be used to draw something on the canvas on which normalization points are selected. Uncomment the following line to see an example.
# cf.plot([4,400,600], [1001,1000,999], 'k--')

# The 'plotNorm' command can be used to draw something on the canvas showing the normalized spectrum (if display is enabled). Uncomment the following line to see an example.
# cf.plotNorm([0.,1000.], [1.001,0.999], 'g:')

# Opens the GUI and starts the interactive session.
c = cf.findContinuum()

# The outcome is a dictionary with the following keys:
#
# - points: A list of two-float tuples holding the x,y location of the selected points.
# - continuum : Array holding the continuum estimate at the given x-values.
# - splineKind : A string specifying the selected spline option.
# - normalizedData : An array holding the normalized data.
```

(continues on next page)
Class documentation

2.2.4 Interactive Gauss/Voigt line fit

The IAGVFit tool allows to interactively fit a series of Gaussian or Voigt profiles to a given data set.

Once the data are on screen (see example below), a component can be added by using the middle mouse button. In particular, three points have to be specified from “left to right”: (1) the outer left points should approximately be placed where the profile reaches half height/depth; (2) the middle points should be placed at the “bottom” of the profile; and (3) the outer right point should be placed at the right-side equivalent of (1). Points (1) and (2) are used to estimate the width of the indicated profile and point (3) is used to estimate the area.

The parameter values are shown in the “Parameters” panel. The “active component” – by default indicated by a black line in the figure – is that whose parameter values are currently shown. The checkboxes preceding “free” indicate whether the associated parameter is considered free in the fit process.

Parameters can be fitted manually using the mouse wheel. Depending on the choice in the “mouse wheel” panel, the parameter under consideration is modified by multiplying with a given factor or adding/subtracting the amount specified in the aforementioned panel. The parameter affected by mouse-wheel action is determined using the outer left checkbox in the “Parameters” panel.

Using the “Set fit range” button, the range of the data to be fitted can be restricted. After clicking the button, to middle-mouse-button clicks into the figure are required to set the “left” and “right” edges of the git range.

Example: Using the interactive fitter

```python
from __future__ import print_function, division
from PyAstronomy import pyaGui
from PyAstronomy import funcFit as fuf
import numpy as np

# Data for the plot
x = np.linspace(5000., 5010, 200)
y = np.ones(len(x))
yerr = np.ones(len(x)) * 0.01
y += np.random.normal(0., 0.01, len(x))

gf = fuf.GaussFit1d()
gf["A"] = -0.3
gf["mu"] = 5004.
gf["sig"] = 0.2
y += gf.evaluate(x)

# Create interactive fitter
igv = pyaGui.IAGVFit(x, y, yerr=yerr, mode="gauss")

r = igv.interactiveFit()

print("Parameters of the fit: ", r[0])
```

(continues on next page)
print("Parameters of active component: ", r[1])
print("No. of components: ", r[2])

Class API documentation

2.3 PyA’s “constants” package

There is probably no one who did not spend some time with looking up constants for some calculation. PyA’s constants package has been introduced to help a little with this task.

This package uses the quantities package to manage units and unit conversion. As most people are, however, interested in having quick access to the numbers, the package focuses on “easy access”.

Note: This package requires the quantities package.

2.3.1 Global vs. class scope

The constants package provides the same functionality in a global (module) scope and a class scope. The global scope has been implemented to allow the easiest possible access. Using the global scope within a function or class is, however, not advisable, because it may cause or suffer from side effects. For this purpose, the constants are also available from an object interface (the class scope), which should be used instead. Both are demonstrated in the examples.

Warning: Using constants from global scope in a function/object may cause or suffer from side effects. Use class scope.

2.3.2 Examples

Below, we give an example of the usage of the constants package. In this example, the module-scope constants are used.

```python
from __future__ import print_function, division
from PyAstronomy import constants as c

# Print a summary of available constants
# on screen
c.summary()

# Which unit system is in use?
print()
print("Current unit system: ", c.getSystem())

# Access a constant
print()
print("Gravitational constant: ", c.G)
# The 'f_' prefix is used as a convention. These
# attributes hold 'Quantity' objects as defined in
```
# the `quantities` package. These encapsulate value
# and unit. The prefixless attribute holds only the
# number.
print(" with unit: ", c.f_G)
print(" error with unit: ", c.f_G_err)

# Change the unit system
print()
print("Change the unit system")
c.setSystem('SI')

# Which unit system is in use?
print("Current unit system: ", c.getSystem())

# Access a constant again...
print()
print("Gravitational constant: ", c.G)
print(" with unit: ", c.f_G)
print(" error with unit: ", c.f_G_err)

# Separate value and unit
print()
print("Value: ", c.f_G.magnitude, ", units: ", c.f_G.units)

# Look up details
print()
print("What exactly was G?")
c.constantDetails("G")

# Apply unit conversion
print()
print("Use some other units")
G_InFeet = c.inUnitsOf("G", "ft**3/(kg * s**2)")
print("G with feet [ft**3/(kg * s**2)]: ", G_InFeet)

from __future__ import print_function, division
from PyAstronomy.constants import PyAConstants

C = PyAConstants()

# Which unit system is in use?
print()
print("Current unit system: ", C.getSystem())

# Access a constant
print()
print("Gravitational constant: ", C.G)
# The 'f_' prefix is used as a convention. These
# attributes hold `Quantity` objects as defined in
# the `quantities` package. These encapsulate value
# and unit. The prefixless attribute holds only the
# number.
print(" with unit: ", C.f_G)
print(" error with unit: ", C.f_G_err)
2.3.3 Custom constants

In order to use a custom set of constants, you need to set up a file holding them in appropriate format. This package uses standard INI-style configuration files. An example of a valid file can look like:

```
[ArbitrarySectionName]
descr = Description of my custom constant
symbol = XXConst
valueSI = 1.7656e58
errSI = 2e58
unitSI = W
unitcgs = erg * s^-1
source = Nonsense for demonstration

[RFootball]
descr = Radius of a professional football
symbol = RFb
valueSI = 0.11
errSI = 0.002
unitSI = m
unitcgs = cm
source = Television
```

Then use `load()` with the name of your custom constants file. `load` will loop through all sections defined in the file and read the definitions. The names of the sections are arbitrary and will be ignored. Each section must contain the following entries:

- **descr**: A description of the constant.
- **symbol**: The symbol (and attribute name) used to represent the constant. Must be unique.
- **valueSI**: The value in SI units.
- **errSI**: The error in SI units (if available, use 0 otherwise).
- **unitSI**: The SI units (used by the `qunatities` package). E.g.: “m^2 * s^-1”.
- **unitcgs**: The unit in the cgs system.
- **source**: From where did you get the numbers?

In case you do not want to predefined constants, you can use the `cleanUp()` function defined in the package. This will delete all currently loaded constants and remove the associated attributes from the package namespace.

2.3.4 Implementation details?

Here we briefly describe how information is managed in this package.

On creation, the module loads a basic constants data-set from a default file. The file is a simple Python configuration (INI format) file. The format is described in the documentation of `load()`, which can also be used to specify and load files holding information about further constants.

All information about the constants is saved in the `inventory` attribute, which is a dictionary mapping “constant symbol”, i.e., the name of attribute used to represent the constant, to a dictionary holding the details.

When a new file has been loaded, the constants now specified in the `inventory` are mapped to module attributes to make them easily accessible. The name of the attribute is the “symbol” defined for the constant. This attribute will only contain the value of the constant. As a convention, attributes with a prefix “f_” will store also the unit. The uncertainty is mapped to an attribute of the for “f_SYMBOL_err”.

The same rules apply to objects of type `PyAConstants`, which provide the same functionality in the scope of a class.
2.3.5 The package API

The PyAConstants class

2.4 The (py)Timing module

The pyTiming module is a collection of submodules implementing algorithms for timing analysis such as Lomb-Scargle periodograms.

Contents:

2.4.1 Harmonic timing analysis using periodograms

The search for harmonic signals in the presence of noise is a fundamental problem of timing analysis. This package provides a collection of periodogram implementations to approach this problem.

The cornerstone of this package is the implementation of the “Generalized Lomb-Scargle (GLS) periodogram”. The class is also available as a stand-alone module via the page of Mathias Zechmeister.

The Generalized Lomb-Scargle Periodogram (GLS)

The GLS class provides an implementation of the Generalized Lomb-Scargle periodogram as introduced by Zechmeister & Kuerster 2009. Compared to the “classical” periodogram presented, e.g., by Lomb 1976, the GLS takes into account measurement errors and a constant term in the fit of the wave function.

The periodogram calculation can be controlled by a number of keyword argument, which can be passed to the constructor. Amongst others, the boundaries of the frequency range and its sampling can be adjusted or specified explicitly, various conventions for the periodogram normalization can be adopted, and the classical Lomb-Scargle periodogram can be obtained.

In the following, we provide a number of examples, presenting some of the functionality.

Example: Simple periodogram with default options

This example shows a basic application of the GLS class, using only default options and an artificial input data set. No errors are specified.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy.pyTiming import pyPeriod

# Create some evenly sampled data including a periodic term.
# Number of data points, input frequency, amplitude of input signal,
# and standard deviation of noise.
N = 1000
f = 0.1
A = 0.15
sig = 0.2

time = np.arange(float(N))
flux = A * np.sin(2. * np.pi*time*f)
```

(continues on next page)
Example: Adding an error column and FAP levels

In this example, an error column will be added. Additionally, we will calculate and show a number of False-Alarm-Probability levels to help distinguish between significant and spurious peaks.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy.pyTiming import pyPeriod

# Create some evenly sampled data including a periodic term.
# Number of data points, input frequency, amplitude of input signal,
# and standard deviation of noise.
N = 1000
f = 0.1
A = 0.05
sig = 0.2

time = np.arange(float(N))
flux = A * np.sin(2. * np.pi*time*f)
# Adding the noise
flux += np.random.normal(0, sig, time.size)
# Adding an error column
err = np.ones(N)*sig

# Compute the GLS periodogram with default options.
# Choose Zechmeister-Kuerster normalization explicitly
clp = pyPeriod.Gls((time, flux, err), norm="ZK")
# Print helpful information to screen
clp.info()

# Define FAP levels of 10%, 5%, and 1%
fapLevels = np.array([0.1, 0.05, 0.01])
# Obtain the associated power thresholds
plevels = clp.powerLevel(fapLevels)

# and plot power vs. frequency.
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.plot(clp.freq, clp.power, 'b.-')
plt.show()
```
# Add the FAP levels to the plot
for i in range(len(fapLevels)):
    plt.plot([min(clp.freq), max(clp.freq)], [plevels[i]*2, '--',
          label="FAP = %4.1f%%" % (fapLevels[i]*100))
plt.legend()
plt.show()

Example: Getting highest-power (best-fit) sine curve

The best-fit sine curve is extracted from the periodogram class.

```python
# Create some evenly sampled data including a periodic term.
# Number of data points, input frequency, amplitude of input signal,
# and standard deviation of noise.
N = 1000
f = 0.171
A = 0.5
sig = 0.2

time = np.arange(float(N))
flux = A * np.sin(2. * np.pi*time*f)
# Adding the noise
flux += np.random.normal(0, sig, time.size)
# Adding an error column
err = np.ones(N)*sig

# Compute the GLS periodogram with default options.
# Choose Zechmeister-Kuerster normalization explicitly
clp = pyPeriod.Gls((time, flux, err), norm="ZK")

# Get index associated with highest power
ifmax = np.argmax(clp.power)
# and highest power and associated frequency
pmax = clp.power[ifmax]
fmax = clp.freq[ifmax]
# Convert frequency into period
hpp = 1./fmax
print("Highest-power period: ", hpp)

# Calculate sine wave associated with 'best-fit' frequency
bestSine = clp.sinmod(time)

plt.subplot(2, 1, 1)
plt.title("Data and sine associated with highest-power frequency")
plt.plot(time, flux, 'b.'
plt.plot(time, bestSine, 'r--')
plt.subplot(2, 1, 2)
plt.title("Folded data")
plt.plot(time/hpp-np.floor(time/hpp), flux, 'b.')
```

(continues on next page)
plt.show()

API documentation

class PyAstronomy.pyTiming.pyPeriod.Gls(lc, fbeg=None, fend=None, Pbeg=None, Pend=None, ofac=10, hifac=1, freq=None, norm='ZK', ls=False, fast=False, verbose=False, **kwargs)

Compute the Generalized Lomb-Scargle (GLS) periodogram.

The Gls class computes the error-weighted Lomb-Scargle periodogram as developed by [ZK09] using various possible normalizations.

The constructor of Gls takes a TimeSeries instance (i.e., a light curve) as first argument. The constructor allows to pass keywords to adjust the freq array, which will be used to calculate the periodogram.

The main result of the calculation, i.e., the power, are stored in the class property power.

Parameters

lc [tuple or list or TimeSeries object] The light curve data either in the form of a TimeSeries object (or any object providing the attributes time, flux, and error) or a tuple or list providing time as first element, flux as second element, and optionally, the error as third element.

fbeg, fend [float, optional] The beginning and end frequencies for the periodogram (inverse units of time axis).

Pbeg, Pend [float, optional] The beginning and end periods for the periodogram (same units as for time axis).

ofac [int] Oversampling factor of frequency grid (default=10).

hifac [float] Maximum frequency \( hifac = \) average Nyquist frequency) (default=1).

freq [array, optional] Contains the frequencies at which to calculate the periodogram. If given, fast and verbose option are not available. If not given, a frequency array will be automatically generated.


ls [boolean, optional] If True, the conventional Lomb-Scargle periodogram will be computed (default is False).

fast [boolean, optional] If True, recursive relations for trigonometric functions will be used leading to faster evaluation (default is False).

verbose [boolean, optional] Set True to obtain some statistical output (default is False).

Examples

Create 1000 unevenly sampled data points with frequency=0.1, measurement error and Gaussian noise

```python
>>> time = np.random.uniform(54000., 56000., 1000)
>>> flux = 0.15 * np.sin(2. * np.pi * time / 10.)
```

Add some noise

2.4. The (py)Timing module   193
```python
>>> error = 0.5 * np.ones(time.size)
>>> flux += np.random.normal(0, error)
```

Compute the full error-weighted Lomb-Periodogram in ‘ZK’ normalization and calculate the significance of the maximum peak.

```python
>>> gls = Gls((time, flux, error), verbose=True)
>>> maxPower = gls.pmax
>>> print("GLS maximum power: ", maxPower)
>>> print("GLS statistics of maximum power peak: ", gls.stats(maxPower))
>>> gls.plot(block=True)
```

**Attributes**

- `power` [array] The normalized power of the GLS.
- `freq` [array] The frequency array.
- `ofac` [int] The oversampling factor of frequency grid.
- `hifac` [float] The maximum frequency.
- `t` [array] The abscissa data values.
- `y` [array] The ordinate data values.
- `e_y` [array] The errors of the data values.
- `norm` [string, {'ZK', 'Scargle', 'HorneBaliunas', 'Cumming', 'wrms', 'chisq'}] The used normalization.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FAP(Pn)</code></td>
<td>Obtain the false-alarm probability (FAP).</td>
</tr>
<tr>
<td><code>info()</code></td>
<td>Prints some basic statistical output screen.</td>
</tr>
<tr>
<td><code>plot([block, period])</code></td>
<td>Create a plot.</td>
</tr>
<tr>
<td><code>pnorm([norm])</code></td>
<td>Assign or modify normalization (can be done afterwards).</td>
</tr>
<tr>
<td><code>powerLevel(FAPlevel)</code></td>
<td>Power threshold for FAP level.</td>
</tr>
<tr>
<td><code>prob(Pn)</code></td>
<td>Probability of obtaining the given power.</td>
</tr>
<tr>
<td><code>probInv(Prob)</code></td>
<td>Calculate minimum power for given probability.</td>
</tr>
<tr>
<td><code>sinmod(t)</code></td>
<td>Calculate best-fit sine curve.</td>
</tr>
<tr>
<td><code>stats(Pn)</code></td>
<td>Obtain basic statistics for power threshold.</td>
</tr>
<tr>
<td><code>toFile(ofile[, header])</code></td>
<td>Write periodogram to file.</td>
</tr>
</tbody>
</table>

**FAP (Pn)**

Obtain the false-alarm probability (FAP).

The FAP denotes the probability that at least one out of M independent power values in a prescribed search band of a power spectrum computed from a white-noise time series is as large as or larger than the threshold, Pn. It is assessed through

\[ FAP(Pn) = 1 - \left(1 - \text{Prob}(P > Pn)\right)^M, \]

where “Prob(P>Pn)” depends on the type of periodogram and normalization and is calculated by using the `prob` method; M is the number of independent power values and is computed internally.
Parameters

\textbf{Pn} \ [\text{float}] \ Power \ threshold.

Returns

\textbf{FAP} \ [\text{float}] \ False \ alarm \ probability.

\textbf{info}()

Prints some basic statistical output screen.

\textbf{plot} (\text{block}=\text{False}, \text{period}=\text{False})

Create a plot.

\textbf{pnorm} (\text{norm}=\text{‘ZK’})

Assign or modify normalization (can be done afterwards).

Parameters

\textbf{norm} \ [\text{string, optional}] \ The \ normalization \ to \ be \ used \ (default \ is \ ‘ZK’).

Examples

\begin{verbatim}
>>> gls.pnorm(‘wrms’)
\end{verbatim}

\textbf{powerLevel} (\text{FAPlevel})

Power threshold for FAP level.

Parameters

\textbf{FAPlevel} \ [\text{float or array_like}] \ “False \ Alarm \ Probability” \ threshold

Returns

\textbf{Threshold} \ [\text{float or array}] \ The \ power \ threshold \ pertaining \ to \ a \ specified \ false-alarm \ probability \ (FAP). \ Powers \ exceeding \ this \ threshold \ have \ FAPs \ smaller \ than \ FAPlevel.

\textbf{prob} (\text{Pn})

Probability of obtaining the given power.

Calculate the probability to obtain a power higher than \textit{Pn} from the noise, which is assumed to be Gaussian.

\textbf{Note:} Normalization (see [ZK09] for further details).

\begin{itemize}
  \item \textit{Scargle}:
    \[ e^{\exp(-Pn)} \]
  \item \textit{HorneBaliunas}:
    \[ \left(1 - 2 \times \frac{Pn}{N - 1}\right)^{(N-3)/2} \]
  \item \textit{Cumming}:
    \[ \left(1 + 2 \times \frac{Pn}{N - 3}\right)^{-(N-3)/2} \]
\end{itemize}

Parameters
**Pn** [float] Power threshold.

**Returns**

**Probability** [float] The probability to obtain a power equal or higher than the threshold from the noise.

**probInv**(*Prob*)
Calculate minimum power for given probability.

This function is the inverse of *Prob(Pn)*. Returns the minimum power for a given probability threshold *Prob*.

**Parameters**

*Prob* [float] Probability threshold.

**Returns**

**Power threshold** [float] The minimum power for the given false-alarm probability threshold.

**sinmod**(*t*)
Calculate best-fit sine curve.

The parameters of the best-fit sine curve can be accessed via the dictionary attribute *hpstat*. Specifically, “amp” holds the amplitude, “fbest” the best-fit frequency, “T0” the reference time (i.e., time of zero phase), and “offset” holds the additive offset of the sine wave.

**Parameters**

*t* [array] Time array at which to calculate the sine.

**Returns**

**Sine curve** [array] The best-fit sine curve (i.e., that for which the power is maximal).

**stats**(*Pn*)
Obtain basic statistics for power threshold.

**Parameters**

*Pn* [float] Power threshold.

**Returns**


**toFile**(*ofile*, header=True)
Write periodogram to file.

The output file is a standard text file with two columns, viz., frequency and power.

**Parameters**

*ofile* [string] Name of the output file.

---

**Classical Lomb-Scargle and FFT**

PyAstronomy provides a class to calculate the Fast Fourier transform (Fourier), and a class to calculate the classical Lomb-Scargle periodogram (LombScargle). The latter can be treated as a special case of the GLS and the GLS class can be used to obtain it.
Base classes

Some commonly needed functionality and data structures are implemented by the below listed classes.

The TimeSeries base class

This class provides an interface for a time series, viz., a time, flux, and (optionally) an error array.

class PyAstronomy.pyTiming.pyPeriod.TimeSeries(time, flux, error=None)

A container class that holds the observed light curve.

Parameters

- time [array] The time array.
- flux [array] The observed flux/data.
- error [array, optional] The error of the data values.

Methods

returnNyquist()  # Calculate the average Nyquist frequency.

returnNyquist()  # Calculate the average Nyquist frequency.

Returns

Nyquist frequency [float] Half the sampling frequency of the time series.

The PeriodBase Base class

This class provides functionality and an interface needed in every periodogram class. The latter are to inherit from this class so that the functionality and interface can be shared.

class PyAstronomy.pyTiming.pyPeriod.PeriodBase

Base class for all periodograms.

This class provides the framework for all periodograms within pyPeriod package.

PeriodBase has a plot method, which can be used to provide a quick-look of the result. The significance of a feature with a power $P_n$ can be assessed using the prob and FAP methods.

Attributes

- power [array] The periodogram power.
- freq [array] The frequencies at which the power are evaluated.

Methods

FAP($P_n$)  # Obtain the false-alarm probability (FAP).

plot(*args, **kwargs)  # Creates a matplotlib figure and axes class instance to visualize the result.

Continued on next page
Table 30 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>powerLevel(FAPlevel)</code></td>
<td>Power threshold for FAP level.</td>
</tr>
<tr>
<td><code>stats(Pn)</code></td>
<td>Obtain basic statistics for power threshold.</td>
</tr>
</tbody>
</table>

**FAP** ($P_n$)

Obtain the false-alarm probability (FAP).

The FAP denotes the probability that at least one out of $M$ independent power values in a prescribed search band of a power spectrum computed from a white-noise time series is as large as or larger than the threshold, $P_n$. It is assessed through

$$FAP(P_n) = 1 - (1 - \text{Prob}(P > P_n))^M,$$

where “Prob($P > P_n$)” depends on the type of periodogram and normalization and is calculated by using the `prob` method; $M$ is the number of independent power values and is computed internally.

**Parameters**

- $P_n$ [float] Power threshold.

**Returns**

- FAP [float] False alarm probability.

**plot(*args, **kwargs)**

Creates a matplotlib figure and axes class instance to visualize the result.

**Parameters:**

- `FAPlevels` - optional, List of false-alarm probability (FAP) levels
- `*args` - optional, Arguments passed to plot method of axes class.
- `**kwargs` - optional, Keyword arguments passed plot method to axes class.

This method provides a quick and simple way to visualize the results of the a periodogram calculation.

**Returns:** The created `Figure` and `Axes` class instances.

**powerLevel(FAPlevel)**

Power threshold for FAP level.

**Parameters**

- `FAPlevel` [float or array] “False Alarm Probability” threshold

**Returns**

- Threshold [float or array] The power threshold pertaining to a specified false-alarm probability (FAP). Powers exceeding this threshold have FAPs smaller than FAPlevel.

**stats(Pn)**

Obtain basic statistics for power threshold.

**Parameters**

- `Pn` [float] Power threshold.

**Returns**

Classical Lomb-Scargle and FFT

Using the classes presented below, several periodograms can be calculated:

- **Fourier transform**
- **Lomb Scargle**

All the presented classes inherit from the PeriodBase class.

The Fourier Transform

```python
class PyAstronomy.pyTiming.pyPeriod.Fourier(lc, norm='Leahy')
```

This class computes the Fast Fourier Transform (FFT) of the input data.

This is actually a wrapper around numpy's FFT routines. The constructor takes the light curve, `lc` (*TimeSeries* instance), as input. The optional argument specifies the normalization of the Fourier Power. Currently, only the normalization according to [Leahy83] is supported, which in the case of purely Poissonian noise results in a mean power of 2.

**Parameters**

- `lc` [TimeSeries instance] The light curve to be analyzed.
- `norm` [optional, string] Normalization method; currently, only default (“Leahy”) is supported.

**Methods**

- `FAP(Pn)` Obtain the false-alarm probability (FAP).
- `Prob(Pn)` Returns the probability to obtain a power larger than the threshold, $P_n$.
- `plot(*args, **kwargs)` Creates a matplotlib figure and axes class instance to visualize the result.
- `powerLevel(FAPlevel)` Power threshold for FAP level.
- `stats(Pn)` Obtain basic statistics for power threshold.

**FAP ($P_n$)**

Obtain the false-alarm probability (FAP).

The FAP denotes the probability that at least one out of $M$ independent power values in a prescribed search band of a power spectrum computed from a white-noise time series is as large as or larger than the threshold, $P_n$. It is assessed through

$$FAP(P_n) = 1 - (1 - \text{Prob}(P > P_n))^M,$$

where “Prob($P$>$P_n$)” depends on the type of periodogram and normalization and is calculated by using the `prob` method; $M$ is the number of independent power values and is computed internally.

**Parameters**

- `Pn` [float] Power threshold.
Returns

FAP  [float] False alarm probability.

Prob(\(Pn\))
Returns the probability to obtain a power larger than the threshold, \(Pn\).
In the calculations, the noise is assumed to be Poisson-distributed.

Note: According to [vdK] the probability to obtain a power larger than a given threshold from the noise is given by

\[
\text{Prob}(p > Pn) = Q(M \times W \times Pn, 2 \times M \times W)
\]

where \(Q(\chi^2, \nu)\) is the cumulative \(\chi^2\) distribution with \(\nu\) d.o.f.

Parameters

\(Pn\) [float] Power threshold.

Returns

FAP [float] The probability to obtain a power larger than the specified threshold from noise.

plot(*args, **kwargs)
Creates a matplotlib figure and axes class instance to visualize the result.

Parameters:

- \(FAP\text{levels}\) - optional, List of false-alarm probability (FAP) levels
- *args - optional, Arguments passed to plot method of axes class.
- **kwargs - optional, Keyword arguments passed plot method to axes class.

This method provides a quick and simple way to visualize the results of the a periodogram calculation.

Returns: The created Figure and Axes class instances.

powerLevel(FAPlevel)
Power threshold for FAP level.

Parameters

FAPlevel [float or array] “False Alarm Probability” threshold

Returns

Threshold [float or array] The power threshold pertaining to a specified false-alarm probability (FAP). Powers exceeding this threshold have FAPs smaller than FAPlevel.

stats(Pn)
Obtain basic statistics for power threshold.

Parameters

\(Pn\) [float] Power threshold.

Returns

Statistics [dictionary] A dictionary containing \{'Pn': \(Pn\), ‘FAP': FAP(\(Pn\)), ‘Prob': Prob(\(Pn\))\} for the specified power threshold, \(Pn\).
The Lomb-Scargle-Periodogram (**fast**)

```python
class PyAstronomy.pyTiming.pyPeriod.LombScargle(lc, ofac, hifac)
```

Calculate the Lomb-Scargle periodogram.

The constructor of `LombScargle` takes a `TimeSeries` instance, i.e., a light curve object, as first argument. It then computes the usual Lomb-Scargle periodogram using a fast algorithm. The frequency array is constructed on the fly based on the oversampling keywords, which are mandatory in this case. The power is normalized according to the prescription of [HB86].

The result, i.e., the power, is stored in the class attribute `power`.

**Note:** Adapted from routine of the same routine in [NR], based on period.pro by Han Wen, August 1996.

### Parameters

- **lc** [TimesSeries instance] The light curve to be analyzed.
- **ofac** [int] Oversampling factor.
- **hifac** [float] Maximum frequency \( freq = hifac \times (\text{average Nyquist frequency}) \).

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FAP(Pn)</code></td>
<td>Obtain the false-alarm probability (FAP).</td>
</tr>
<tr>
<td><code>Prob(Pn)</code></td>
<td>Outdated – use “prob” instaed.</td>
</tr>
<tr>
<td><code>ProbInv(Prob)</code></td>
<td>Outdated – use “probInv” instaed.</td>
</tr>
<tr>
<td><code>plot(*args, **kwargs)</code></td>
<td>Creates a matplotlib figure and axes class instance to visualize the result.</td>
</tr>
<tr>
<td><code>powerLevel(FAPlevel)</code></td>
<td>Power threshold for FAP level.</td>
</tr>
<tr>
<td><code>prob(Pn)</code></td>
<td>Returns the probability to obtain a power larger than the threshold.</td>
</tr>
<tr>
<td><code>probInv(Prob)</code></td>
<td>Returns the minimum power for a given probability level ( Prob ).</td>
</tr>
<tr>
<td><code>stats(Pn)</code></td>
<td>Obtain basic statistics for power threshold.</td>
</tr>
</tbody>
</table>

**FAP** \((Pn)\)

Obtain the false-alarm probability (FAP).

The FAP denotes the probability that at least one out of \( M \) independent power values in a prescribed search band of a power spectrum computed from a white-noise time series is as large as or larger than the threshold, \( Pn \). It is assessed through

\[
FAP(Pn) = 1 - (1 - \text{Prob}(P > Pn))^M,
\]

where “\text{Prob}(P>Pn)” depends on the type of periodogram and normalization and is calculated by using the `prob` method; \( M \) is the number of independent power values and is computed internally.

**Parameters**

- **Pn** [float] Power threshold.

**Returns**

- **FAP** [float] False alarm probability.
Prob($P_n$)
Outdated – use “prob” instead.

ProbInv($Prob$)
Outdated – use “probInv” instead.

plot(*args, **kwargs)
Creates a matplotlib figure and axes class instance to visualize the result.

Parameters:
- *FAPlevels* - optional, List of false-alarm probability (FAP) levels
- *args* - optional, Arguments passed to plot method of axes class.
- **kwargs* - optional, Keyword arguments passed to plot method of axes class.

This method provides a quick and simple way to visualize the results of the a periodogram calculation.

Returns: The created `Figure` and `Axes` class instances.

powerLevel($FAPlevel$)
Power threshold for FAP level.

Parameters
- *FAPlevel* [float or array] “False Alarm Probability” threshold

Returns
- *Threshold* [float or array] The power threshold pertaining to a specified false-alarm probability (FAP). Powers exceeding this threshold have FAPs smaller than FAPlevel.

prob($P_n$)
Returns the probability to obtain a power larger than the threshold.

Compute the probability of obtaining a power value as large or larger the the threshold based on the noise, which is assumes to be Gaussian.

Note: LombScargle calculates the quantity (N-1)/2.*p=p' (in the formalism of [ZK09]), which is de facto the normalization prescription of [HB86]. In this scheme the probability $P(p'>P_n)$ is given by the following statement:

$$P(p' > P_n) = \left(1 - 2 \frac{P_n}{N-1}\right)^{(N-3)/2}$$

If properly normalized to the population variance of the time series, which must be known a priori (usually not the case), the power $p/p_n = p''$ is a direct measure of the SNR as proposed by [Scargle82]:

$$P(p'' > P_n) = exp(-P_n) .$$

This formula is often used erroneously in this context.

Parameters
- *Pn* [float] Power threshold

Returns
- *FAP* [float] False alarm probability.
probInv (Prob)
Returns the minimum power for a given probability level Prob.
This is the inverse of prob(Pn).

Parameters
Prob [float] Probability

Returns
MPL [float] Minimum power level required.

stats (Pn)
Obtain basic statistics for power threshold.

Parameters
Pn [float] Power threshold.

Returns

Examples

In the following, the basic usage of the periodogram classes is demonstrated.

Fourier spectrum of evenly-sampled Poisson data

```python
from __future__ import print_function, division
import numpy
import matplotlib.pylab as plt
# Import pyTiming
from PyAstronomy.pyTiming import pyPeriod

# Create some evenly sampled artificial data (Poisson noise)
time = numpy.arange(1000.)/10.
flux = numpy.random.poisson(1, len(time))

# Build the TimeSeries instance
lc = pyPeriod.TimeSeries(time, flux)

# Compute the Leahy-normalized Fourier transform, plot the time series, and check that the mean power level is 2 as expected.
fft = pyPeriod.Fourier(lc)
fig, ax = fft.plot()
print('Mean power level:', numpy.mean(fft.power))
plt.show()
```

Error-weighted (generalized) Lomb periodogram
from __future__ import print_function, division
import numpy
import matplotlib.pylab as plt
from PyAstronomy.pyTiming import pyPeriod

# Create unevenly sampled data with frequency=0.1, measurement error and Gaussian noise
time = numpy.arange(1000.) + numpy.random.normal(0., 0.1, 1000)
flux = 0.15 * numpy.sin(2. * numpy.pi * time / 10.)
# Add some noise
flux += numpy.random.normal(0, 1, time.size) * 0.5
error = numpy.ones(time.size) * 0.5

# Plot the light curve in top panel
plt.subplot(3, 1, 1)
plt.errorbar(time, flux, yerr=error)

# Build the TimeSeries instance
lc = pyPeriod.TimeSeries(time, flux, error)

# Compute and plot fast Lomb-Scargle periodogram, which does not take errors into account.
ls = pyPeriod.LombScargle(lc, ofac=1, hifac=1)
# Plot the Lomb-Scargle periodogram in middle panel
plt.subplot(3, 1, 2)
plt.plot(ls.freq, ls.power, 'r-')

# Compute the full error-weighted Lomb-Periodogram in 'Cumming' normalization and calculate the significance of the maximum peak.
clp = pyPeriod.Gls(lc, ofac=10, hifac=1, norm="Cumming")
maxPower = numpy.max(clp.power)
print("GLS maximum power: ", maxPower)
print("GLS statistics of maximum power peak: ", clp.stats(maxPower))

# Plot the generalized Lomb-Scargle periodogram in bottom panel.
plt.subplot(3, 1, 3)
plt.plot(clp.freq, clp.power)
# Show the results
plt.show()

References

2.4.2 The Phase Dispersion Minimization (PDM) analysis

The PDM analysis is described by Stellingwerf 1978\(^1\). It is a tool to find (eventually nonsinusodial) periodic variation in time series data by minimizing the dispersion of the folded data set. The dispersion is obtained bin-wise.

Scanner and PDM class

The `Scanner` and `PyPDM` class are used to carry out the PDM analysis.

The Scanner class

class PyAstronomy.pyTiming.pyPDM.Scanner(minVal=0.1, maxVal=10.0, dVal=0.05, mode='period')

The Scanner class is used to define the period/frequency range and steps used in the PDM analysis. It is iterable.

Parameters:
- \(\text{minVal}\) - float, Minimum value,
- \(\text{maxVal}\) - float, Maximum value,
- \(\text{dVal}\) - float, Delta value,
- \(\text{mode}\) - string, optional, Either “period” or “frequency” (default = “period”).

The PyPDM class

class PyAstronomy.pyTiming.pyPDM.PyPDM(time, mag)

This class allows to carry out a “Phase Dispersion Minimization” (PDM) analysis as described in Stellingwerf 1978.

Parameters:
- \(\text{time}\) - array, The time array (arbitrary time units),
- \(\text{mag}\) - array, The corresponding flux in magnitudes.

Properties:
- \(\text{minBinPoints}\) - The minimum number of data points, which may be contained in an individual bin (default = 3).

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdmEquiBin(nbins, scanner)</td>
<td>Carry out the PDM analysis using equidistant bins.</td>
</tr>
<tr>
<td>pdmEquiBinCover(nbins, covers, scanner)</td>
<td>Carry out the PDM analysis using multiple sequences of equidistant bins.</td>
</tr>
<tr>
<td>phase(time, period)</td>
<td>Calculate phases and ordering.</td>
</tr>
</tbody>
</table>

pdmEquiBin \(\text{nbins, scanner}\)

Carry out the PDM analysis using equidistant bins.

This method uses equidistant bins, yet, it pays attention to the number of data points contained in individual bins. If this number is insufficient (as defined by the \(\text{minBinPoints}\) property), adjacent bins are combined (see documentation of \(\text{__setUpEquiBlocks}\)).

Parameters:
- \(\text{nbins}\) - int, Number of bins to use.
- \(\text{scanner}\) - \text{An instance of the Scanner class, defining, which periods/frequencies} are tested.

Returns two arrays: the periods and the associated values of the Theta statistic. These are also saved to the class properties \(\text{periods}, \text{frequencies}, \text{and} \ \text{theta}\).
Note: Whether the first return value is frequency or period depends on the mode of the scanner.

\texttt{pdmEquiBinCover (nbins, covers, scanner)}

Carry out the PDM analysis using multiple sequences of equidistant bins.

The bins used by this method are equidistant, but the phase axis and, thus, the data points are covered by multiple bins. The first sequence of bins is the usual division of the 0-1 interval into \textit{nbins} bins; the following sequences are the same but offset of 1/(nbins*covers) in phase (the phase axis is cyclic).

This method does not check whether bins contain “enough” data points, i.e., it neglects the \textit{minBinPoints} property taken into account by \texttt{pdmEquiBin}. If, however, less than two data points are contained within a bin, it is neglected.

\textbf{Parameters:}

- \textit{nbins} - int, Number of bins to use.
- \textit{scanner} - An instance of the \texttt{Scanner} class, defining, which periods/frequencies are tested.
- \textit{covers} - int, The number of covers, i.e., phase-shifted bin sets.

Returns two arrays: the periods/frequencies and the associated values of the Theta statistic. These are also saved to the class properties \texttt{‘periods’}, \texttt{‘frequencies’}, and \texttt{‘theta’}.

Note: Whether the first return value is frequency or period depends on the mode of the scanner.

\texttt{phase (time, period)}

Calculate phases and ordering.

\textbf{Parameter:}

- \textit{time} - array, Time stamps,
- \textit{period} - float, The period used to calculate phases.

\textbf{Returns:} An unsorted phase array and an array with indices specifying the order. Thus, phase[indi], mag[indi] is the phase-sorted phased light curve.

\section*{Examples}

The following examples demonstrate the functionality of the PyPDM module.

\section*{Using the Scanner}

This example demonstrates the use of the \texttt{Scanner} class.

```python
from __future__ import print_function, division
# Import PDM module
from PyAstronomy.pyTiming import pyPDM

# Get Scanner instance
scanner = pyPDM.Scanner(minVal=0.5, maxVal=1.0, dVal=0.05, mode="period")
# Print the periods covered by the scanner
print("Periods: ", end=' ')
for period in scanner:
    print(period, end=' ')
```
Carry out a PDM analysis

Here we demonstrate how to use the pyPDM class to carry out a PDM analysis.

```python
import numpy
import matplotlib.pylab as plt
from PyAstronomy.pyTiming import pyPDM

# Create artificial data with frequency = 3,
# period = 1/3
x = numpy.arange(100) / 100.0
y = numpy.sin(x*2.0*numpy.pi*3.0 + 1.7)

# Get a `scanner', which defines the frequency interval to be checked.
# Alternatively, also periods could be used instead of frequency.
S = pyPDM.Scanner(minVal=0.5, maxVal=5.0, dVal=0.01, mode="frequency")

# Carry out PDM analysis. Get frequency array
# (f, note that it is frequency, because the scanner's
# mode is `frequency') and associated Theta statistic (t).
# Use 10 phase bins and 3 covers (= phase-shifted set of bins).
P = pyPDM.PyPDM(x, y)
f1, t1 = P.pdmEquiBinCover(10, 3, S)
# For comparison, carry out PDM analysis using 10 bins equidistant
# bins (no covers).
f2, t2 = P.pdmEquiBin(10, S)

# Show the result
plt.figure(facecolor='white')
plt.title("Result of PDM analysis")
plt.xlabel("Frequency")
plt.ylabel("Theta")
plt.plot(f1, t1, 'bp-')
plt.plot(f2, t2, 'gp-')
plt.legend(["pdmEquiBinCover", "pdmEquiBin"])
plt.show()
```

2.4.3 The stringlength method

The `stringlength` method is a technique to search for potentially nonsinusoidal, periodic variation in a data set. The idea consists to fold the data with a number of trial periods. Consecutive points in the phased data set are connected by hypothetical lines and the total length (the string length) is calculated. The strong length statistic assumes a minimum, when the folding produces a well ordered set. The stringlength method is described by in detail by Dworetsky 1983, MNRAS 203, 917-924.

Example: A string length analysis

```python
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyTiming as pyt

period = 1.75
```

(continues on next page)
x = np.linspace(0, 10, 100)
y = 20 * np.sin(2*np.pi*x/period)
y += np.random.normal(0, 1, len(x))

# Trial periods to be tested (200 trial periods between 0.5 and 4.5; 
# same units as x-axis)
tps = (0.5, 4.5, 200)

# Calculate string length
p, sl = pyt.stringlength_dat(x, y, tps)

# Show the string length. An alias at the double period 
# (half frequency) is obvious.
plt.plot(p, sl, 'b.-')
plt.ylabel("String length")
plt.xlabel("Trial period")
plt.show()

### Function documentation

**PyAstronomy.pyTiming.stringlengthDat**

`stringlengthDat(x, m, tps, norm='default', isFreq=False, closed=True)`

Compute string length for data set.

**Parameters**

- `x, m [arrays]` x and y coordinates of data points.
- `tps [tuple or array]` The trial periods (or frequencies): Either a three-tuple specifying (pmin, pmax, nperiods) used by numpy’s linspace or an array of trial periods. If isFreq is True, tps is interpreted as frequencies.
- `isFreq [boolean, optional]` If True, the input tps will be assumed to refer to frequencies instead of periods.
- `norm [string, {default, no}]` If ‘default’ (default), the data points (mi) will be renormalized according to Eq. 3 in Dworetsky 1983, i.e., using mnew = (mi - min(m)) / (2*(max(m) - min(m))). If ‘no’ is specified, the data will not be changed.
- `closed [boolean, optional]` If True (default), first and last point on the phase axis will be connected (close the loop).

**Returns**

- **Trial periods/frequencies [array]** The tested periods (or frequencies if isFreq is True).
- **String length [array]** Associated string lengths

**PyAstronomy.pyTiming.stringlengthPm**

`stringlengthPm(p, m, norm='default', closed=True)`

Compute the string length for phased data set.

**Parameters**

- `m [array]` Data array
- `norm [string, {default, no}]` If ‘default’ (default), the data points (mi) will be renormalized according to Eq. 3 in Dworetsky 1983, i.e., using mnew = (mi - min(m)) / (2*(max(m) - min(m))). If ‘no’ is specified, the data will not be changed.
closed [boolean, optional] If True (default), first and last point on the phase axis will be connected (close the loop).

Returns

sl [float] The string length

PyAstronomy.pyTiming.stringlength_norm(m, norm)
Normalize string length data set.

Parameters

m [array] Data array

norm [string, {default, no}] If ‘default’ (default), the data points (mi) will be renormalized according to Eq. 3 in Dworetsky 1983, i.e., using mnew = (mi - min(m)) / (2*(max(m) - min(m))). If ‘no’ is specified, the data will not be changed.

Returns

ms [array] The normalized data

2.5 Model fitting

2.5.1 funcFit - A convenient fitting interface

The funcFit package provides a convenient interface to the fitting algorithms provided by the popular SciPy and pymc packages. It implements a very flexible and simple parameter handling mechanism making fitting in Python a much more enjoyable experience.

See also:

PyAstronomy's modelSuite. The funcFit package itself comes with only a few basic fitting models such as a Gaussian. More complex models are provided in the frame of the model suite.

Note: If you use the 2.x series of Python, funcFit requires Python 2.7.x. The 2.6.x series (and prior) has a bug affecting the copying of dynamically created class methods, which has not been (and will not be) corrected. This interferes with many of funcFit’s algorithms.

The tutorial

The funcFit tutorial gives you an introduction to the capabilities of the package. The most important aspects are demonstrated by example.

The funcFit tutorial

This tutorial is supposed to enable you to exploit the funcFit functionality by presenting examples and delineate the basic principles.

Please note that this tutorial is not intended to be an introduction to Python or the numpy, SciPy, or matplotlib package. Basic knowledge of these components is desirable.
Sections of this tutorial

- The funcFit tutorial
  - Prerequisites
  - What is it good for?
  - Diving into the first example
  - Introducing a custom model
  - Applying relations
  - Combining models
  - Applying conditional restrictions
  - Minimize the Cash statistic
  - Using “steppar” to determine confidence intervals
  - Use errorConfInterval to determine confidence intervals
  - Using custom objective functions
  - Using an overbinned model
  - Fit two models simultaneously

Prerequisites

To run the example in this tutorial you need to have installed the following packages:

- numpy
- SciPy
- matplotlib

For MCMC sampling, at least one of the following packages should be installed

- pymc (in version 2.x, unfortunately, version 3.x is not supported)
- emcee

We also assume that you have a basic knowledge of Python and these packages (except for pymc).

Warning: funcFit relies on Python 2.7.x. The 2.6.x series (and prior) has a bug affecting the copying of dynamically created class methods, which has not been (and will not be) corrected. This interferes with many of funcFit’s algorithms.

What is it good for?

Popular Python packages such as SciPy already offer implementations of the most common fitting algorithms. Although a great deal can be achieved with them alone, some specific tasks need considerable effort to be achieved. What, for example, if you wish to try different sets of parameters, maybe freeze some of them at a specific value for now and let them vary next? This is possible, yet probably not quite convenient. What if the parameters are restricted or even related by some function?
These are the problems to be approached by the funcFit package. Basically, it is a wrapper around minimization routines defined in other packages (e.g., SciPy). The whole magic is to provide a framework to handle parameters and provide a convenient interface; those of you familiar with XSPEC will probably experience a déjà vu. Read on to find out how it works and how it can make your work easier.

**Note:** In this tutorial, it is assumed that you also have matplotlib installed, which provides a neat interface for plotting under Python. If you did not already install it, have a look at their we page (matplotlib).

### Diving into the first example

After you have installed PyAstronomy (PyA), the funcFit package is ready for being used. As a very first step, let us import the package and see whether we will be able to actually fit something. Therefore, execute the following lines as a script or from the Python interactive command shell:

```python
from PyAstronomy import funcFit as fuf
fuf.status()
```

Depending on your installation the output should look like:

```
Status of funcFit:
--------------------------
Is scipy.optimize available? yes
Is pymc available? yes
  pymc is available in version: 2.3.4
Is emcee available? yes
```

Now let us dive into the business of fitting. The first example shown below demonstrates how to exploit the functionality of funcFit to fit a Gaussian to artificially created data. It shows how free parameters can be specified and restrictions can be applied.

```python
from __future__ import print_function, division
# Import numpy and matplotlib
from numpy import arange, sqrt, exp, pi, random, ones
import matplotlib.pyplot as plt
# ... and now the funcFit package
from PyAstronomy import funcFit as fuf
# Before we can start fitting, we need something to fit.
# So let us create some data...

# Creating a Gaussian with some noise
# Choose some parameters...
gPar = {"A": -5.0, "sig": 10.0, "mu": 10.0, "off": 1.0, "lin": 0.0}
# Calculate profile
x = arange(100) - 50.0
y = gPar["off"] + gPar["A"] / sqrt(2*pi*gPar["sig"]**2) \  
   * exp(-(x-gPar["mu"])**2/(2*gPar["sig"]**2))
# Add some noise
y += random.normal(0.0, 0.01, x.size)
# Let us see what we have done...
plt.plot(x, y, 'bp')

# Now we can start exploiting the funcFit functionality to
```
(continues on next page)
# fit a Gaussian to our data. In the following lines, we
# create a fitting object representing a Gaussian and set guess parameters.

# First, we create the Gauss1d fit object
gf = fuf.GaussFit1d()

# See what parameters are available
print("List of available parameters: ", gf.availableParameters())

# Set guess values for the parameters
gf["A"] = -10.0
gf["sig"] = 15.77
gf["off"] = 0.87
gf["mu"] = 7.5

# Let us see whether the assignment worked
print("Parameters and guess values: ")
print(" A : ", gf["A"])
print(" sig : ", gf["sig"])
print(" off : ", gf["off"])
print(" mu : ", gf["mu"])
print(""

# Now some of the strengths of funcFit are demonstrated; namely, the
# ability to consider some parameters as free and others as fixed.
# By default, all parameters of the GaussFit1d are frozen.

# Show values and names of frozen parameters
print("Names and values of FROZEN parameters: ", gf.frozenParameters())

# Some of the strengths of funcFit are demonstrated; namely, the
# ability to consider some parameters as free and others as fixed.
# By default, all parameters of the GaussFit1d are frozen.

# Which parameters shall be variable during the fit?
# 'Thaw' those (the order is irrelevant)
gf.thaw(["A", "sig", "off", "mu"])

# Let us assume that we know that the amplitude is negative, i.e.,
# no lower boundary (None) and 0.0 as upper limit.
gf.setRestriction({"A": [None, 0.0]})

# Now start the fit
gf.fit(x, y, yerr=ones(x.size)*0.01)

# Write the result to the screen and plot the best fit model
gf.parameterSummary()
plt.plot(x, gf.model, 'r--')
plt.show()
Optimization terminated successfully.
Current function value: 111.455503
Iterations: 176
Function evaluations: 310

Parameters for Component: Gaussian

Parameter: A Gaussian, [ A], value: -4.92037, free: True, restricted: True, related: False
Restriction: [None, 0]
Parameter: mu Gaussian, [ mu], value: 9.83938, free: True, restricted: False, related: False
Parameter: lin Gaussian, [lin], value: 0, free: False, restricted: False, related: False
Parameter: sig Gaussian, [sig], value: 9.97104, free: True, restricted: False, related: False
Parameter: off Gaussian, [off], value: 0.999786, free: True, restricted: False, related: False

Some points in the example shall be emphasized:

- The names of the parameters are defined by the fitting object (in this case GaussFit1d),
- Parameter values can be set and obtained using brackets,
- Individual parameters can be thawed or frozen depending on the needs of the user,
- Restrictions on the parameter ranges can be applied either on both or just on side of the range,
- After the fit, the best-fit values become the current parameters, i.e., they can be obtained using the bracket operator,
- After the fit, the best-fit model can be accessed through the model property.

The central step of the script is the call to fit. The method takes at least two arguments: the x-axis and corresponding y-axis values; errors on the y-axis values can be given optionally via the yerr keyword as shown in the example. In default configuration, the fit method uses the fmin routine provided by SciPy.optimize to minimize either the sum of quadratic residuals of no error is provided, or $\chi^2$ if errors (yerr) are given.

Note: Restrictions are implemented using a penalty function. The steepness of the penalty may be changed by the setPenaltyFactor method or by accessing the penaltyFactor property directly.

Introducing a custom model

The funcFit package comes with some fitting models, but in many cases it will be necessary to use custom models. Introducing a new model is easy in funcFit and will be demonstrated in the next example. Here we implement a straight line and fit it to some artificial data.

```python
# Import numpy and matplotlib
from numpy import arange, random
import matplotlib.pyplot as plt
# ... and now the funcFit package
from PyAstronomy import funcFit as fuf
```

```python
(continues on next page)```
class StraightLine(fuf.OneDFit):
    ""
    Implements a straight line of the form \( y = "off" + x \times "lin". \)
    ""
    def __init__(self):
        fuf.OneDFit.__init__(self, ["off", "lin"])
    def evaluate(self, x):
        ""
        Calculates and returns model according to the current parameter values.
        
        Parameters:
        - `x` - Array specifying the positions at which to evaluate the model.
        ""
        y = self["off"] + (self["lin"] * x)
        return y

# Generate some data and add noise
x = arange(100)
y = 10.0 + 2.0 * x + random.normal(0.0, 5.0, 100)

# Create fitting class instance and set initial guess
# Note that all parameters are frozen by default
lf = StraightLine()
lf["off"] = 20.0
lf["lin"] = 1.0
# Thaw parameters
lf.thaw(["off", "lin"])

# Start fitting
lf.fit(x, y)

# Investigate the result
lf.parameterSummary()
plt.plot(x, y, 'bp')
plt.plot(x, lf.model, 'r--')
plt.show()

This example resembles the first one, but here we defined a custom fitting model at the top instead of using the GaussFit1d class as in the first example.

A new fitting model is a class, which inherits from the OneDFit class. Additionally, two methods (\_init\_ and evaluate) must be implemented. In the example, we provide a minimal constructor (\_init\_ method), which only consists of a call to the base class (OneDFit) constructor. The argument is a list of strings with the names of the variables characterizing the model. The evaluate method takes a single argument, which is an array of values at which to evaluate the model. It returns the function values at the given position. Note how, e.g., self["off"], is used to get the current value if the offset variable in evaluate.

**Applying relations**

In funcFit relations refer to a functional dependence between two or more model parameters. To demonstrate the application of such a relation, we slightly extend the previous example. In particular, we will assume that the gradient...
of our line is a multiple of the offset.

```python
# import numpy and matplotlib
from numpy import arange, random
import matplotlib.pylab as plt
# ... and now the funcFit package
from PyAstronomy import funcFit as fuf

class StraightLine(fuf.OneDFit):
    
    Implements a straight line of the form $y = \text{"off"} + x \times \text{"lin"}$. 

    def __init__(self):
        fuf.OneDFit.__init__(self, ["off", "lin"])

    def evaluate(self, x):
        
        Calculates and returns model according to the current parameter values.

        Parameters:
        x - Array specifying the positions at which to evaluate the model.

        y = self["off"] + (self["lin"] \times x)
        return y

# Create a function, which defines the relation.

def getLinearRelation(factor):
    def linOffRel(off):
        
        Function used to relate parameters "lin" and "off".

        return factor \times off
    return linOffRel

# Note, above we used a nested function (a closure) to define
# the relation. This approach is very flexible. If we were already
# sure about the value of `factor` (e.g., 10.0), we could
# simply have used:
#
# def linOffRel(off):
#     return 10.0 \times off

# Generate some data with noise
x = arange(100)
y = 100.0 + 2.0 \times x + random.normal(0.0, 5.0, 100)

# Create fitting class instance and set initial guess
lf = StraightLine()
lf["off"] = 20.0
lf["lin"] = 1.0
# Thaw parameters
lf.thaw(["off", "lin"])
```

(continues on next page)
# Assume we know about a relation between 'lin' and 'off'
# In particular, lin = 9.0 * off. We use the function getLinearRelation
to obtain a function object defining the relation.
lf.relate("lin", ["off"], getLinearRelation(9))

# Start fitting
lf.fit(x, y)

# Investigate the result
lf.parameterSummary()
plt.plot(x, y, 'bp')
plt.plot(x, lf.model, 'r--')
plt.show()

The output of the script reads (numbers may differ):

```
Optimization terminated successfully.
  Current function value: 251539.530679
  Iterations: 27
  Function evaluations: 54
---------------------------------
Parameters for Component: unnamed
---------------------------------
Parameter: lin , [lin], value: 3.5004, free: False, restricted: False,
  related: True
  Relation: lin = f(off)
Parameter: off , [off], value: 0.388933, free: True, restricted: False,
  related: False
```

**Note:** The lin parameter is no longer free, as it depends on off.

The relate method takes three arguments. The first is the name of the dependent variable (in this case “lin”). The second is a list containing the names of the independent variables (in this case only “off”). The third argument is a callable object, which provides the numerical relation between the independent and the dependent variables (there may be more than one independent variable).

## Combining models

The funcFit package allows to combine two models. That means that models (then becoming model components) can be added, subtracted, divided, multiplied, and even used as exponents. This can be very useful in creating more complex models and requires only little effort. The following example shows how two Gaussians models can be summed.

```python
from __future__ import print_function, division
# Import numpy and matplotlib
from numpy import arange, sqrt, exp, pi, random, ones
import matplotlib.pylab as plt
# ... and now the funcFit package
from PyAstronomy import funcFit as fuf

# Creating Gaussians with some noise
# Choose some parameters...
gPar1 = {"A": -5.0, "sig": 10.0, "mu": 20.0, "off": 1.0, "lin": 0.0}
(continues on next page)
```
gPar2 = {"A": +10.0, "sig": 10.0, "mu": -20.0, "off": 0.0, "lin": 0.0}

# Calculate profile
x = arange(100) - 50.0
y = gPar1["off"] + gPar1["A"] / sqrt(2*pi*gPar1["sig"]**2) \
    * exp(-(x-gPar1["mu"])**2/(2*gPar1["sig"]**2))
y -= gPar2["off"] + gPar2["A"] / sqrt(2*pi*gPar2["sig"]**2) \
    * exp(-(x-gPar2["mu"])**2/(2*gPar2["sig"]**2))

# Add some noise
y += random.normal(0.0, 0.01, x.size)

# Let us see what we have done...
plt.plot(x, y, 'bp')

# Now let us come to the fitting
# First, we create two Gauss1d fit objects
gf1 = fuf.GaussFit1d()
gf2 = fuf.GaussFit1d()

# Assign guess values for the parameters
gf1["A"] = -0.3
gf1["sig"] = 3.0
gf1["off"] = 0.0
gf1["mu"] = +5.0

gf2["A"] = 3.0
gf2["sig"] = 15.0
gf2["off"] = 1.0
gf2["mu"] = -10.0

# Which parameters shall be variable during the fit?
# 'Thaw' those (the order is irrelevant)
gf1.thaw(["A", "sig", "mu"])
gf2.thaw(["sig", "mu", "off"])

# Our actual model is the sum of both Gaussians
twoG = gf1 + gf2

# Show a description of the model depending on the
# names of the individual components
print()
print("Description of the model: ", twoG.description())
print()

# Note that now the parameter names changed!
# Each parameter is now named using the "property"
# (e.g., 'A' or 'sig') as the first part, the component
# "root name" (in this case 'Gaussian') and a component
# number in parenthesis.
print("New parameter names and values: ")
twoG.parameterSummary()

# We forgot to thaw the amplitude of the second Gaussian, but
# we can still do it, but we have to refer to the correct name:
# either by using the (new) variable name:
twoG.thaw("A_Gaussian(2)")
# or by specifying property name, root name, and component number
# separately (note that a tuple is used to encapsulate them):
twoG.thaw(('A', "Gaussian", 2))
# We decide to rather freeze the offset of the second
# Gaussian (we could have used a tuple here, too).
twoG.freeze("off_Gaussian(2)"

# Start fit as usual
twoG.fit(x, y, yerr=ones(x.size)*0.01)

# Write the result to the screen and plot the best fit model
print()
print("--------------------------------")
print("Parameters for the combined fit:")
print("--------------------------------")
twoG.parameterSummary()

# Show the data and the best fit model
plt.plot(x, twoG.model, 'r--')
plt.show()

Note: twoG contains copies (not references) two its “ancestors” (gf1 and gf2). You can, thus, continue using those as usual.

When the models are combined (added in this case), funcFit adds “component identifiers” to the variable names to ensure that they remain unique. A component identifier is simply an appendix to the variable name consisting of an underscore, the model name, and a number. The combined model behaves exactly like the individual ones. It should also be noted that model characteristics such as relations, restrictions, etc., are preserved in the combined model.

### Applying conditional restrictions

Via conditional restrictions complex penalty (or reward) functions can be defined, which keep the fit out or force it into a specific subspace of the parameter space. Conditional restrictions are self-defined callables such as function, which take a number of parameters and return a float, which specifies the penalty. The latter is added to the objective function.

Conditional restrictions are referred to by a unique ID, which is generated as soon as it is added to the model. Note that this ID does not change, when models are combined.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import funcFit as fuf

# Get fitting object for a Gaussian ...
g = fuf.GaussFit1d()
# .. and define the parameters
g["A"] = 0.97
g["mu"] = 0.1
g["sig"] = 0.06

# Generate some "data" with noise included
x = np.linspace(-1.0, 1.0, 200)
y = g.evaluate(x) + np.random.normal(0.0, 0.1, len(x))
yerr = np.ones(len(x)) * 0.1
```

(continues on next page)
def myRestriction(A, sig):
    """
    A conditional restriction.
    Returns
    -------
    Penalty : float
        A large value if condition is violated
        and zero otherwise.
    """
    if A > 10.0*sig:
        return np.abs(A-10.0*sig + 1.0)*1e20
    return 0.0

# Add the conditional restriction to the model and save
# the unique ID, which can be used to refer to that
# restriction.
uid = g.addConditionalRestriction(["A", "sig"], myRestriction)
print("Conditional restriction has been assigned the ID: ", uid)
print()

# Now see whether the restriction is really in place
g.showConditionalRestrictions()

# Define free parameters ...
g.thaw(["A", "mu", "sig"])
# ... and fit the model (restriction included)
g.fit(x, y, yerr=yerr)

# Save the resulting best-fit model
restrictedModel = g.model.copy()

# Remove the conditional restriction and re-fit
g.removeConditionalRestriction(uid)
g.fit(x, y, yerr=yerr)

# Save new model
unrestrictedModel = g.model.copy()

# Plot the result
plt.errorbar(x, y, yerr=yerr, fmt='b.')
plt.plot(x, restrictedModel, 'r--', label="Restricted")
plt.plot(x, unrestrictedModel, 'g--', label="Unrestricted")
plt.legend()
plt.show()
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy import funcFit as fuf

# Get a Gaussian fitting object and
# set some parameters
g = fuf.GaussFit1d()
g["A"] = 5.1
g["sig"] = 0.5
g["mu"] = 3.94

# Generate some data with Poisson statistics
x = np.linspace(0.0, 7., 50)
y = np.zeros(len(x))
for i in range(len(x)):
    y[i] = np.random.poisson(g.evaluate(x[i]))

# Choose free parameters and "disturb" the
# starting parameters for the fit a little.
g.thaw(["A", "sig", "mu"])
for par in g.freeParamNames():
    g[par] += np.random.normal(0.0, g[par]*0.1)

# Fit using Cash statistic and print out
# result.
g.fit(x, y, miniFunc="cash79")
g.parameterSummary()

# Plot the result
plt.plot(x, y, 'bp')
plt.plot(x, g.evaluate(x), 'r--')
plt.show()

Using “steppar” to determine confidence intervals

The “steppar” command can be used to analyze the behavior of the objective function (e.g., \( \chi^2 \)) as the parameter values are varied. In particular, the specified parameter(s) are set to a number of values and the remaining free parameters are fitted.

The example below shows how to determine a confidence interval for the normalization of a Gaussian.

import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy import funcFit as fuf

# Set up a Gaussian model
# and create some "data"
x = np.linspace(0, 2, 100)
gf = fuf.GaussFit1d()
gf["A"] = 0.87
gf["mu"] = 1.0
gf["sig"] = 0.2
y = gf.evaluate(x)
y += np.random.normal(0.0, 0.1, len(x))

# Thaw parameters, which are to be fitted. Note
# that those parameters will also be fitted during
# the stepping; no further parameters will be thawed.
gf.thaw(["A", "mu", "sig"])

# ... and "disturb" starting values a little.
gf["A"] = gf["A"] + np.random.normal(0.0, 0.1)
gf["mu"] = gf["mu"] + np.random.normal(0.0, 0.1)
gf["sig"] = gf["sig"] + np.random.normal(0.0, 0.03)

# Find the best fit solution
gf.fit(x, y, yerr=np.ones(len(x)) * 0.1)

# Step the amplitude (area of the Gaussian) through
# the range 0.8 to 0.95 in 20 steps. Note that the
# last part of 'ranges' ('lin') is optional. You may
# also use 'log'; in this case, the stepping would be
# equidistant in the logarithm.
# In each step of 'A', "mu" and "sig" will be fitted,
# because they had been thawed earlier.
sp = gf.steppar("A", ranges={"A": [0.8, 0.95, 20, 'lin']})

# Extract the values for the Gaussian normalization
# (amplitude) ...
As = list(map(lambda x: x[0], sp))

# ... and chi square.
chis = list(map(lambda x: x[1], sp))

# Find minimum chi square

cmin = min(chis)

# Plot A vs. chi square
plt.title('A vs. $\chi^2$ with 68% and 90% confidence levels')
plt.xlabel("A")
plt.ylabel("$\chi^2$")
plt.plot(As, chis, 'bp-')
plt.plot(As, [cmin+1.0]*len(As), 'k--')
plt.plot(As, [cmin+2.706]*len(As), 'k:')
plt.show()

The next example demonstrates how to step two parameters through given ranges and plot the resulting confidence contours.

```python
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy import funcFit as fuf

# Set up a Gaussian model
# and create some "data"
x = np.linspace(0, 2, 100)
gf = fuf.GaussFit1d()
gf["A"] = 0.87
gf["mu"] = 1.0
gf["sig"] = 0.2
y = gf.evaluate(x)
y += np.random.normal(0.0, 0.1, len(x))

# Thaw parameters, which are to be fitted ...
gf.thaw(["A", "mu", "sig"])
# ... and "disturb" starting values a little.
```
gf["A"] = gf["A"] + np.random.normal(0.0, 0.1)
gf["mu"] = gf["mu"] + np.random.normal(0.0, 0.1)
gf["sig"] = gf["sig"] + np.random.normal(0.0, 0.03)

# Find the best fit solution
gf.fit(x, y, yerr=np.ones(len(x))*0.1)

# Step the amplitude (area of the Gaussian) and the
# center ("mu") of the Gaussian through the given
# ranges.
sp = gf.steppar(["A", "mu"], ranges={"A": [0.8, 0.95, 20],
                                       "mu": [0.96, 1.05, 15]})

# Get the values for 'A', 'mu', and chi-square
# from the output of steppar.
As = list(map(lambda x: x[0], sp))
mus = list(map(lambda x: x[1], sp))
chis = list(map(lambda x: x[2], sp))

# Create a chi-square array using the
# indices contained in the output.
z = np.zeros((20, 15))
for s in sp:
    z[s[3]] = s[2]

# Find minimum chi-square and define levels
# for 68%, 90%, and 99% confidence intervals.
cm = min(chis)
levels = [cm+2.3, cm+4.61, cm+9.21]

# Plot the contours to explore the confidence
# interval and correlation.
plt.xlabel("mu")
plt.ylabel("A")
plt.contour(np.sort(np.unique(mus)), np.sort(np.unique(As)), z,
            levels=levels)

# Plot the input value
plt.plot([1.0], [0.87], 'k+', markersize=20)
plt.show()

Use errorConfInterval to determine confidence intervals

The steppar example shows how confidence intervals may be estimated by exploring the behavior of the objective function manually. The errorConfInterval strives to find the confidence interval automatically.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import funcFit as fuf

# Set up a Gaussian model
# and create some "data"
x = np.linspace(0, 2, 100)
gf = fuf.GaussFit1d()
gf["A"] = 0.87
gf["mu"] = 1.0
```
gf["sig"] = 0.2
y = gf.evaluate(x)
y += np.random.normal(0.0, 0.1, len(x))

# Thaw parameters, which are to be fitted. Note
# that those parameters will also be fitted during
# the stepping; no further parameters will be thawed.
gf.thaw(["A", "mu", "sig"])
# ... and "disturb" starting values a little.
gf["A"] = gf["A"] + np.random.normal(0.0, 0.1)
gf["mu"] = gf["mu"] + np.random.normal(0.0, 0.1)
gf["sig"] = gf["sig"] + np.random.normal(0.0, 0.03)
# Find the best fit solution
gf.fit(x, y, yerr=np.ones(len(x))*0.1)

# Step the amplitude (area of the Gaussian) through
# the range 0.8 to 0.95 in 20 steps. Note that the
# last part of `ranges` ('lin') is optional. You may
# also use 'log'; in this case, the stepping would be
# equidistant in the logarithm.
# In each step of `A`, "mu" and "sig" will be fitted,
# because they had been thawed earlier.
sp = gf.steppar("A", ranges={"A": [0.8, 0.95, 20, 'lin']})
# Extract the values for the Gaussian normalization
# (amplitude) ...
As = [x[0] for x in sp]
# ... and chi square.
chis = [x[1] for x in sp]

# Calculate the confidence interval automatically
cfi90 = gf.errorConfInterval("A", dstat=2.706)
print("90% Confidence interval: ", cfi90["limits")
print(" corresponding objective function values: ", cfi90["OFVals")
print(" number of iterations needed: ", cfi90["iters")

cfi68 = gf.errorConfInterval("A", dstat=1.0)
print("68% Confidence interval: ", cfi68["limits")
print(" corresponding objective function values: ", cfi68["OFVals")
print(" number of iterations needed: ", cfi68["iters")

# Plot A vs. chi square
plt.title('A vs. $\chi^2$ 90% (black) and 68% (blue) confidence intervals')
plt.xlabel("A")
plt.ylabel("\$\chi^2$")
plt.plot(As, chis, 'bp-')
# Indicate confidence levels by vertical lines
plt.plot(As, [cfi90["OFMin"] + 1.0]*len(As), 'g:)
plt.plot(As, [cfi90["OFMin"]+2.706]*len(As), 'g:)
# Plot lines to indicate confidence intervals
plt.plot([cfi90["limits"][0]]*2, [min(chis), max(chis)], 'k--')
plt.plot([cfi90["limits"][1]]*2, [min(chis), max(chis)], 'k--')
plt.plot([cfi68["limits"][0]]*2, [min(chis), max(chis)], 'b--')
plt.plot([cfi68["limits"][1]]*2, [min(chis), max(chis)], 'b--')
plt.show()
Using custom objective functions

By default, funcFit minimizes $\chi^2$ when an error is given and the quadratic model deviation otherwise. It may, however, be necessary to minimize something else such as the likelihood for instance. The following example shows how to apply a custom objective function, in this case, we simply use the linear deviation between model and data (weighted by the error) to define the fit quality.

```python
from __future__ import print_function, division  # Import numpy and matplotlib
from numpy import arange, exp, random, ones, sum, abs
import matplotlib.pylab as plt
# Import funcFit
from PyAstronomy import funcFit as fuf

# Define parameters of faked data
A = 1.0
tau = 10.
off = 0.2
t0 = 40.

# Calculate fake data set
x = arange(100)
y = A*exp(-(x-t0)/tau) * (x > t0) + off
y += random.normal(0., 0.1, 100)
yerr = ones(100)*0.01

# Exponential decay model
edf = fuf.ExpDecayFit1d()

# Define free quantities
edf.thaw(["A", "tau", "off", "t0"])
# Let the amplitude be positive
edf.setRestriction(["A": [0.0, None]])
# Define initial guess
edf.assignValue({"A": 1.0, "tau": 15., "off": 0.2, "t0": 50.})

# Do not use chi square, but the linear deviation from model
# to evaluate quality of fit.
# Use the "MiniFunc" decorator to define your custom objective
# function. This decorator takes the fitting object as an
# argument. The function has to accept two arguments: the
# fitting object and the list of free parameters.

@fuf.MiniFunc(edf)
def mini(edf, P):
m = sum(abs(edf.model - edf.y)/edf.yerr)
print("mini - current parameters: ", P, ", value is: ", m)
return m

# Carry out fit WITH SELF-DEFINED OBJECTIVE FUNCTION
edf.fit(x, y, yerr=yerr, miniFunc=mini)

# Show parameter values and plot best-fit model.
edf.parameterSummary()
plt.errorbar(x, y, yerr)
```

(continues on next page)
### Some points may be highlighted in this example:

- You may have noticed that although the parameter $P$ is given to the `mini` function, it is not used there. You cannot leave it out, however, because the decorator, in fact, creates a more complex object, which needs this information.
- The penalty assignment (for restricted parameters) is done automatically. You do not have to include it in your objective function.
- The custom objective function has to be specified on call to the fit routine (`miniFunc` keyword).

### Using an overbinned model

In some cases it may be necessary to evaluate a model at more points than actually required by, e.g., an observation. The final model is then obtained by averaging a number of points. This may be necessary to take finite integration times of your instrument into account as can be the case in planetary transit modeling.

The `turnIntoRebin` method of `funcFit` provides a convenient way to work with such “overbinned” models; a demonstration is given in the example below.

```python
from __future__ import print_function, division
# Import numpy and matplotlib
from numpy import arange, sqrt, exp, pi, random, ones
import matplotlib.pylab as plt
# ... and now the funcFit package
from PyAstronomy import funcFit as fuf

# Creating a Gaussian with some noise
# Choose some parameters...
gPar = {"A": -5.0, "sig": 10.0, "mu": 10.0, "off": 1.0, "lin": 0.0}
# Calculate profile
x = arange(20)/20.0 * 100.0 - 50.0
y = gPar["off"] + gPar["A"] / sqrt(2*pi*gPar["sig"]**2) \
    * exp(-(x-gPar["mu"])**2/(2*gPar["sig"]**2))
# Add some noise
y += random.normal(0.0, 0.01, x.size)
# Let us see what we have done...
plt.plot(x, y, 'bp')

# First, we create a "GaussFit1d_Rebin" class object (note that the
class object has still to be instantiated, the name is arbitrary).
GaussFit1d_Rebin = fuf.turnIntoRebin(fuf.GaussFit1d)
# Do the instantiation and specify how the overbinning should be
carried out.
gf = GaussFit1d_Rebin()
gf.setRebinArray_Ndt(x, 10, x[1]-x[0])
# See what parameters are available
print("List of available parameters: ", gf.availableParameters())
# Set guess values for the parameters
gf["A"] = -10.0
gf["sig"] = 15.77
gf["off"] = 0.87
gf["mu"] = 7.5
```

(continues on next page)
Let us see whether the assignment worked

```python
print("Parameters and guess values: ")
print(" A : ", gf["A"])
print(" sig : ", gf["sig"])
print(" off : ", gf["off"])
print(" mu : ", gf["mu"])
```

Now some of the strengths of funcFit are demonstrated; namely, the ability to consider some parameters as free and others as fixed.

By default, all parameters of the GaussFit1d are frozen.

Show values and names of frozen parameters

```python
print("Names and values if FROZEN parameters: ", gf.frozenParameters())
```

Which parameters shall be variable during the fit?

'Thaw' those (the order is irrelevant)

```python
gf.thaw(["A", "sig", "off", "mu"])  # Let us assume that we know that the amplitude is negative, i.e., no lower boundary (None) and 0.0 as upper limit.
gf.setRestriction({"A": [None, 0.0]})
```

Now start the fit

```python
gf.fit(x, y, yerr=ones(x.size)*0.01)
```

Write the result to the screen and plot the best fit model

```python
gf.parameterSummary()
```

Plot the final best-fit model

```python
plt.plot(x, gf.model, 'rp--')
```

Show the overbinned (=unbinned) model, indicate by color which point are averaged to obtain a point in the binned model.

```python
for k, v in gf.rebinIdent.items():
    c = "y"
    if k % 2 == 0:
        c = "k"
    plt.plot(gf.rebinTimes[v], gf.unbinnedModel[v], c+'.')
```

Show the data and the best fit model

```python
plt.show()
```

This example is very similar to the very first one. Some differences shall, however, be emphasized:

- Obtaining the model: In this example, we used a model that we called `GaussFit1d_Rebin`. We created the model by calling the `turnIntoRebin` method giving GaussFit1d (by name NOT instance, i.e., we use the class object) as the parameter. The return value of this function is another class object, in particular, GaussFit1d extended by the overbinning functionality. In the next line, we instantiate this extended model and use it, just as we would use the original model.

- In the end, the overbinned model and the final averaged model are juxtaposed to highlight the effect.

Fit two models simultaneously

The following example demonstrates how the `SyncFitContainer` class can be used to fit two different models with a partly overlapping parameter set, but differing x-axes simultaneously.
from __future__ import print_function, division
from PyAstronomy import funcFit as fuf
import numpy
import matplotlib.pylab as plt

# Set up two different x axes.
x1 = numpy.arange(100.)/100. - 0.5
x2 = numpy.arange(150.)/150. - 0.25

# Getting the models ...
gauss = fuf.GaussFit1d()
calor = fuf.CauchyLorentz1d()
# and assign parameters.
gauss.assignValue({"A": 0.02, "sig": 0.1, "mu": 0.0, "off": 1.0, "lin": 0.0})
calor.assignValue({"A": 0.07, "g": 0.1, "mu": 0.2, "off": 1.0, "lin": 0.0})

# Create noisy data.
y1 = gauss.evaluate(x1) + numpy.random.normal(0., 0.01, 100)
y2 = calor.evaluate(x2) + numpy.random.normal(0., 0.01, 150)

# Plot the noisy data.
plt.subplot(2, 1, 1)
plt.errorbar(x1, y1, yerr=numpy.ones(100)*0.01)
plt.subplot(2, 1, 2)
plt.errorbar(x2, y2, yerr=numpy.ones(150)*0.01)

# Now, get ready two fit the data sets simultaneously.
sf = fuf.SyncFitContainer()
# Tell the class about the two components and save the
# component numbers assigned to them:
 gaussCno = sf.addComponent(gauss)
calorCno = sf.addComponent(calor)

print("Component numbers in the syncFit container:")
print( "Gauss: ", gaussCno, " Cauchy-Lorentz: ", calorCno)
print()

# See what happened to the parameters in the
# simultaneous fitting class.
# The variable names have changed.
sf.parameterSummary()

# Thaw all parameters (for later fit) ....
sf.thaw(list(sf.parameters()))
# but not the linear term.
sf.freeze(["lin_Gaussian[s1]", "lin_CauLor[s2]"])

# Tell the class about the identity of parameters,
# either by using the "property name" of the parameter:
sf.treatAsEqual("off")
# or by specifying the names explicitly.
sf.treatAsEqual(["g_CauLor[s2]", "sig_Gaussian[s1]"])

# See what happened to the parameters in the
# simultaneous fitting class.
print()
print("Parameters after 'treatAsEqual' has been applied:")

(continues on next page)
The n-dimensional fitting tutorial

The funcFit package supports n-dimensional fitting, which means that both the domain and the range may be multidimensional.

An example for a multidimensional range would be the fitting of a circle or orbit, when time (1d) is mapped to image coordinates (2d). Fitting, for example, the structure of a point spread function (PSF) would be an example for a multidimensional domain, viz., the image coordinates, which are mapped onto a one-dimensional range (the flux or intensity).

Fitting a circular orbit

This example demonstrates how to fit a 2d model (location in plane) depending on a single variable (time), so that there is a mapping of the form

\[ f : (\mathbb{R} \times \mathbb{R}^m) \rightarrow \mathbb{R}^n, \]

where the \( \mathbb{R}^m \) denotes the parameter vector.

In particular, we assume x,y-position measurements at a number of times. Furthermore, all x and y measurements have an error.

```python
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy import funcFit as fuf
```
# Get the circular model and assign parameter values
c = fuf.Circle2d()
c["r"] = 1.0  
c["t0"] = 0.0  
c["per"] = 3.0  

# Evaluate the model at a number of time stamps
t = np.linspace(0.0, 10.0, 20)
pos = c.evaluate(t)

# Add some error to the "measurement"
pos += np.reshape(np.random.normal(0.0, 0.2, pos.size), pos.shape)
err = np.reshape(np.ones(pos.size), pos.shape) * 0.2

# Define free parameters and fit the model
c.thaw(["r", "t0", "per"])
c.fit(t, pos, yerr=err)
c.parameterSummary()

# Evaluate the model at a larger number of points for plotting
(tt = np.linspace(0.0, 10.0, 200)
model = c.evaluate(tt)

# Plot the result
plt.errorbar(pos[:, 0], pos[:, 1], yerr=err[:, 1],  
             xerr=err[:, 0], fmt='bp')
plt.plot(model[:, 0], model[:, 1], 'r--')
plt.show()

Note: The only difference between the “normal” one-dimensional case and that exemplified here, is the dimension of the y and error arrays.

### Creating “coordinate arrays”

If one has multiple coordinate axes, such as in the case of an image, it is often convenient to represent the coordinates using an appropriate array. `funcFit` provides the `coordinateGrid` function to help constructing an appropriate array. In particular, for n coordinate axes, the function will construct an array with n+1 dimensions, which gives the physical coordinates for every array index.

Note: The user is free to use any format to give the coordinates. For example, the `meshgrid` in numpy could be applied to obtain a similar result. The **Fitting image data with a 2d-Gaussian model** example demonstrates both use cases.

```python
from __future__ import print_function, division
from PyAstronomy import funcFit as fuf
import numpy as np

# Constructing the two individual coordinate axes
(continues on next page)```
x = np.linspace(-2., 2., 50)
y = np.linspace(-2., 2., 50)

# Applying funcFit's "coordinateGrid" helper function
# to build appropriate array-index -> coordinate mapping
# needed for nD fitting.
g = fuf.coordinateGrid(x, y)

print("(x, y) coordinates at index (11, 28): ", g[11, 28])

Fitting image data with a 2d-Gaussian model

The following two examples demonstrate how to fit 2d image data with a 2d Gaussian model using funcFit, i.e., we have the mapping

\[ f : (R^2 \times R^m) \rightarrow R, \]

where, again, \( R^m \) denotes the parameter vector, \( R^2 \) is the image coordinate, and the result is the “level”, “flux”, or whatever the image shows.

In this case, we need to take care of the parameter representation. In principle, we may choose the coordinate format ad libitum, however, the `evaluate` method of the fitting objects has to understand the format. Below, we show two conceivable approaches to specify the parameters.

Let us say, we are given both an x- and a y-axis specifying the coordinates in the image. As demonstrated in Creating “coordinate arrays”, the function `coordinateGrid()` can be used to construct an appropriate coordinate array mapping array index to physical coordinates. The class `GaussFit2d` expects such a coordinate array, and the example below demonstrates how to use it.

**Note:** Below the example, the `evaluate` method of `GaussFit2d` is shown to illustrate the use of the coordinate array.

```python
from PyAstronomy import funcFit as fuf
import numpy as np
import matplotlib.pylab as plt

# Constructing the individual coordinate axes
x = np.linspace(-2., 2., 50)
y = np.linspace(-2., 2., 50)

# Applying funcFit's "coordinateGrid" helper function
# to build appropriate array-index -> coordinate mapping
# needed for nD fitting.
g = fuf.coordinateGrid(x, y)

gf = fuf.GaussFit2d()
gf["sigx"] = 0.75
gf["sigy"] = 0.4
gf["A"] = 1.0
gf["rho"] = 0.4

# Get the "data" by evaluating the model
# and adding some noise. Note that the coordinate
```
# mapping (array g) is passed to evaluate here.
im = gf.evaluate(g)
im += np.reshape(np.random.normal(0.0, 0.1, 2500), (50, 50))
err = np.ones((50, 50))*0.1

# Thaw parameters and fit
gf.thaw("A", "rho")
gf.fit(g, im, yerr=err)

# Show the resulting parameter values ...
gf.parameterSummary()

# ... and plot the result.
plt.title("Image data")
plt.imshow(np.transpose(im), origin="lower")
plt.show()
plt.title("Residuals")
plt.imshow(np.transpose(im - gf.evaluate(g)), origin="lower")
plt.show()

The evaluate method of the GaussFit2d class has the following from:

```python
def evaluate(self, co):
    """
    Evaluates the model for current parameter values.

    Parameters
    ----------
    co : array
        Specifies the points at which to evaluate the model.
    """
    if (self["sigx"] <= 0.0) or (self["sigy"] <= 0.0):
        raise PE.PyAValError("Width(s) of Gaussian must be larger than zero.", \
                             solution="Change width ('sigx/y').")
    if self["rho"] > 1.0:
        raise PE.PyAValError("The correlation coefficient must be 0 <= rho <= 1.", \
                             solution="Change width ('sigx/y').")
    result = 1.0/(2.*pi*self["sigx"]*self["sigy"]*sqrt(1.-self["rho"]**2)) * \
              exp( ((co[:,:,0]-self["mux"])**2/self["sigx"]**2 + (co[:,:,1]-self["muy"])**2/self["sigy"]**2 + \n                     2.*self["rho"]*(co[:,:,0]-self["mux"])*(co[:,:,1]-self["muy"]))/(self["sigx"]*self["sigy"])) / \n             (-2.*sqrt(1.-self["rho"]**2))
    return result
```

Alternatively, you may find it more hand the coordinate axes directly to the evaluate function. This is possible and demonstrated in the following example, which relies on the GaussFit2dTuple class, whose evaluate method expects a tuple of coordinate axes. Note the difference in the evaluate method. In this case, we use numpy’s meshgrid function to create coordinate mapping similar to the one used before.

```python
from PyAstronomy import funcFit as fuf
```

2.5. Model fitting
import numpy as np
import matplotlib.pyplot as plt

# Constructing the individual coordinate axes
x = np.linspace(-2., 2., 50)
y = np.linspace(-2., 2., 50)

# Create the 2d-Gaussian model and assign
# some model parameters.
gf = fuf.GaussFit2dTuple()
gf["sigx"] = 0.75
gf["sigy"] = 0.4
gf["A"] = 1.0
gf["rho"] = 0.4

# Get the "data" by evaluating the model
# and adding some noise. Note that the coordinate
# mapping (array g) is passed to evaluate here.
im = gf.evaluate((x, y))
im += np.reshape(np.random.normal(0.0, 0.1, 2500), (50, 50))
err = np.ones((50, 50))*0.1

# Thaw parameters and fit
gf.thaw(["A", "rho"])
gf.fit((x, y), im, yerr=err)

# Show the resulting parameter values ...
gf.parameterSummary()

# ... and plot the result.
plt.title("Image data")
plt.imshow(np.transpose(im), origin="lower")
plt.show()
plt.title("Residuals")
plt.imshow(np.transpose(im - gf.evaluate((x, y))), origin="lower")
plt.show()

The associated evaluate method looks like this:

def evaluate(self, co):
    ""
    Evaluates the model for current parameter values.
    ""
    Parameters
    ----------
    co : array
        Specifies the points at which to evaluate the model.
    ""
    if (self["sigx"] <= 0.0) or (self["sigy"] <= 0.0):
        raise(PE.PyAValError("Width(s) of Gaussian must be larger than zero.", \
                             solution="Change width ('sigx/y')."))
    if self["rho"] > 1.0:
```python
def model(co, sigx, sigy, rho, mux, muy):
    raise PE.PyAValError("The correlation coefficient must be 0 <= rho <= 1.", \
                        solution="Change width ('sigx/y').")

    xx, yy = meshgrid(co[0], co[1])
    result = 1.0/(2.*pi*sigx*sigy*sqrt(1.-rho**2)) * \
             exp( ((xx-mux)**2/sigx**2 + (yy-muy)**2/sigy**2 - \ 
                 2.*rho*(xx-mux)*(yy-muy)/(sigx*sigy)) / \ 
                 (-2.*(1.-rho**2)) )
    return result
```

### MCMC sampling with funcFit tutorial

Currently, funcFit supports MCMC sampling either via the `pymc` or the `emcee` package. To do this, the model objects provides the `fitMCMC` method (pymc) and the `fitEMCEE` method (emcee). Both are basically independent and can be used separately.

Although the method names start with `fit`, we emphasize that fitting, in the sense of optimization, is not the exact purpose of the analysis carried out here, but sampling from the posterior.

**Note:** The Markov chains produced by `fitMCMC` and `fitEMCEE` can be analyzed using the `TraceAnalysis` class. See this tutorial: *Analyze Markov-Chains using TraceAnalysis*.

### Sections of this tutorial

- **MCMC sampling with funcFit tutorial**
  - Sampling using `pymc` (`fitMCMC`)
    * Using the Markov-Chain Monte-Carlo (MCMC) sampler
    * Simplifying initialization (`MCMCautoParameters`)
    * Sampling with Gaussian and uniform priors
  - Sampling using `emcee` (`fitEMCEE`)
    * Basic sampling with `emcee`
    * Sampling with prior information
  - Sampling from specific distribution using `emcee` (`sampleEMCEE`)
    * Sampling from a Gaussian distribution
    * Estimate mean and STD from Gaussian sample

### Sampling using `pymc` (`fitMCMC`)

The `fitMCMC` method provided by funcFit is not an MCMC sampler itself, but it is a wrapper around functionality provided by a third party package, namely, `PyMC`.

`pymc` is a powerful Python package providing a wealth of functionality concerning Bayesian analysis. `fitMCMC` provides an easy to use interface to `pymc` sampling, which allows to carry out a basic Bayesian data analysis quickly.
Note: To run these examples, pymc must be installed (check the output of funcFit.status() shown at the beginning of this tutorial to see whether this is the case on your system).

Warning: Unfortunately, pymc is only supported in version 2.x. In particular, version 3.x is not supported.

Using the Markov-Chain Monte-Carlo (MCMC) sampler

The following example demonstrates how the funcFit interface can be used to carry out a Bayesian analysis using pymc. For a deeper understanding of the working, adaptability, and logic implemented by pymc, we refer the reader to their web page (PyMC).

```python
from __future__ import print_function, division
from numpy import arange, sqrt, exp, pi, random, ones
import matplotlib.pylab as plt
import pymc
from PyAstronomy import funcFit

# Creating a Gaussian with some noise
# Choose some parameters...
gPar = {"A": -5.0, "sig": 10.0, "mu": 10.0, "off": 1.0, "lin": 0.0}
# Calculate profile
x = arange(100) - 50.0
y = gPar["off"] + gPar["A"] / sqrt(2*pi*gPar["sig"]**2) * exp(-(x-gPar["mu"])**2/(2*gPar["sig"]**2))
# Add some noise
y += random.normal(0.0, 0.01, x.size)
# Now let us come to the fitting
# First, we create the Gauss1d fit object
gf = fuf.GaussFit1d()
# See what parameters are available
print("List of available parameters: ", gf.availableParameters())
# Set guess values for the parameters
gf["A"] = -10.0
gf["sig"] = 15.77
gf["off"] = 0.87
gf["mu"] = 7.5
# Let us see whether the assignment worked
print("Parameters and guess values: ", gf.parameters())

# Which parameters shall be variable during the fit?
# 'Thaw' those (the order is irrelevant)
gf.thaw(["A", "sig", "off", "mu"])

# Now start a simplex fit
gf.fit(x, y, yerr=ones(x.size)*0.01)

# Obatin the best-fit values derived by the simplex fit.
# They are to be used as start values for the MCMC sampling.
# Note that 'A' is missing - we will introduce this later.
```

(continues on next page)
X0 = {"sig": gf["sig"], "off": gf["off"], "mu": gf["mu"]}

# Now we specify the limits within which the individual parameters
# can be varied (for those parameters listed in the 'X0' dictionary).
Lims = {"sig": [-20., 20.], "off": [0., 2.], "mu": [5., 15.]}  

# For the parameters contained in 'X0', define the step widths, which
# are to be used by the MCMC sampler. The steps are specified using
# the same scale/units as the actual parameters.
steps = {"A": 0.01, "sig": 0.1, "off": 0.1, "mu": 0.1}

# In this example, we wish to define our 'own' PyMC variable for the parameter
# 'A'. This can be useful, if nonstandard behavior is desired. Note that this
# is an optional parameter and you could simply include the parameter 'A' into
# The framework of X0, Lims, and steps.
ppa = {}
ppa["A"] = pymc.Uniform("A", value=gf["A"], lower=-20.,
                upper=10.0, doc="Amplitude")

# Start the sampling. The resulting Markov-Chain will be written
# to the file 'mcmcExample.tmp'. In default configuration, pickle
# is used to write that file.
# To save the chain to a compressed 'hdf5'
# file, you have to specify the dbArgs keyword; e.g., use:
# dbArgs = {"db":"hdf5", "dbname":"mcmcExample.hdf5"}
gf.fitMCMC(x, y, X0, Lims, steps, yerr=ones(x.size)*0.01,
    pymcPars=ppa, iter=2500, burn=0, thin=1,
    dbfile="mcmcExample.tmp")

# Reload the database (here, this is actually not required, but it is
# if the Marchov chain is to be analyzed later).
db = pymc.database.pickle.load('mcmcExample.tmp')
# Plot the trace of the amplitude, 'A'.
plt.hist(db.trace("A", 0)[:])
plt.show()

Some points shall be emphasized in this example:

- For MCMC sampling the exact same fit object is used as for “normal” fitting.
- If the yerr keyword is specified in the call to fitMCMC, a Gaussian distribution is assumed for the data points. Otherwise a Poisson distribution is assumed.
- We used the normal simplex fit to obtain starting values for the Markov chain. You may also use, e.g., burn-in.
- In the example, we demonstrated how a uniformly distributed PyMC variable is created. Normally, the fitMCMC method does this for you.
- The result, i.e., the Markov chain, is saved to the file mcmcExample.tmp and is reloaded to obtain the trace of the amplitude.

Note: A convenient analysis of the resulting traces can be carried out using the TraceAnalysis class (see Analysis of Markov-Chains)

We emphasize that PyMC is a powerful and highly adaptable package, which can do a lot more. A more detailed introduction is, however, beyond the scope of this tutorial.
Simplifying initialization (MCMCautoParameters)

It can become cumbersome to define the starting values, steps, and ranges for uniform priors as done in the above example. Using the “auto” methods defined in the fitting class, you can take a short cut. However, be warned:

```
from __future__ import print_function, division
from PyAstronomy import funcFit as fuf
import numpy as np
import matplotlib.pylab as plt

x = np.linspace(0, 30, 1000)
gauss = fuf.GaussFit1d()
gauss["A"] = 1
gauss["mu"] = 23.
gauss["sig"] = 0.5
# Generate some "data" to fit
yerr = np.random.normal(0., 0.05, len(x))
y = gauss.evaluate(x) + yerr
# Thaw the parameters A, mu, and sig
gauss.thaw(["A", "mu", "sig"])
# Define the ranges, which are used to construct the uniform priors and step sizes.
# Note that for "sig", we give only a single value.
# In this case, the limits for the uniform prior will be constructed as [m0-1.5, m0+1.5], where m0 is the # starting value interpreted as the current value of mu (23. in this case).
ranges = {"A": [0, 10], "mu": 3, "sig": [0.1, 1.0]}
# Generate default input for X0, lims, and steps
X0, lims, steps = gauss.MCMCautoParameters(ranges)

# Show what happened...
print()
print("Auto-generated input parameters:")
print("X0: ", X0)
print("lims: ", lims)
print("steps: ", steps)
print()
# Call the usual sampler
gauss.fitMCMC(x, y, X0, lims, steps, yerr=yerr, iter=1000)

# and plot the results
plt.plot(x, y, 'k+')
plt.plot(x, gauss.evaluate(x), 'r--')
plt.show()
```

You may even shorten the short-cut by using the autoFitMCMC method. However, note that the same warning remains valid here.

```
from PyAstronomy import funcFit as fuf
import numpy as np
import matplotlib.pylab as plt
```
x = np.linspace(0, 30, 1000)
gauss = fuf.GaussFit1d()
gauss["A"] = 1
#gauss["mu"] = 23.
gauss["sig"] = 0.5
# Generate some "data" to fit
yerr = np.random.normal(0., 0.05, len(x))
y = gauss.evaluate(x) + yerr

# Define the ranges, which are used to construct the
# uniform priors and step sizes.
# Note that for "sig", we give only a single value.
# In this case, the limits for the uniform prior will
# be constructed as [m0-1.5, m0+1.5], where m0 is the
# starting value interpreted as the current value of
# mu (23. in this case).
ranges = {"A": [0, 10], "mu": 3, "sig": [0.1, 1.0]}

# Call the auto-sampler
# Note that we set picky to False here. In this case, the
# parameters specified in ranges will be thawed automatically.
# All parameters not mentioned there, will be frozen.
gauss.autoFitMCMC(x, y, ranges, yerr=yerr, picky=False, iter=1000)

# and plot the results
plt.plot(x, y, 'k+')
plt.plot(x, gauss.evaluate(x), 'r--')
plt.show()
gf.thaw(['A', 'off', 'mu', 'sig'])

# Set up a normal prior for the offset parameter
# Note!---The name (first parameter) must correspond to that
# of the parameter.
# The expectation value is set to 0.9 while the width is given
# as 0.01 (tau = 1/sigma**2). The starting value is specified
# as 1.0.
offPar = pymc.Normal('off', mu=0.9, tau=(1./0.01)**2, value=1.0)
# Use a uniform prior for mu.
muPar = pymc.Uniform('mu', lower=0.95, upper=0.97, value=0.96)

# Collect the "extra"-variables in a dictionary using
# their names as keys
pymcPars = {'mu': muPar, 'off': offPar}

# Specify starting values, X0, and limits, lims, for
# those parameter distributions not given specifically.
X0 = {'A': gf['A'], 'sig': gf['sig']}
lims = {'A': [-1.0, 0.0], 'sig': [0., 1.0]}
# Still, the steps dictionary has to contain all
# parameter distributions.
steps = {'A': 0.02, 'sig': 0.02, 'mu': 0.01, 'off': 0.01}

# Carry out the MCMC sampling
gf.fitMCMC(x, y, X0, lims, steps, yerr=np.ones(len(x))*0.05, pymcPars=pymcPars, burn=1000, iter=3000)

# Setting parameters to mean values
for p in gf.freeParameters():
    gf[p] = gf.MCMC.trace(p)[:].mean()

# Show the "data" and model in the upper panel
plt.subplot(2, 1, 1)
plt.title('Data and model')
plt.errorbar(x, y, yerr=np.ones(len(x))*0.05, fmt='bp')
plt.plot(x, gf.evaluate(x), 'r--')

# Show the residuals in the lower panel
plt.subplot(2, 1, 2)
plt.title('Residuals')
plt.errorbar(x, y-gf.evaluate(x), yerr=np.ones(len(x))*0.05, fmt='bp')
plt.plot([min(x), max(x)], [0.0, 0.0], 'r-')
plt.show()

Clearly, the plot shows that the solution does not fit well in a Chi-square sense, because the prior information has a
significant influence on the outcome. Whether this should be considered reasonable or not is not a question the sampler
could answer.

**Sampling using emcee (fitEMCEE)**

The *emcee* package relies on sampling with an ensemble of chains, the so-called walkers. In this way, the sampling is
automatically adapted to the scale of the problem, which simplifies obtained reasonable acceptance rates.
Basic sampling with emcee

The following example shows a basic application of fitEMCEE. By default, the resulting Markov chain is saved to a file called 'chain.emcee'.

```python
# Import numpy and matplotlib
from numpy import arange, sqrt, exp, pi, random, ones
import matplotlib.pyplot as plt

# ... and now the funcFit package
from PyAstronomy import funcFit as fuf

# Before we can start fitting, we need something to fit.
# So let us create some data...

# Choose some signal-to-noise ratio
snr = 25.0

# Creating a Gaussian with some noise
# Choose some parameters...
gf = fuf.GaussFit1d()
gf.assignValues({'A': -5.0, 'sig': 2.5, 'mu': 10.0, 'off': 1.0, 'lin': 0.0})

# Calculate profile
x = arange(100) - 50.0
y = gf.evaluate(x)

# Add some noise
y += random.normal(0.0, 1.0/snr, x.size)

# Define the free parameters
gf.thaw(["A", "sig", "mu", "off"])

# Start a fit (quite dispensable here)
gf.fit(x, y, yerr=ones(x.size)/snr)

# Say, we want 200 burn-in iterations and, thereafter,
# 1000 further iterations (per walker).
sampleArgs = {'iters': 1000, 'burn': 200}

# Start the sampling (ps could be used to continue the sampling)
ps = gf.fitEMCEE(x, y, yerr=ones(x.size)/snr, sampleArgs=sampleArgs)

# Plot the distributions of the chains
# NOTE: the order of the parameters in the chain object is the same
# as the order of the parameters returned by freeParamNames()
for i, p in enumerate(gf.freeParamNames()):
    plt.subplot(len(gf.freeParamNames()), 1, i+1)
    plt.hist(gf.emceeSampler.flatchain[::, i], label=p)
    plt.legend()
plt.show()
```

Sampling with prior information

In this example, we use a very simple constant model to, first, compare the result from sampling with classical error estimation; note that in this simple case Bayesian credibility intervals are, indeed, numerically identical to classical confidence intervals, which is, however, not generally the case. Second, we introduce (strong) prior information and repeat the sampling.
A number of ready-to-use priors are implemented here: emceePriors

```python
from __future__ import print_function, division
# Import numpy and matplotlib
from numpy import arange, sqrt, exp, pi, random, ones
import matplotlib.pylab as plt
# ... and now the funcFit package
from PyAstronomy import funcFit as fuf
import numpy as np

# Before we can start fitting, we need something to fit.
# So let us create some data...

# Choose some signal-to-noise ratio
snr = 25.0

# Choosing an arbitrary constant and ...
c = 10.0
# ... an equally arbitrary number of data points
npoint = 10

# Define 'data'
x = arange(npoint)
y = np.ones(len(x)) * c
# Add some noise
y += random.normal(0.0, 1.0/snr, x.size)

# A funcFit object representing a constant
pf = fuf.PolyFit1d(0)
pf["c0"] = c

# The only parameter shall be free
pf.thaw("c0")

# Say, we want 200 burn-in iterations and, thereafter,
# 2500 further iterations (per walker).
sampleArgs = {"iters": 2500, "burn": 200}

# Start the sampling (ps could be used to continue the sampling)
ps = pf.fitEMCEE(x, y, yerr=ones(x.size)/snr, sampleArgs=sampleArgs)
print()

# Plot the distributions of the chains
# NOTE: the order of the parameters in the chain object is the same
# as the order of the parameters returned by freeParamNames()
hs = plt.hist(pf.emceeSampler.flatchain[:, 0], label="c0", normed=True)
# Construct "data points" in the middle of the bins
xhist = (hs[1][1:] + hs[1][0:-1]) / 2.0
yhist = hs[0]

# Fit the histogram using a Gaussian
gf = fuf.GaussFit1d()
gf.assignValues({"A": 1.0, "mu": c, "sig": 1.0/snr/np.sqrt(npoint)})
# First fitting only "mu" is simply quite stable
gf.thaw("mu")
gf.fit(xhist, yhist)
gf.thaw(["A", "sig"])
gf.fit(xhist, yhist)
```

(continues on next page)
print()
print("--- Sampling results ---")
print("Posterior estimate of constant: ", np.mean(pf.emceeSampler.flatchain[:, 0]))
print("Nominal error of the mean: ", 1.0/snr/np.sqrt(npoint))
print("Estimate from Markov chain: ", np.std(pf.emceeSampler.flatchain[:, 0]), end='→')
print(" and from Gaussian fit to distribution: ", gf["sig"])  # Evaluate best-fit model ...
xmodel = np.linspace(c - 10.0/snr, c + 10.0/snr, 250)
ymodel = gf.evaluate(xmodel)  # ... and plot
plt.plot(xhist, yhist, 'rp')
plt.plot(xmodel, ymodel, 'r--')
plt.legend()
plt.show()

# Defining a prior on c0. Prior knowledge tells us that its value
# is around 7. Let us choose the standard deviation of the prior so
# that the estimate will lie in the middle between 7 and 10. Here we
# exploit symmetry and make the prior information as strong as the
# information contained in the likelihood function.
priors = {"c0": fuf.FuFPrior("gaussian", sig=1.0/snr/np.sqrt(npoint), mu=7.0)}

# Start the sampling (ps could be used to continue the sampling)
ps = pf.fitEMCEE(x, y, yerr=ones(x.size)/snr, sampleArgs=sampleArgs, priors=priors)
print()
print("--- Sampling results with strong prior information ---")
print("Posterior estimate of constant: ", np.mean(pf.emceeSampler.flatchain[:, 0]), end='→')
print(" +/-", np.std(pf.emceeSampler.flatchain[:, 0]))
plt.hist(pf.emceeSampler.flatchain[:, 0], label="c0", normed=True)
plt.show()

Sampling from specific distribution using emcee (sampleEMCEE)

The function sampleEMCEE() allows to sample arbitrary distributions using the emcee implementation of MCMC sampling without the need to define a formal funcFit model. This function is only a wrapper around emcee functionality. The resulting chains can be studied using the TraceAnalysis class.

Sampling from a Gaussian distribution

The following example shows how to use sampleEMCEE() to obtain realizations from a Gaussian probability distribution. Of course there are more advantageous ways of doing this.

```python
from __future__ import print_function
import numpy as np
from PyAstronomy import funcFit as fuf
import matplotlib.pyplot as plt
```
```python
def lfGauss(v, sigma, mu):
    r"""
    Gaussian density
    Parameters
    ----------
    v : dictionary
        Holds current values of "x"
    mus, sigma : float
        Mean and standard deviation of the Gaussian. Specified via
        the 'largs' argument.
    Returns
    -------
    lp : float
        Natural logarithm of the density.
    """
    result = 0.0
    # Log(density)
    result += -0.5*np.log(2.*np.pi*sigma**2) - (v["x"] - mu)**2/(2.*sigma**2)
    return result

# Sampling arguments
# burn: Number of burn-in steps per walker
# iters: Number of iterations per walker
sa = {"burn": 1000, "iters": 5000}

# Starting values
fv0 = {"x": 0.5}
# Specify standard deviation and mean of Gaussian
la = {"mu": 0.5, "sigma": 0.25}

# Sample from distribution
ps = fuf.sampleEMCEE(["x"], fv0, lfGauss, largs=la, sampleArgs=sa, nwalker=4, dbfile="gauss.emcee")
print()

# Use TraceAnalysis to look at chains
ta = fuf.TraceAnalysis("gauss.emcee")
print("Available chains: ", ta.availableParameters())
print("Mean and STD of chain: ", np.mean(ta["x"]), np.std(ta["x"]))

# Check distribution of chain
# Plot histogram of chain
plt.hist(ta["x"], 60, normed=True)
# Overplot Gaussian model
xx = np.linspace(la["mu"]-6*la["sigma"], la["mu"]+6*la["sigma"], 1000)
yy = 1./np.sqrt(2.*np.pi*la["sigma"]**2) * np.exp(-(xx - la["mu"])**2/(2.*la["sigma"]**2))
plt.plot(xx, yy, 'r--')
plt.show()
```

242 Chapter 2. PyAstronomy's sub-packages
Estimate mean and STD from Gaussian sample

MCMC sampling is frequently used to sample from posterior distributions and obtain marginal distributions. The following example demonstrates how to estimate the mean and standard deviation from a sample of Gaussian data using `sampleEMCEE()`.

```python
from __future__ import print_function
import numpy as np
from PyAstronomy import funcFit as fuf

def lfGaussMS(v, x=None):
    
    Gaussian posterior with 1/sigma prior on sigma.

    Parameters
    ----------
    v : dictionary
        Holds current values of "sigma" and "mu"
    x : array
        The 'data' observed. Will be specified by the 'largs' keyword.

    Returns
    -------
    lp : float
        Natural logarithm of the density.
    
    if v["sigma"] < 0.:
        # Penalize negative standard deviations
        return -1e20*abs(v["sigma"])

    result = 0.0
    # Apply prior on sigma
    result -= np.log(v["sigma"])
    # Add log(likelihood)
    result += np.sum(-0.5*np.log(2.*np.pi*v["sigma"]**2) - (x - v["mu"])**2/(2.*v["sigma"]**2))
    return result

# Sampling arguments
# burn: Number of burn-in steps per walker
# iters: Number of iterations per walker
sa = {"burn": 1000, "iters": 5000}

# Starting values
fv0 = {"sigma": 1., "mu": 1.}
# 'Observed' data
la = {"x": np.random.normal(0., 1., 1000)}

print("Mean of 'data': ", np.mean(la["x"]))
print("Standard deviation of 'data': ", np.std(la["x"]))

# Scale width for distributing the walkers
s = {"mu": 0.01, "sigma": 0.5}
ps = fuf.sampleEMCEE("mu", "sigma"), fv0, lfGaussMS, largs=la, sampleArgs=sa,
    nwalker=4,
```

Analyze Markov-Chains using TraceAnalysis

After having using $pymc$, e.g., via PyAstronomy’s wrapper $fitMCMC$ and having created a Markov Chain, the information needs to be extracted from that Chain. TraceAnalysis is a helper class designed to help you doing this.

This collection of examples demonstrates how to create a Markov Chain and carry out a basic analysis.

Create a Markov-Chain to work with

The following example demonstrates how to create the Markov-Chain, which will later be analyzed.

```python
from __future__ import print_function, division
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import funcFit as fuf

# Starting from with Voigt profile
vp = fuf.Voigt1d()
# Set some values to create a model
vp["A"] = -0.4
vp["al"] = 0.7
vp["mu"] = 5500.
vp["ad"] = 0.3
vp["off"] = 1.0

x = np.linspace(5490., 5510., 200)
# Create our data with some noise
yerr = np.ones(len(x))*0.01
y = vp.evaluate(x) + np.random.normal(0.0, 0.01, len(x))

# Say, we have a guess of the parameters, which is, however,
# not entirely correct
vp["A"] = -0.376
vp["al"] = 0.9
vp["mu"] = 5499.7
vp["ad"] = 0.4
vp["off"] = 1.0
```
# Plot the data and our guess
plt.errorbar(x, y, yerr=yerr, fmt='b.-')
plt.plot(x, vp.evaluate(x), 'r--')
plt.show()

# Thaw the parameters, which we wish to vary
during the sampling
vp.thaw(['A', 'al', 'mu', 'ad'])

# Use current parameters as starting point for the sampling
X0 = vp.freeParameters()
print("Starting point for sampling: ", X0)

# Using pymc for sampling, for emcee see below

lims = {'A': [-1.0, 0.0], 'al': [0.0, 3.0], 'ad': [0.0, 3.0], 'mu': [5495., 5505.]}  

# Provide a guess for the proposal step widths.
# Try to guess the scale of the problem in the individual
# parameters.
steps = {'A': 0.02, 'al': 0.01, 'ad': 0.01, 'mu': 0.05}

# Start the sampling. The resulting Marchov-Chain will be written
to the file 'mcmcTA.tmp'. In default configuration, pickle
# is used to write that file.
# To save the chain to a compressed 'hdf5'
# file, you have to specify the dbArgs keyword; e.g., use:
# dbArgs = {'db':'hdf5', 'dbname':'mccmExample.hdf5'}
vp.fitMCMC(x, y, X0, lims, steps, yerr=yerr,
iter=2500, burn=0, thin=1,
       dbfile="mcmcTA.tmp")

# Uncomment to use emcee for sampling
# priors = {'al':fuf.FuFPrior("limuniform", lower=0.0, upper=100.),
# "ad":fuf.FuFPrior("limuniform", lower=0.0, upper=100.)}

# Note that the filename should end in .emcee. Substitute this filename
# in the following examples.
# vp.fitEMCEE(x, y, yerr=yerr, sampleArgs={'iters':2500},
# dbfile="mcmcTA.emcee", priors=priors)

Investigate convergence behavior using the “deviance”

This example demonstrates how to investigate whether the chain has converged using the deviance and setting an appropriate burn-in.
from PyAstronomy import funcFit as fuf

# Create an instance of TraceAnalysis
# telling it which file to use
ta = fuf.TraceAnalysis("mcmcTA.tmp")

# Have a look at the deviance to check if and when
# the chains reached equilibrium.
ta.plotTrace("deviance")
ta.show()

# Say, we are sure that after 500 iterations, the chain
# reached equilibrium. We use this as the burn-in phase
# setBurn(500)

# Have a second look at the deviance, this time considering
# the burn-in. Note that the first 500 iterations are not
# removed from the chain. They are just not considered any
# more.
ta.plotTrace("deviance")
ta.show()

Accessing the traces

Here, we show how to access the traces and calculate mean, median, standard deviation, and Highest Probability Density (HPD) credibility intervals.

from __future__ import print_function, division
from PyAstronomy import funcFit as fuf

# Create an instance of TraceAnalysis
# telling it which file to use
ta = fuf.TraceAnalysis("mcmcTA.tmp")

# Use the burn-in from the previous example
# setBurn(500)

# See which model parameters have been sampled
print("Available parameters: ", ta.availableParameters())

# Access the traces of these parameters
print("Trace for A: ", ta["A"])""

# Calculate mean, median, standard deviation, and
# credibility interval for the available parameters
for p in ta.availableParameters():
    hpd = ta.hpd(p, cred=0.95)
    print("Parameter {}s, mean = {} g, median = {} g, std = {} g, 95%% HPD = {} g - {} g" \
          "(p, ta.mean(p), ta.median(p), ta.std(p), hpd[0], hpd[1])")

Analyze parameter correlations

Parameter correlations are important in many analysis. This example shows how such analyses can be studied using the Markov-Chain.
from __future__ import print_function, division
from PyAstronomy import funcFit as fuf

# Create an instance of TraceAnalysis
# telling it which file to use
ta = fuf.TraceAnalysis("mcmcTA.tmp")

# Use the burn-in from the previous example
ta.setBurn(500)

# Have a look at the parameter correlations
ta.correlationTable()

# Calculate Pearson's and Spearman's r-coefficients
print("Pearson: ", ta.pearson("ad", "al"))
print("Spearman: ", ta.spearmanr("ad", "al"))

# Show a plot of the correlation
# Note that the plotCorrEnh method can also
# be used, which is useful in the case of long
# chains.
ta.plotCorr(parsList=["ad", "al"])
ta.plotCorrEnh(parsList=["ad", "al"])
ta.show()

Obtain lowest-deviance, mean, and median set of parameters

This example demonstrates how to obtain certain parameter sets and use them with a model.

from __future__ import print_function, division
from PyAstronomy import funcFit as fuf
import matplotlib.pyplot as plt
import numpy as np

# Create an instance of TraceAnalysis
# telling it which file to use
ta = fuf.TraceAnalysis("mcmcTA.tmp")

# Use the burn-in from the previous example
ta.setBurn(500)

# Find sets of parameters
# First, the lowest deviance set
lds, index = ta.parameterSet(prescription="lowestDev")
print("Lowest deviance set: ", lds)
print(" at chain index: ", index)
means = ta.parameterSet(prescription="mean")
print("Set of mean values: ", means)
medians = ta.parameterSet(prescription="median")
print("Set of median values: ", medians)

# Create Voigt model and plot the models belonging
# to the lowest deviance, mean, and median parameter
# set.
vp = fuf.Voigt1d()
# Generate the model wavelength axis
(continues on next page)
x = np.linspace(5490., 5510., 200)
# Calculate and plot the models
vp.assignValues(lds)
plt.plot(x, vp.evaluate(x), 'b.-')
vp.assignValues(means)
plt.plot(x, vp.evaluate(x), 'r.-')
vp.assignValues(medians)
plt.plot(x, vp.evaluate(x), 'g.-')
plt.show()

Other useful visualization utilities

This example shows a couple of other useful routines to visualize and analyze the Markov-Chains.

```python
from PyAstronomy import funcFit as fuf

# Create an instance of TraceAnalysis
# telling it which file to use
ta = fuf.TraceAnalysis("mcmcTA.tmp")

# Use the burn-in from the previous example
# and its distribution.
# Combine trace and distribution
# Plot correlations
```

Basic models

Common fitting models

Several often needed models are shipped with funcFit.

Common fitting model available

- Gaussian profile (class `GaussFit1d`)
- Multicomponent Gaussian (class `MultiGauss1d`)
- Cauchy-Lorentz profile (class `CauchyLorentz1d`)
- Voigt profile (class `Voigt1d`)
- Multicomponent Voigt (class `MultiVoigt1d`)
- Sine wave (class `SinusFit1d`)

248 Chapter 2. PyAstronomy’s sub-packages
- Exponential decay (class `ExpDecayFit1d`)
- Polynomial of degree $n$ (class `PolyFit1d`)
- Hyperbolic secant (class `Sech1d`)
- Constant (class `ConstantFit1d`)
- 2d Gaussian (class `GaussFit2d`)
- Multicomponent 2d Gaussian (class `MultiGauss2d`)

The use of these models is demonstrated in the tutorial.

**Single Gaussian with linear continuum**

class `PyAstronomy.funcFit.GaussFit1d`

Implements a one dimensional Gaussian.

The functional form is:

$$A \sqrt{\frac{2\pi}{2\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} + x \times \text{lin} + \text{off}$$

Here, `lin` and `off` denote the linear and the offset term.

*Fit parameters:*
- $A$ - Amplitude (the area of the Gaussian)
- $\mu$ - Center of the Gaussian
- $\sigma$ - Standard deviation
- $\text{off}$ - Offset
- $\text{lin}$ - Linear term

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MCMCautoParameters(ranges[, picky, ...])</code></td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td><code>addConditionalRestriction(*args)</code></td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td><code>assignValue(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>assignValues(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</code></td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td><code>availableParameters()</code></td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td><code>delRestriction(parName)</code></td>
<td>Delete restriction</td>
</tr>
<tr>
<td><code>description([parenthesis])</code></td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td><code>errorConfInterval(par[, dstat, statTol, ...])</code></td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td><code>evaluate(x)</code></td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td><code>fit(x, y[, yerr, X0, minAlgo, mAA, ...])</code></td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td><code>fitEMCEE([x, y, yerr, nwalker, priors, ...])</code></td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td><code>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</code></td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td><code>freeParamNames()</code></td>
<td>Get the names of the free parameters.</td>
</tr>
</tbody>
</table>
Table 34 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parametersSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

Multicomponent Gaussian with linear continuum

class PyAstronomy.funcFit.MultiGauss1d(n)
A multicomponent Gaussian with a single linear continuum component.

The parameters are the same as for the GaussFit1d, except that all receive a number specifying the Gaussian component to which they belong. Therefore, they are, e.g., named $A1$, $mu2$, and so on, only $off$ and $lin$ remain unnumbered.

Parameters

- $n$ [int] The number if Gaussian components.

Methods

- MCMCautoParameters(ranges[, picky,...]) Convenience function to generate parameters for MCMC fit.
- addConditionalRestriction(*args) Define a conditional restriction.
- assignValue(specval) Assign new values to variables.
- assignValues(specval) Assign new values to variables.
- autoFitMCMC(x, y, ranges[, picky, stepsize, ...]) Convenience function to using auto-generated sampling parameters in MCMC.
- availableParameters() Provides a list of existing parameters.
- delRestriction(parName) Delete restriction
- description([parenthesis]) Returns a description of the model based on the names of the individual components.

Continued on next page
Table 35 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorConfInterval(par[, dstat, statToI, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evalComponent(x, p)</td>
<td>Evaluate the model considering only a single component.</td>
</tr>
<tr>
<td>evaluate(x)</td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Limits, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

**Cauchy-Lorentz with linear continuum**

**class** PyAstronomy.funcFit.CauchyLorentz1d

A Cauchy-Lorentz profile

**Fit parameters:**

- $A$ - Amplitude
- $g$ - Scale parameter (usually gamma)
- $\mu$ - Center
- $off$ - A constant offset
- $lin$ - A linear contribution

**Methods**
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(x)</td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAa, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

**Voigt with linear continuum**

**Class** `PyAstronomy.funcFit.Voigt1d`

Implements a Voigt profile (convolution of Cauchy-Lorentz and Gaussian distribution).

**Note:** The profile is implemented so that \( a_l \) is half the FWHM of the Cauchy-Lorentz distribution.

**Fit parameters:**
• \( A \) - Area under the curve
• \( al \) - Scale parameter of the Cauchy-Lorentz distribution (half its FWHM)
• \( ad \) - The width (standard deviation) of the Gaussian (usually called sigma)
• \( mu \) - Center
• \( off \) - Constant offset
• \( lin \) - Linear contribution

Notes

The Voigt profile \( V \) is defined as the convolution

\[
V(x) = A \int G(x')L(x - x')dx'
\]

of a Gaussian distribution

\[
G = \frac{1}{\sqrt{2\pi ad}} \exp\left(-\frac{(x - mu)^2}{2 ad^2}\right)
\]

and a Cauchy-Lorentz distribution

\[
L = \frac{al}{\pi ((x - mu)^2 + al^2)}.
\]

We here take into account an additional offset and linear term so that

\[
V'(x) = V(x) + (lin \cdot x + off).
\]


Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FWHM()</td>
<td>Calculates an approximation of the FWHM.</td>
</tr>
<tr>
<td>MCMCAutoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(x)</td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 37 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

### Multicomponent Voigt profile with linear continuum

**class** PyAstronomy.funcFit.MultiVoigt1d(n)

Multicomponent Voigt with a single linear continuum component.

The parameters are the same as for Voigt1d, except that all are extended by a number specifying the component to which they belong. Therefore, they read, e.g., $A1$, $mu2$, and so on; only $off$ and $lin$ remain unnumbered.

**Parameters**

- **n** [int] The number of Voigt components.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCAutoParameters(ranges[, picky,...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize,...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 38 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errorConfInterval(par[, dstat, statTol, ...])</code></td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td><code>evalComponent(x, p)</code></td>
<td>Evaluate the model considering only a single component.</td>
</tr>
<tr>
<td><code>evaluate(x)</code></td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td><code>fit(x, y[, yerr, X0, minAlgo, mAa, ...])</code></td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td><code>fitEMCEE([x, y, yerr, nwalker, priors, ...])</code></td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td><code>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</code></td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td><code>freeParamNames()</code></td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td><code>freeParameters()</code></td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td><code>freeze(specifiers)</code></td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td><code>frozenParameters()</code></td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td><code>getRelationsOf(specifier)</code></td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td><code>getRestrictions()</code></td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td><code>hasVariable(specifier)</code></td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td><code>numberOfFreeParams()</code></td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td><code>parameterSummary([toScreen, prefix, sorting])</code></td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td><code>parameters()</code></td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td><code>relate(dependentVar, independentVars[, func])</code></td>
<td>Define a relation.</td>
</tr>
<tr>
<td><code>removeConditionalRestriction(*args)</code></td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td><code>renameVariable(oldName, newName)</code></td>
<td>Change name of variable.</td>
</tr>
<tr>
<td><code>restoreState(resource)</code></td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td><code>saveState(*args, **kwargs)</code></td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td><code>setObjectiveFunction([miniFunc])</code></td>
<td>Define the objective function.</td>
</tr>
<tr>
<td><code>setPenaltyFactor(penalFac)</code></td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td><code>setRestriction(restricts)</code></td>
<td>Define restrictions.</td>
</tr>
<tr>
<td><code>setRootName(root[, rename])</code></td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td><code>showConditionalRestrictions(**kwargs)</code></td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><code>steppar(pars, ranges[, extractFctVal, quiet])</code></td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><code>thaw(specifiers)</code></td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

**evalComponent** \((x, p)\)
Evaluate the model considering only a single component.

**Parameters**
- \(x\) [array] The abscissa.
- \(p\) [int] Component number (starts with one).

**Returns**

**Single component model** [array] The model considering a single component. Note that the linear continuum is included.

**evaluate** \((x)\)
Evaluates the model for current parameter values.

**Parameters**
- \(x\) [array] Specifies the points at which to evaluate the model.
Sine fit

class PyAstronomy.funcFit.SinusFit1d

Implements a sinusoidal function of the form: \( A \times \sin(2\pi \times (\nu \times t + \phi)) + \text{off} \).

Fit parameters:

- \( A \) - Amplitude
- \( \nu \) - Frequency (= 1/Period)
- \( \phi \) - Phase
- \( \text{off} \) - Offset

Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(x)</td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 39 – continued from previous page

- `showConditionalRestrictions(**kwargs)` Show conditional restrictions.
- `steppar(pars, ranges[, extractFctVal, quiet])` Allows to step a parameter through a specified range.
- `thaw(specifiers)` Consider variables fixed.
- `untie(parName[, forceFree])` Remove all relations of parameter `parName`, i.e., the parameter is not dependend on other parameters.
- `updateModel()` Recalculate the model using current settings.

Exponential decay

**class PyAstronomy.funcFit.ExpDecayFit1d**

Implements an exponential decay function of the form $A \cdot \exp\left( -\frac{(t-t_0)}{\tau} \right) + \text{off}$.

**Fit parameters:**
- $A$ - Amplitude
- $\tau$ - Mean lifetime (=1/decay rate)
- $t_0$ - Onset time
- $\text{off}$ - Continuum offset

**Methods**

- `MCMCAutoParameters(ranges[, picky, ...])` Convenience function to generate parameters for MCMC fit.
- `addConditionalRestriction(*args)` Define a conditional restriction.
- `assignValue(specval)` Assign new values to variables.
- `assignValues(specval)` Assign new values to variables.
- `autoFitMCMC(x, y, ranges[, picky, stepsize, ...])` Convenience function to using auto-generated sampling parameters in MCMC.
- `availableParameters()` Provides a list of existing parameters.
- `delRestriction(parName)` Delete restriction
- `description([parenthesis])` Returns a description of the model based on the names of the individual components.
- `errorConfInterval(par[, dstat, statTol, ...])` Calculate confidence interval for a parameter.
- `evaluate(x)` Evaluates the model for current parameter values.
- `fit(x, y[, yerr, X0, minAlgo, mAA, ...])` Carries out a fit.
- `fitEMCEE([x, y, yerr, nwalker, priors, ...])` MCMC sampling using emcee package.
- `fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])` Carry out MCMC fit/error estimation.
- `freeParamNames()` Get the names of the free parameters.
- `freeParameters()` Get names and values of free parameters.
- `freeze(specifiers)` Consider variables free to float.
- `frozenParameters()` Get names and values of frozen parameters.
- `getRelationsOf(specifier)` Return relations of a variable.
- `getRestrictions()` Get all restrictions.
- `hasVariable(specifier)` Determine whether the variable exists.
- `numberOfFreeParams()` Get number of free parameters.
- `parameterSummary([toScreen, prefix, sorting])` Writes a summary of the parameters in text form.
- `parameters()` Obtain parameter names and values.
- `relate(dependentVar, independentVars[, func])` Define a relation.

Continued on next page
Table 40 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

Polynomial of degree n

class PyAstronomy.funcFit.Poly1d(degree, xoff=0.0)
Implements a polynomial fit.

Fit parameters:
- cn - Here n is a number indicating degree (e.g., c0 + c1*x + c2*x**2 ...)

If xoff is specified, the polynomial will be evaluated at the points x-xoff. This can be useful to suppress correlation.

Methods

MCMCAutoParameters(ranges[, picky,...]) Convenience function to generate parameters for MCMC fit.
addConditionalRestriction(*args) Define a conditional restriction.
asNPPoly() Construct a numpy.poly1d object from the coefficients.
assignValue(specval) Assign new values to variables.
assignValues(specval) Assign new values to variables.
autoFitMCMC(x, y, ranges[, picky, stepsize,...]) Convenience function to using auto-generated sampling parameters in MCMC.
availableParameters() Provides a list of existing parameters.
delRestriction(parName) Delete restriction
description([parenthesis]) Returns a description of the model based on the names of the individual components.
errorConfInterval(par[, dstat, statTol,...]) Calculate confidence interval for a parameter.
evaluate(x) Evaluates the model for current parameter values.
fit(x, y[, yerr, X0, minAlgo, mAA,...]) Carries out a fit.
fitEMCEE([x, y, yerr, nwalker, priors,...]) MCMC sampling using emcee package.
fitMCMC(x, y, X0, Lims, Steps[, yerr,...]) Carry out MCMC fit/error estimation.
freeParamNames() Get the names of the free parameters.
freeParameters() Get names and values of free parameters.
freeze(specifiers) Consider variables free to float.
Table 41 – continued from previous page

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary(toScreen, prefix, sorting)</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction(miniFunc)</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

Hyperbolic secant

**class** PyAstronomy.funcFit.Sech1d

Implements a one dimensional hyperbolic secant

The functional form is:

\[
\frac{2A}{e^{(x-mu)/w} + e^{-(x-mu)/w}} + x \times \text{lin} + \text{off}
\]

Here, \(\text{lin}\) and \(\text{off}\) denote the linear and the offset term.

**Note:** The area under the curve is given by \(\pi \times A\)

**Fit parameters:**

- \(A\) - Amplitude (maximum/minimum of the curve, not area)
- \(mu\) - Center of the hyperbolic secant
- \(w\) - Width parameter
- \(off\) - Offset
- \(lin\) - Linear term

**Methods**
### MCMCautoParameters

Convenience function to generate parameters for MCMC fit.

### addConditionalRestriction(*args)

Define a conditional restriction.

### assignValue(specval)

Assign new values to variables.

### assignValues(specval)

Assign new values to variables.

### autoFitMCMC(x, y, ranges[, picky, stepsize, ...])

Convenience function to using auto-generated sampling parameters in MCMC.

### availableParameters()

Provides a list of existing parameters.

### delRestriction(parName)

Delete restriction.

### description([parenthesis])

Returns a description of the model based on the names of the individual components.

### errorConfInterval(par[, dstat, statToL, ...])

Calculate confidence interval for a parameter.

### evaluate(x)

Evaluates the model for current parameter values.

### fit(x, y[, yerr, X0, minAlgo, mAA, ...])

Carries out a fit.

### fitEMCEE([x, y, yerr, nwalker, priors, ...])

MCMC sampling using emcee package.

### fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])

Carry out MCMC fit/error estimation.

### freeParamNames()

Get the names of the free parameters.

### freeParameters()

Get names and values of free parameters.

### freeze(specifiers)

Consider variables free to float.

### frozenParameters()

Get names and values of frozen parameters.

### getRelationsOf(specifier)

Return relations of a variable.

### getRestrictions()

Get all restrictions.

### hasVariable(specifier)

Determine whether the variable exists.

### numberOfFreeParams()

Get number of free parameters.

### parameterSummary([toScreen, prefix, sorting])

Writes a summary of the parameters in text form.

### parameters()

Obtain parameter names and values.

### relate(dependentVar, independentVars[, func])

Define a relation.

### removeConditionalRestriction(*args)

Remove an existing conditional constraint.

### renameVariable(oldName, newName)

Change name of variable.

### restoreState(resource)

Restores parameter values from file or dictionary.

### saveState(*args, **kwargs)

Save the state of the fitting object.

### setObjectiveFunction([miniFunc])

Define the objective function.

### setPenaltyFactor(penalFac)

Change the penalty factor.

### setRestriction(restricts)

Define restrictions.

### setRootName(root[, rename])

Define the root name of the model.

### showConditionalRestrictions(**kwargs)

Show conditional restrictions.

### steppar(pars, ranges[, extractFctVal, quiet])

Allows to step a parameter through a specified range.

### thaw(specifiers)

Consider variables fixed.

### untie(parName[, forceFree])

Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.

### updateModel()

Recalculate the model using current settings.

## Constant

**class** PyAstronomy.funcFit.ConstantFit1d

Implements a constant model.

### Fit parameters:

- c - The constant.
Methods

MCMCAutoParameters(ranges[, picky, ...]) Convenience function to generate parameters for MCMC fit.
addConditionalRestriction(*args) Define a conditional restriction.
assignValue(special) Assign new values to variables.
assignValues(specval) Assign new values to variables.
autoFitMCMC(x, y, ranges[, picky, stepsize, ...]) Convenience function to using auto-generated sampling parameters in MCMC.
availableParameters() Provides a list of existing parameters.
delRestriction(parName) Delete restriction
description([parenthesis]) Returns a description of the model based on the names of the individual components.
errorConfInterval(par[, dstat, statTol, ...]) Calculate confidence interval for a parameter.
evaluate(x) Evaluates the model for current parameter values.
fit(x, y[, yerr, X0, minAlgo, mA, ...]) Carries out a fit.
fitEMCEE([x, y, yerr, nwalker, priors, ...]) MCMC sampling using emcee package.
fitMCMC(x, y, X0, Lims, Steps[, yerr, ...]) Carry out MCMC fit/error estimation.
freeParamNames() Get the names of the free parameters.
freeParameters() Get names and values of free parameters.
freeze(specifiers) Consider variables free to float.
frozenParameters() Get names and values of frozen parameters.
getRelationsOf(specifier) Return relations of a variable.
getRestrictions() Get all restrictions.
hasVariable(specifier) Determine whether the variable exists.
numberOfFreeParams() Get number of free parameters.
parameterSummary([toScreen, prefix, sorting]) Writes a summary of the parameters in text form.
parameters() Obtain parameter names and values.
relate(dependentVar, independentVars[, func]) Define a relation.
removeConditionalRestriction(*args) Remove an existing conditional constraint.
renameVariable(oldName, newName) Change name of variable.
restoreState(resource) Restores parameter values from file or dictionary.
saveState(*args, **kwargs) Save the state of the fitting object.
setObjectiveFunction([miniFunc]) Define the objective function.
setPenaltyFactor(penalFac) Change the penalty factor.
setRestriction(restricts) Define restrictions.
setRootName(root[, rename]) Define the root name of the model.
showConditionalRestrictions(**kwargs) Show conditional restrictions.
steppar(pars, ranges[, extractFctVal, quiet]) Allows to step a parameter through a specified range.
thaw(specifiers) Consider variables fixed.
untie(parName[, forceFree]) Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.
updateModel() Recalculate the model using current settings.

Two-dimensional Gaussian

class PyAstronomy.funcFit.GaussFit2d

Implements a two dimensional Gaussian.

Expects a coordinate array to evaluate model.
The functional form is:

\[
\frac{A}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)} \left(\frac{(x-\mu_x)^2}{\sigma_x^2} + \frac{(y-\mu_y)^2}{\sigma_y^2} - \frac{2\rho(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y}\right)\right)
\]

Here, \textit{lin} and \textit{off} denote the linear and the offset term.

**Fit parameters:**

- \(A\) - Amplitude (the area of the Gaussian)
- \(\mu_x\) - Center of the Gaussian (x-axis)
- \(\mu_y\) - Center of the Gaussian (y-axis)
- \(\sigma_x\) - Standard deviation (x-axis)
- \(\sigma_y\) - Standard deviation (y-axis)
- \(\rho\) - Correlation

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters([ranges[, picky, ...]])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(co)</td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 44 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(<strong>kwargs</strong>)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

**Multicomponent two-dimensional Gaussian**

```python
class PyAstronomy.funcFit.MultiGauss2d(n)
```

Implements a multicomponent, two dimensional Gaussian.

Expects a coordinate array to evaluate model.

The functional form is:

\[
A \frac{1}{2\pi\sigma_x\sigma_y \sqrt{1 - \rho^2}} \exp \left( -\frac{1}{2(1 - \rho^2)} \left( \frac{(x - \mu_x)^2}{\sigma_x^2} + \frac{(y - \mu_y)^2}{\sigma_y^2} - \frac{2\rho(x - \mu_x)(y - \mu_y)}{\sigma_x\sigma_y} \right) \right)
\]

**Parameters**

- **n** [int] The number of Gaussian components.

**Fit parameters**:  
- \(A\) - Amplitude (the area of the Gaussian)  
- \(mu_x\) - Center of the Gaussian (x-axis)  
- \(mu_y\) - Center of the Gaussian (y-axis)  
- \(sig_x\) - Standard deviation (x-axis)  
- \(sig_y\) - Standard deviation (y-axis)  
- \(rho\) - Correlation  
- \(off\) - Constant offset

**Methods**

- **MCMCautoParameters(ranges[, picky,...])** Convenience function to generate parameters for MCMC fit.  
- **addConditionalRestriction(*args)** Define a conditional restriction.  
- **assignValue(specval)** Assign new values to variables.  
- **assignValues(specval)** Assign new values to variables.  
- **autoFitMCMC(x, y, ranges[, picky, stepsize,...])** Convenience function to using auto-generated sampling parameters in MCMC.  
- **availableParameters()** Provides a list of existing parameters.  
- **delRestriction(parName)** Delete restriction  
- **description([parenthesis])** Returns a description of the model based on the names of the individual components.

Continued on next page
### Table 45 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errorConfInterval(par[, dstat, statTol, ...])</code></td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td><code>evaluate(co)</code></td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td><code>fit(x, y[, xerr, X0, minAlgo, mAA, ...])</code></td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td><code>fitEMCEE([x, y, xerr, nwalker, priors, ...])</code></td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td><code>fitMCMC(x, y, X0, Lim, Steps[, xerr, ...])</code></td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td><code>freeParamNames()</code></td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td><code>freeParameters()</code></td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td><code>freeze(specifiers)</code></td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td><code>frozenParameters()</code></td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td><code>getRelationsOf(specifier)</code></td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td><code>getRestrictions()</code></td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td><code>hasVariable(specifier)</code></td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td><code>numberOfFreeParams()</code></td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td><code>parameterSummary([toScreen, prefix, sorting])</code></td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td><code>parameters()</code></td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td><code>relate(dependentVar, independentVars[, func])</code></td>
<td>Define a relation.</td>
</tr>
<tr>
<td><code>removeConditionalRestriction(*args)</code></td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td><code>renameVariable(oldName, newName)</code></td>
<td>Change name of variable.</td>
</tr>
<tr>
<td><code>restoreState(resource)</code></td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td><code>saveState(*args, **kwargs)</code></td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td><code>setObjectiveFunction([miniFunc])</code></td>
<td>Define the objective function.</td>
</tr>
<tr>
<td><code>setPenaltyFactor(panFac)</code></td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td><code>setRestriction(restricts)</code></td>
<td>Define restrictions.</td>
</tr>
<tr>
<td><code>setRootName(root[, rename])</code></td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td><code>showConditionalRestrictions(**kwargs)</code></td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><code>stepPar(pars, ranges[, extractFctVal, quiet])</code></td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><code>thaw(specifiers)</code></td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

### Demonstration models

Models mainly used for demonstration purposes.

#### Simple circular orbit

**Class** `PyAstronomy.funcFit.Circle2d`  
Implements a simple, two-dimensional circular orbit.

The functional form is:

\[(x, y) = (r \cos(\omega(t - t_0)), r \sin(\omega(t - t_0)))\]

where \(\omega = 2\pi / \text{Period}\).

**Fit parameters:**

- `r` - Radius of the circle
- `per` - Period of orbit
- `t0` - Starting time of orbit
Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCAutoParameters(ranges[, picky,...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction.</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(t)</td>
<td>Evaluates the model for current parameter values.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAa, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

Internal and external fitters

Internal and external optimizers

A “fitter” or “optimizer” is the algorithm used to carry out the actual optimization. In funcFit, there are “internal” and “external” optimizers. The internal optimizers are those invoked by a call to the fit-method of the OneDFit object. External optimizers take a OneDFit object as an object and carry out the fitting without actually becoming a part of
the object.

While, quite generally, internal optimizers are very convenient, it may be a little easier to exert control over the screws determining the details of the algorithm, if it needs to be tweaked to solve a specific problem.

Internal optimizers

Internal optimizers are invoked by OndeDFit’s fit method. Basically, an internal optimizer is a callable object, which carries out the optimization when called. A base class for internal optimizers is provided by funcFit.

External optimizers

An external fitter is a fitting algorithm not implemented as a part of funcFit’s OneDFit class. Further algorithms can thus be supported, which may give, e.g., stricter control over individual steps or internal settings.

A convenient way to pass data to the external fitters is the “funcFit data set” fufDS.

List of internal and external optimizers

Internal optimizers

Base class for internal optimizers

```
class PyAstronomy.funcFit.IFitterBase
  Base class for internal fitter.

  Methods

  __call__(*args, **kwargs) Wrapper around the actual fit method.
  fit(minifunc, x0) Carry out the minimization.
  getObjFuncValue() Access value of objective function.
```

```
fit (minifunc, x0)
  Carry out the minimization.

  Parameters

  minifunc [callable] Objective function.
  x0 [list] Starting values

  Returns

  Result [list or tuple] First item is a list of the best-fit values and second item is the value of the objective function.
```

```
getObjFuncValue ()
  Access value of objective function.

  Returns
```
value [float] Value of objective function.

ScipyFMIN

class PyAstronomy.funcFit.ScipyFMIN(*args, **kwargs)
    Wrapper around scipy.optimize.fmin.

Methods

__call__(*args, **kwargs)  Wrapper around the actual fit method.
fit(miniFunc, x0, *fminpars, **fminargs)  Wrapper around scipy.optimize.fmin.
getObjFuncValue()  Access value of objective function.

fit (miniFunc, x0, *fminpars, **fminargs)
    Wrapper around scipy.optimize.fmin.

FuFNM

FuFNM is based on the Nelder-Mead-Simplex implemenetation of funcFit (see NelderMead()).

class PyAstronomy.funcFit.FuFNM(*args, **kwargs)
    Downhill-Simplex algorithm for minimization.

This implementation is based on the publication: Nelder and Mead, The Computer Journal 7, 308-313, 1965 (NM 1965)

Halting criterion

The default stop criterion is the one used by NM 1965. In particular, the value
\[
\sqrt{\sum(y - y_i)/n}
\]
is calculated. If it falls below the limit defined by the attribute nmCritLim, the iteration stops.

Parameters

abg [tuple of three floats, optional] Values for alpha, beta, and gamma.
isw [float, optional] Initial step width for simplex.
critlim [float, optional] Critical limit for stopping criterion.

Attributes

alpha, beta, gamma [float] The reflection-, expansion-, and contraction-coefficients. The default values (after NM 1965) are 1.0, 0.5, and 2.0. These coefficients control the modification of the simplex.

initialStepWidthFac [float] This factor determines how the initial simplex is calculated. The first simplex point is the starting value, the others are constructed by adding a fraction defined by this factor to the starting value. The default is 0.05.
nmCritLim [float] Critical value for the NM 1965 stopping criterion. The default is 1e-8.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong>(*args, **kwargs)</td>
<td>Wrapper around the actual fit method.</td>
</tr>
<tr>
<td>fit(miniFunc, x0, *fminpars, **fminargs)</td>
<td>Wrapper around funcFit’s NelderMead implementation.</td>
</tr>
<tr>
<td>getObjFuncValue()</td>
<td>Access value of objective function.</td>
</tr>
</tbody>
</table>

**fit (miniFunc, x0, *fminpars, **fminargs)**

Wrapper around funcFit’s NelderMead implementation.

See the implementation of the fit method of NelderMead() for the available keyword arguments (fminargs).

External optimizers

Nelder-Mead downhill simplex

**class** PyAstronomy.funcFit.NelderMead(abg=(1.0, 0.5, 2.0), isw=0.05, critlim=1e-08)

Downhill-Simplex algorithm for minimization.

This implementation is based on the publication: Nelder and Mead, The Computer Journal 7, 308-313, 1965 (NM 1965)

**Halting criterion**

The default stop criterion is the one used by NM 1965. In particular, the value

\[ \sqrt{\frac{\sum(y - y_i)^2}{n}} \]

is calculated. If it falls below the limit defined by the attribute nmCritLim, the iteration stops.

**Parameters**

- **abg** [tuple of three floats, optional] Values for alpha, beta, and gamma.
- **isw** [float, optional] Initial step width for simplex.
- **critlim** [float, optional] Critical limit for stopping criterion.

**Attributes**

- **alpha, beta, gamma** [float] The reflection-, expansion-, and contraction-coefficients. The default values (after NM 1965) are 1.0, 0.5, and 2.0. These coefficients control the modification of the simplex.
- **initialStepWidthFac** [float] This factor determines how the initial simplex is calculated. The first simplex point is the starting value, the others are constructed by adding a fraction defined by this factor to the starting value. The default is 0.05.
- **nmCritLim** [float] Critical value for the NM 1965 stopping criterion. The default is 1e-8.

**Methods**

**fit(m, ds[, objf, initDelta, maxIter, ...])**

Carry out the model fit.
_initSimplex_(m, initDelta)

Define the initial simplex.

Parameters

  m [Instance of OneDFit] The model.

  initDelta [dictionary] Maps parameter name to the initial step width for the simplex.

_step_(m)

Proceed one step of the simplex algorithm.

_stopNM1965()

Apply the NM 1965 stopping criterion.

Returns

  Reached [boolean] True, if the stopping criterion has been reached.

_fit_(m, ds, objf='chisqr', initDelta=None, maxIter=10000.0, callback=None, nmCritLim=None)

Carry out the model fit.

After the iteration, the _iterCount_ attribute contains the number of iterations. The _maxIterReached_ attribute flag is False, if the maximum number of iterations has not been reached and True otherwise.

Parameters

  m [Instance of OneDFit] The model to be fitted.

  ds [Instance of FufDS] The data.

  objf [string] The objective function to be used. Possible choices are “chisqr” (default), “sqrdf”, and “cash79”.

  initDelta [dictionary, optional] A dictionary mapping parameter names to the initial step width. This can be very useful, if the starting values are zero or very small. The here defined step will be added to the starting value to construct the simplex.

  maxIter [int, optional] The maximum number of iterations. The default is 10000.

  nmCritLim [float, optional] Critical value for stopping criterion. The default is 1e-8.

  callback [callable, optional] If not None, “callback” will be called with the three parameters: number of iteration (int), current best parameter set (array), and current simplex (array).

Returns

  Best-fit values [dictionary] Maps parameter name to the best-fit value.

Example: Application of the NelderMead class

```python
from PyAstronomy import funcFit as fuf
import numpy as np

# Construct the fitter
nm = fuf.NelderMead()

# Get a GaussFit object ...
gf = fuf.GaussFit1d()
gf.thaw(["A", "mu"])
# ... and define some initial values.
```

(continues on next page)
gf[“A”] = 5.0
gf[“mu”] = 2.0
gf[“sig”] = 1.0

# Construct some “data” ...
x = np.arange(100)/20.
y = gf.evaluate(x) + np.random.normal(0.0, 0.08, len(x))
yerr = np.ones(len(x)) * 0.08
# ... and define a “data set” object
ds = fuf.FufDS(x, y, yerr=yerr)

# On purpose, we vail our knowledge about the correct
# parameters
gf[“A”] = 1.0
gf[“mu”] = 1.0

# We carry out the fit. In particular, the squared
# distance between the model and the data is minimized
# (sqrdiff).
bfpars = nm.fit(gf, ds, objf=“sqrdiff”, maxIter=100)
# Have a look at the result
print “Best-fit parameters: “, bfpars
print
# Have a look at the entire fit object
gf.parameterSummary()

print
# Again, we change the starting parameters. Note that ...
gf[“A”] = 1.0
# … mu has been set to zero.
gf[“mu”] = 0.0

# To allow the construction of an appropriate start simplex,
# we need to provide an “initDelta” for mu, which should
# reflect the typical scale of the problem.
bfpars = nm.fit(gf, ds, objf=“chisqr”, initDelta={“mu”: 0.1})
print “Second fit, best-fit parameters: “, bfpars

API, parameters and further functionality

Parameter Management

class PyAstronomy.funcFit.Params(paramNames)

Manage a set of parameters.

This class provides a framework for managing parameter values and several associated aspects such as restric-
tions and relation. Parameter management is implemented in view of later use by a fit routine.

Parameters

paramNames [list of strings] Specifies the names of the parameters to be managed.
Notes

Different models make different demands on the variables. For example, only certain ranges may be valid, some are constant and others not, or there may be a functional dependence between different variables. This class provides a framework to manage a parameter set appropriately.

Depending on what kind of model is currently adapted, the number, names, allowed ranges, and interdependencies of variables can differ. On creation, this class is given a list with parameter names to manage. Those can then be assigned values. Parameters can be “thawed”, i.e., regarded free during the fit process, or frozen. The allowed ranges can be restricted either on one or both sides, and interdependencies can be defined via the `relate` method.

Attributes

__params [dict] A dictionary containing entries of the kind [variable-name:value]. The __params dictionary may only be access by the `assignValue` function to prevent causing inconsistencies, which may occur especially if relations exist.

paramNum [dict] A dictionary assigning every parameter name to a number (e.g., paramNum[2] = “XYZ”). Such a numbering is important to guarantee the correct order of variables in the fitting process. The ordering is the same as the order in which the constructor obtains the parameter names.

isFree [dict] A dictionary associating each parameter with a boolean saying whether it shall be considered a free parameter during optimization.

isRestricted [dict] A dictionary associating each parameter with a boolean saying whether there is a restriction to the allowed range (see restrictions).

restrictions [dict] A dictionary containing entries of the form [parName:[lower, upper]. Here ‘lower’ and ‘upper’ are floats defining lower and upper limits for the variable’s value.

relations [dict] Parameters may be interrelated, e.g., A = B + C. This dictionary holds the definition of such relations. In particular, the entries have the form {“b”:[ “a”, func, “b”, “c”], […]}. This means that a = func(b, c). func is a function pointer. Whenever b is assigned a new value, 'a' has to be updated as well. Note that it is important that the independent variables “know” about the relation, because if their value changes, the value of the dependent variable (in this case a) has to be updated.

conditionalRestrictions [dict] A dictionary holding the ‘conditional restrictions’, i.e., complex restrictions, which may, e.g., depend in the values of other parameters. The dictionary key is a unique ID generated, when a conditional restriction is added. For each key, the dictionary holds a tuple of which the first entry is a list holding the names of the parameters on which the conditional restrictions depends and the second is a callable, which is called with the values of those parameters specified in the first entry. The callable must return a float that specifies the penalty (or reward) depending on the given parameter values. Because conditional restrictions are referred to using a unique ID, their name (i.e., ID) does not change if models are combined.

Methods

addConditionalRestriction(pars, func) Define a conditional restriction.

applyConditionalRestrictions([fullout]) Apply all conditional restrictions.

assignValue(namval) Define new parameter values.

availableParameters() Provides a list of existing parameters.

delRestriction(parName) Delete restriction

Continued on next page
Table 51 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(name)</td>
<td>Freeze parameter(s) (contrary of thaw).</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getFreeParams()</td>
<td>Get values of free parameters.</td>
</tr>
<tr>
<td>getPenalty((penaltyFact))</td>
<td>Get applied penalty for current parameter set.</td>
</tr>
<tr>
<td>getRelationsOf(parName)</td>
<td>Obtain relations for a parameter.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasParam(name)</td>
<td>Check whether parameter exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary((lines, toScreen, prefix, ...))</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(parName1, pars[, func, force])</td>
<td>Apply functional relation between parameters.</td>
</tr>
<tr>
<td>removeConditionalRestriction(id)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameParameter(old, new)</td>
<td>Rename an existing parameter.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState((fn, clobber))</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setFreeParams(X)</td>
<td>Change the values of the free parameters.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Apply restrictions to parameter ranges.</td>
</tr>
<tr>
<td>showConditionalRestrictions([toScreen])</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>thaw(name)</td>
<td>Thaw (regard as free) a parameter.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter “parName”.</td>
</tr>
</tbody>
</table>

addConditionalRestriction (pars, func)

Define a conditional restriction.

Conditional restrictions can be used to modify the behavior in a more complex manner. For instance, penalties can be added to the objective function depending on the relation of one or more parameters.

The given function is evaluated in each iteration and its return value (a float) is added to the objective function (e.g., chi square).

Parameters

pars [list of strings] The names of the parameters the given function needs as input.

func [callable object] This callable object must take the specified parameters (in that exact order) as input. It must return a float, which is added to the value of the objective function.

Returns

identifier [string] A unique ID used to refer to the conditional restriction.

applyConditionalRestrictions (fullout=False)

Apply all conditional restrictions.

Parameters

fullout [boolean, optional] If True, a dictionary holding the values of the individual restrictions is returned. The IDs are used as dictionary keys. The default is False.

Returns

Modification [float] The summed value of the existing conditional restrictions.

Individual values [dictionary, optional] The contributions of the individual conditional restrictions.
assignValue (namval)
Define new parameter values.

Parameters

namval [dict] A dictionary containing [‘name’:’value’] pairs.

availableParameters ()
Provides a list of existing parameters.

Returns

Parameters [list of strings] A list with the names of available parameters.

delRestriction (parName)
Delete restriction

Parameters

parName [string] Name of restricted parameter

freeParamNames ()
Get the names of the free parameters.

Returns

Free parameters [list of strings] The names of the free parameters. The order is determined by the paramNum attribute.

freeParameters ()
Get names and values of free parameters.

Returns

Free parameters: dict Dictionary containing the names and values of all free parameters ({{“parName”:value, . . . }}).

freeze (name)
Freeze parameter(s) (contrary of thaw).

Parameters

name [string or list of strings] The name(s) of the parameter(s) to be frozen.

defrozenParameters ()
Get names and values of frozen parameters.

Returns

Frozen parameters: dict Dictionary containing the names and values of all frozen parameters ({{“parName”:value, . . . }}).

defree (name)
Defreeze parameter(s) (contrary of freeze).

Parameters

name [string or list of strings] The name(s) of the parameter(s) to be defrozen.

getFreeParams ()
Get values of free parameters.

Returns

Values [list of floats] The values of the free parameters. Note that the order is determined by the paramNum attribute.

defgetPenalty (penaltyFact=1e+20)
Get applied penalty for current parameter set.

Parameters

penaltyFact [float, optional] The higher the number the higher the penalty for small deviations (default is 10**20).
Returns

**Penalty** [float] The applied penalty for current parameter set.

**Penalties** [dict] A dictionary with a key for every parameter values for which a bound is violated. The value is the amount by which the bound (upper or lower) is violated.

getRelationsOf *(parName)*
Obtain relations for a parameter.

Parameters

**parName** [string] The name of the parameter of which relations shall be searched.

Returns

**Relations** [list of relations] Those relations in which *parName* is the dependent variable, i.e., *parName* = f(x,y).

getRestrictions ()
Get all restrictions.

Returns

**Restrictions** [dict] Dictionary associating parameter name and restriction (see restrictions attribute).

hasParam *(name)*
Check whether parameter exists.

Parameters

**name** [string] The parameter name.

Returns

**flag** [boolean] True if parameter exists, False otherwise.

numberOfFreeParams ()
Get number of free parameters.

Returns

**n** [int] The number of free parameters (determined by *isFree*).

parameterSummary *(lines=None, toScreen=True, prefix=",", onlyParams=False)*
Writes a summary of the parameters in text form.

Parameters

**lines** [list of strings, optional] If given, the output will be attached to this list.

**toScreen** [boolean, optional] If False, screen output will be suppressed (default is True).

**prefix** [string, optional] A prefix applied to every output line (e.g., ‘#’)

@FIXME - This is antique…
Attaches the text lines
to the lines list if given. If lines is not given, ‘toScreen’ will be set to True
and the result is written to the screen.

Returns either a new list containing the text, or (if given) the ‘lines’ list
with result appended.
A ‘prefix’ may be specified (e.g., ‘#’), which will preceded every line.
parameters ()
Obtain parameter names and values.

Returns
Name-value [dict] A dictionary with the names and values of all parameters ({"par-
Name":value, ... }).

relate(parName1, pars, func=None, force=False)
Apply functional relation between parameters.
Relates parameters, i.e., par1 = func(pars). The two values are related by ‘func’. In order to be tied, all
involved parameters have to be free.

Parameters
parName1 [string] The name of the dependent variable.
pars [list of strings] Names of independent variables.
func [callable] The function that determines the form of the dependence. The default is
“equal”.
force [boolean, optional] Set to True in order to suppress error when the dependent vari-
able is frozen.

removeConditionalRestriction(id)
Remove an existing conditional constraint.

Parameters
id [string] The identifier used to refer to the conditional constraint (returned by addCon-
ditionalRestriction).

renameParameter(old, new)
Rename an existing parameter.

Parameters
old [string] The existing (old) name.
new [string] The new name.

restoreState(resource)
Restores parameter values from file or dictionary.

Parameters
resource [string or dictionary] If string, it is interpreted as filename of a pickle file hold-
ing the data dictionary. If dictionary, it uses the data saved in it; note that a valid data
dictionary is returned by saveState.

saveState(fn=None, clobber=False)
Save the state of the fitting object.
This method collects the parameter values, the applied restrictions, and the information whether param-
eters are free or frozen and saves them to the specified file (if given) using pickle. The saved state can be
restored using the restoreState method.

Note: Unfortunately, “relations” cannot be saved.

Parameters
fn [string, optional] The filename to which the state shall be written. If None, the output will not be written to a file.

clobber [boolean, optional] If True, existing files will be overwritten (default is False).

Returns
Saved data [dict] The dictionary containing the data saved to the file.

setFreeParams \( (X) \)
Change the values of the free parameters.

Parameters
\( X \) [list of floats] Contains the values for all free parameters. Note that the numbering is according to the \textit{paramNum} attribute.

Notes
This method is primarily implemented to be used by fit routines.

setRestriction \( (restricts) \)
Apply restrictions to parameter ranges.

Parameters
\( restricts \) [dict] A dictionary associating name and [lower-bound, upper-bound]. If no boundary shall exist on one side, use ‘None’.

showConditionalRestrictions \( (toScreen=True) \)
Show conditional restrictions.

Parameters
\( toScreen \) [boolean, optional] If True (default), the output is written to stdout.

Returns
Output [list of strings] The output as a list of strings.

thaw \( \text{name} \)
Thaw (regard as free) a parameter.

Parameters
\( \text{name} \) [string or list of strings] The name(s) of the parameter(s) to be thawed.

untie \( (\text{parName}, forceFree=False) \)
Remove all relations of parameter “parName”.
After this operation, the parameter no longer depends on other parameters. Unless foreFree is True, the parameter referred to by “parName” is “frozen”.

Parameters
\( \text{parName} \) [string] The name of the dependent variable, which shall be untied.
\( \text{forceFree} \) [boolean] Set parName to “free” instead of “frozen” if set to True.
Fitting object base class

The OneDFit class provides a convenient interface to fitting algorithms.

```python
class PyAstronomy.funcFit.OneDFit(parList, **kwargs)
The base class for fitting objects.
```

Parameters

- `parList` [list of strings]: Contains the names of the properties defining the model. By default, variables of the same name are used to represent them.

Notes

The purpose of the class

The purpose of this class is to provide a convenient interface to various fitting algorithms. It provides the functionality, which allows for parameter fitting, but does not implement a particular model. The class can be used to fit any kind of model, which has to be implemented in a class, which inherits from the OneDFit class.

Management of fitting parameters

The fitting parameters are managed by a `Params` class instance, which provides a wealth of possibilities to influence the behavior of the parameters during the fitting process. This includes deciding whether a particular parameter is supposed to be a free fitting parameter, applying restrictions to limit the valid range for a parameter, or the introduction of functional dependencies among different parameters.

Properties versus variable names

Each model is described by a number of properties, such as, for example, mass and radius. These may be represented by arbitrarily named variables. Normally, it is convenient to name the variables according to the properties they describe, which is the default behavior. However, in some cases, for example if a model consists of two equal subcomponents, such a naming scheme leads to nonunique variable names, which has to be avoided. Now it is necessary to distinguish between the property and the describing variable. This class uses the `propMap` dictionary, which maps property name to variable name to manage these situations.

Combining fitting objects

Often, it can be convenient to combine a number of simple models to form a new, more complex one. The OneDFit class allows to combine objects using the arithmetic operators `+-*/`, and the power (`**`) operator.

Naming scheme for models

For simple models it is convenient to use a one-to-one mapping between property and variable name. It may, however, become necessary to deviate from this scheme, for example, to keep variable names unique. This class supports the following naming scheme: Each model has a “root name”, which is supposed to be a concise string describing the model (for instance, “Gaussian”). The root name is attached to the property name using an underscore. If a complex model consists of more than one component with the same root name, a component counter, enclosed in parenthesis, is attached to the variable name. A variable name could, for example, look like: “mu_Gaussian(1)”.

Methods to be implemented in a model class

A valid model class inheriting this interface class must provide the following methods

- `__init__()`: The constructor.
Defines the set of properties describing the model.

- **evaluate(x)** - An `evaluate` method.

  This method takes a single argument, `x`, which is an array of points at which the model is to be evaluated. To access the current model parameters, this method should use the `set/getitem` methods. The return value is an array holding the model evaluated at the points given by `x`.

Attributes

- **model** [array] Used by the `updateModel` method to store the evaluated model for current parameter settings. Holds the best-fit model after a call to a fit method.

- **penaltyFactor** [float] The penalty factor used to apply penalties for enforcing restrictions (default = $10^{20}$).

Methods

- **MCMCAutoParameters**(ranges[, picky, ...]) Convenience function to generate parameters for MCMC fit.

- **addConditionalRestriction**(args) Define a conditional restriction.

- **assignValue**(specval) Assign new values to variables.

- **assignValues**(specval) Assign new values to variables.

- **autoFitMCMC**(x, y, ranges[, picky, stepsize, ...]) Convenience function to using auto-generated sampling parameters in MCMC.

- **availableParameters**() Provides a list of existing parameters.

- **delRestriction**(parName) Delete restriction.

- **description**(parenthesis) Returns a description of the model based on the names of the individual components.

- **errorConfInterval**(par[, dstat, statTol, ...]) Calculate confidence interval for a parameter.

- **fit**(x, y[, yerr, X0, minAlgo, mAA, ...]) Carries out a fit.

- **fitEMCEE**(x, y, nwalkers, priors, ...) MCMC sampling using `emcee` package.

- **fitMCMC**(x, y, X0, Lims[, yerr, ...]) Carry out MCMC fit/error estimation.

- **freeParamNames**() Get the names of the free parameters.

- **freeParameters**() Get names and values of free parameters.

- **freeze**(specifiers) Consider variables free to float.

- **frozenParameters**() Get names and values of frozen parameters.

- **getRelationsOf**(specifier) Return relations of a variable.

- **getRestrictions**() Get all restrictions.

- **hasVariable**(specifier) Determine whether the variable exists.

- **numberOfFreeParams**() Get number of free parameters.

- **parameterSummary**(toList, prefix, sorting) Writes a summary of the parameters in text form.

- **parameters**() Obtain parameter names and values.

- **relate**(dependentVar, independentVars[, func]) Define a relation.

- **removeConditionalRestriction**(args) Remove an existing conditional constraint.

- **renameVariable**(oldName, newName) Change name of variable.

- **restoreState**(resource) Restores parameter values from file or dictionary.

- **saveState**(args, **kwargs) Save the state of the fitting object.

- **setObjectiveFunction**(miniFunc) Define the objective function.

- **setPenaltyFactor**(penalFac) Change the penalty factor.

- **setRestriction**(restricts) Define restrictions.

Continued on next page
Table 52 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>setRootName</strong>(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td><strong>showConditionalRestrictions</strong>(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><strong>steppar</strong>(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><strong>thaw</strong>(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><strong>untie</strong>(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.</td>
</tr>
<tr>
<td><strong>updateModel()</strong></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

**MCMCautoParameters**(ranges, picky=True, stepsize=0.01, setRestrictionsFromPriors=False)

Convenience function to generate parameters for MCMC fit.

This function prepares the $X_0$, $Lims$, and $Steps$ dictionaries needed to call $fitMCMC()$.

For $X_0$, the current parameter values are used. $Lims$ is constructed using the $ranges$ parameter, and $Steps$ is defined on the basis of the $stepsize$ and $ranges$ parameters.

The $picky$ parameter determines how this function handles behaves if it encounters parameters in the $range$ dictionary, which were either not thawed or have been thawed but have not been specified in $ranges$. If $picky$ is True (the default), this function throws an error if $ranges$ does not cover all and only the free parameters. If picky is False, the function will automatically thaw all parameters specified through $ranges$ and freeze the rest.

**Warning:** There is NO guarantee that the sampling parameters (start values, limits for the uniform priors, and initial step sizes for the sampler) are reasonable. You need to check the results.

**Parameters**

- **ranges** [dictionary] Holds the fit ranges for the individual parameters. If single values are given, the sampling range (uniform prior) will be arranged symmetrically around the parameter's current value. It is also possible to specify a range directly using, e.g., “A1”:[0,100].
- **stepsize** [float, optional] Defines the step size as a fraction of the fit range given in $ranges$.
- **picky** [boolean, optional] If True (default), the list of free parameters has to match exactly the list of parameters specified in $ranges$. If False, the list of free parameters will be adapted to those given in $ranges$.
- **setRestrictionsFromPriors** [boolean, optional] Default: False. If True, parameter restrictions are applied according to the ranges of the uniform priors.

**Returns**

- **$X_0$** [dictionary] Maps parameter name to start value.
- **lims** [dictionary] Maps parameter name to [lower, upper] limit.
- **steps** [dictionary] Maps parameter name to step size.

**Examples**

```python
from PyAstronomy import funcFit as fuf
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)
```python
x = np.linspace(0, 30, 1000)
gauss = fuf.GaussFit1d()
gauss["A"] = 1
gauss["mu"] = 23.
gauss["sig"] = 0.5
yerr = np.random.normal(0., 0.05, len(x))
y = gauss.evaluate(x) + yerr
# This step is not necessary if <picky>=False in MCMCautoParameters.
gauss.thaw(['A', 'mu', 'sig'])
X0, lims, steps = gauss.MCMCautoParameters({"A":[0, 10], "mu":3, "sig":[0.1, 1. -0]})
gauss.fitMCMC(x, y, X0, lims, steps, yerr=yerr, iter=1000)
plt.plot(x, y, 'k+')
plt.plot(x, gauss.evaluate(x), 'r--')
plt.show()
```

**addConditionalRestriction** (*args*)

Define a conditional restriction.

Conditional restrictions can be used to modify the behavior in a more complex manner. For instance, penalties can be added to the objective function depending on the relation of one or more parameters.

The given function is evaluated in each iteration and its return value (a float) is added to the objective function (e.g., chi square).

**Parameters**

- **pars** [list of strings] The names of the parameters the given function needs as input.
- **func** [callable object] This callable object must take the specified parameters (in that exact order) as input. It must return a float, which is added to the value of the objective function.

**Returns**

- **identifier** [string] A unique ID used to refer to the conditional restriction.

**assignValue** (**specval**)  
Assign new values to variables.

**Parameters**

- **specval** [dictionary] Dictionary mapping variable names (wither given as string or specifier tuple) to value.

**assignValues** (**specval**)  
Assign new values to variables.

**Parameters**

- **specval** [dictionary] Dictionary mapping variable names (wither given as string or specifier tuple) to value.

**autoFitMCMC** (**x**, **y**, **ranges**, **picky**=**True**, **stepsize**=0.001, **yerr**=**None**, **pymcPars**={}, **pyy**=**None**, **potentials**=[], **dbfile**='mcmcSample.tmp', **dbArgs**={}, **sampleArgs**)

Convenience function to using auto-generated sampling parameters in MCMC.

This function is essentially a wrapper around **fitMCMC**(). It allows you to use the start values, step sizes, and limits constructed using `MCMCautoParameters()`. This method takes all parameters but `X0`, `Lims`, and `Steps`, which are expected by `fitMCMC()`. Additionally, the following parameters are available.
Parameters

ranges [dictionary] Holds the fit ranges for the individual parameters. If a single values is given, the fit range will be arranged symmetrically around the parameter’s current value. It is also possible to specify a range directly using, e.g., “A1”: [0, 100].

stepsizes [float, optional] Defines the step size as a fraction of the fit range given in ranges.

picky [boolean, optional] If True (default), the list of free parameters has to match exactly the list of parameters specified in ranges. If False, the list of free parameters will be adapted to those given in ranges.

Examples

```python
from PyAstronomy import funcFit as fuf
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 30, 1000)
gauss = fuf.GaussFit1d()
gauss["A"] = 1
gauss["mu"] = 23.
gauss["sig"] = 0.5
yerr = np.random.normal(0., 0.05, len(x))
y = gauss.evaluate(x) + yerr
ranges = {"A": [0, 5], "mu": 10, "sig": [0.1, 1.0]}
gauss.autoFitMCMC(x, y, ranges, yerr=yerr, iter=1000, picky=False)

plt.plot(x, y, 'k+')
plt.plot(x, gauss.evaluate(x), 'r--')
plt.show()
```

availableParameters()

Provides a list of existing parameters.

Returns

Parameters [list of strings] A list with the names of available parameters.

delRestriction (parName)

Delete restriction

Parameters

parName [string] Name of restricted parameter

description (parenthesis=False)

Returns a description of the model based on the names of the individual components.

Parameters

parenthesis [boolean, optional] If True, the entire expression/description will be enclosed in parenthesis. The default is False.

Returns
Description  [string] Description of the model.

errorConfInterval( par, dstat=2.706, statTol=0.01, hardLimit=None, maxiter=100, scale=None )
Calculate confidence interval for a parameter.

This function uses linear extrapolation (similar to Newton’s method) to find the points, where the objective function reaches an offset of \( dstat \) from the minimum value.

Note: You need to have carried out a fit before you can use errorConfInterval.

Parameters

par  [string] The parameter name for which to determine the error.

dstat  [float, optional] The offset from the minimum to be reached in the objective function’s value. For chi-square, a value of 1.0 corresponds to a one sigma (68%) confidence interval whereas the default of 2.706 refers to a 90% confidence interval.

statTol  [float, optional] The acceptable (absolute) tolerance in achieving the target value in the objective function. The default is 0.01.

hardLimit  [list of two floats, optional] Hard limits for the confidence interval. You can specify hard limits independently of parameter restrictions defined otherwise. To specify limits, specify a list of two floats defining the lower and upper limit. Use None where no limit is needed: e.g., use hardLimits=[0.0, None].

maxiter  [int, optional] The maximum number of acceptable iterations. If exceeded, the calculation of the confidence interval is stopped and None is returned. The default is 100.

scale  [float, optional] This number should reflect the typical scale or extent of the problem. If nothing is specified, a tenth of the current parameter value is used.

Returns

Confidence interval  [dictionary]

The dictionary contains the following keys:

- “limits” [A list of two floats specifying the lower and upper] limits of the confidence interval.
- “OFVals” [A list of two floats containing the values of the] objective function at the lower and upper limit.
- “OFMin” : The minimum value of the objective function.
- “parMinVal” [The parameter value at the minimum value of the] objective function (best-fit value).
- “iters” [The number of iterations needed to reach the result.] Note that the function returns None if the maximum number of iterations is exceeded.

fit(x, y, yerr=None, X0=None, minAlgo=None, mAA=None, miniFunc=None, printTime=False, *fminPars, **fminArgs)
Carries out a fit.

Uses an internal optimizer to find the best-fit parameters. By default, the method uses the scipy.optimize.fmin algorithm (Nelder-Mead Simplex) to find the best-fit parameters. After the fit, the parameter values are set to the best-fit values.

Parameters
x, y [arrays] Specify the abscissa and ordinate values of the data points.

yerr [array, optional] Error of data values.

X0 [list, optional] The initial guess. If not provided, it will be assumed that the current parameter values already contains the initial guess.

minAlgo [callable or string, optional] The minimization algorithm. If not specified, scipy’s ‘fmin’ implementation will be used. If a callable is given, it must adhere to funcFit’s internal optimizer model. Valid strings are:

- ‘spfmin’ : scipy.optimize.fmin
- ‘fufnm’ : funcFit’s implementation of the Nelder-Mead simplex algorithm.

mAA [dictionary, optional] Keyword arguments handed to the constructor of minAlgo, i.e., minAlgo Arguments (mAA). Valid keywords depend on the choice of the fit algorithm.

fminArgs [dict] Keywords passed to the minimization method (e.g., xtol or ftol for scipy.optimize.fmin).

fminPars : Non-keyword arguments passed to the minimization method (e.g., fprime in scipy.optimize.fmin_ncg).

miniFunc [None, string {“chisqr”, “cash79”}, or function] Function to be minimized. If None or “chisqr” is given, chi-square will be minimized. If “cash79” is given, the Cash statistics (Cash 1979, ApJ 228, 939, Eq. 5) will be minimized. If a function is specified, that, potentially user defined, function will be used to calculated the statistics, which will be minimized.

printTime: boolean, optional If True, the number of seconds needed to carry out the fit is printed. Default is False. At any rate, the time in seconds is stored in the “requiredTime” attribute.

fitEMCEE (x=None, y=None, yerr=None, nwalker=None, priors=None, pots=None, scales=None, sampleArgs=None, dbfile='chain.emcee', ps=None, emcp=None, toMD=True) MCMC sampling using emcee package.

Sample from the posterior probability distribution using the emcee package. By default the likelihood is calculated as -0.5 times the model chi-square value.

The emcee sampler can be accessed via the emceeSampler attribute, which may be used to continue or manipulate sampling.

Parameters

nwalker [int, optional] The number of walker to be used. By default, two times the number of free parameters is used.

scales [dictionary, optional] The scales argument can be used to control the initial distribution of the walkers. By default, all walkers are distributed around the location given by the current state of the object, i.e., the current parameter values. In each direction, the walker are randomly distributed with a Gaussian distribution, whose default standard deviation is one. The scales argument can be used to control the width of Gaussians used to distribute the walkers.

sampleArgs [dictionary, optional] Number controlling the sampling process. Use ‘burn’ (int) to specify the number of burn-in iterations (default is 0). Via ‘iters’ (int) the numbers of iterations after the burn-in can be specified (default 1000). The ‘process’ (int) key can be used to control the number of iterations after which the progress bar is updated (default is iters/100). Note that the ‘progressbar’ package
must be installed to get a progress bar. Otherwise more mundane print statements will be used.

**priors** [dictionary, optional] For each parameter, a primary can be specified. In particular, a prior is a callable, which is called with two arguments: first, a dictionary mapping the names of the free parameters to their current values, and second, a string specifying the name of the parameter for which the prior is to apply. The return value must be the logarithmic prior probability (natural logarithm). A number of default priors are available in the form of the `FuFPrior` class. By default, a uniform (improper) prior is used for all parameter, for which no other prior was specified.

**pots** [list, optional] A list of ‘potentials’. A potential is a function, which is called using a dictionary holding the current value for all parameters and returns the logarithm of the associated probability. Potentials may, e.g., be used to implement certain relations between parameter values not otherwise accounted for.

**dbfile** [string, optional] The result of the sampling, i.e., the chain(s), the corresponding values of the posterior, and the names of the free parameters are saved to the specified file (by default ‘chain.emcee’ is used). The traces stored there can be analyzed using the ‘TraceAnalysis’ class. Set this parameter to ‘None’ to avoid saving the results.

**ps** [tuple, optional] A tuple holding the current position and state of the sampler. This tuple is returned by this method. The `ps` argument can be used to continue sampling from the last state. Note that no burn-in will ne carried out and the other arguments should be given as previously to continue sampling successfully.

**emcp** [dictionary, optional] Extra arguments handed to `EnsembleSampler` object.

**toMD** [boolean, optional] If True (default), the object is set to the lowest-deviance solution after sampling. Otherwise, it remains in a random state.

```python
fitMCMC(x, y, X0, Lims, Steps, yerr=None, pymcPars=None, pyy=None, potentials=None, dbfile='mcmcSample.tmp', quiet=False, dbArgs=None, adaptiveMetropolis=False, **sampleArgs)
```

Carry out MCMC fit/error estimation.

This member is designed to provide a flexible but easy to use interface to the capabilities of pymc. In the simplest case, it assumes a Poisson or Gaussian distribution of data points and uses continuous, uniform variables (all free fitting variables) with starting values defined by `X0`, Limits given by `Lims`, and step sizes given by `Steps` to sample from the posterior.

**Note:** The result (the Marchov-Chain/PyMC MCMC-object) will be contained in the self.MCMC property; the output dictionary of MCMC.stats() (Mean, HPD interval etc.) is saved to self.basicStats.

**Parameters**

- **x, y** [array] Array providing the abscissa and ordinate values of the data points. For the MCMC process, it is essential to know the underlying distribution of the data points. `fitMCMC` assumes Poisson distributed data if `yerr` is not specified and Gaussian data if it is specified. If other distributions shall be used, the `pyy` parameter must contain a PyMC random variable specifying it.

- **yerr** [array, optional] The error of the data points. If not specified, Poisson distributed data will be assumed.
pyy [PyMC stochastic, optional] A PyMC variable containing the data. This can be useful if the distribution is neither Poisson nor Gaussian as otherwise assumed by this method.

X0 [dict] A dictionary holding \{"parName":value, \ldots\} specifying the start values. Note that parameters treated in pymcPars must not be part of this.

Lims [dict] A dictionary of type \{"ParName:[min,max], \ldots\} specifying the lower and upper limit of a variable. Note that parameters treated in pymcPars must not be part of this.

Steps [dict] A dictionary providing the step sizes for the MCMC sampler.

pymcPars [dict, optional] Holds a dictionary of the form \{"parName":PyMC-Variable, \ldots\}. pymcPars can be used to specify a nonuniform distribution for a parameter.

potentials [list of PyMC potentials, optional] Can be used to provide a list of PyMC potentials, which may be needed to provide priors.

dbfile [string, optional] The name of the output file, which is to hold the MCMC chain produced during sampling (default=“mcmcSample.tmp”).

quiet [bool, optional] Set to True in order to suppress the text output.

sampleArgs [dict, optional] Here additional keywords can be specified, which will be handed to the isample member of PyMC. Most notably it is useful to specify iter, burn, and thin. For other possibilities see PyMC documentation.

dbArgs [dict, optional] Keyword arguments specifying the trace database to be used. By default a pickle database named according to the dbfile keyword. You may also use an hdf5 database.

adaptiveMetropolis [bool, optional] Set to true in order to use the AdaptiveMetropolis step method.

freeParamNames ()
Get the names of the free parameters.

Returns

Free parameters [list of strings] The names of the free parameters. The order is determined by the paramNum attribute.

freeParameters ()
Get names and values of free parameters.

Returns

Free parameters: dict Dictionary containing the names and values of all free parameters (\{"parName":value, \ldots\}).

freeze (specifiers)
Consider variables free to float.

Parameters

specifiers [list of strings or tuples] The names of the variables to be thawed. Either given as string or specifier tuple.

frozenParameters ()
Get names and values of frozen parameters.

Returns
Frozen parameters: `dict` Dictionary containing the names and values of all frozen parameters (``parName``:value, … ).

``getRelationsOf(specifier)``
Return relations of a variable.

Parameters

   specifier [string or tuple] Variable name or specifier tuple.

Returns

   Relations [list of relations] Those relations in which specifier is the dependent variable, i.e., specifier = f(x,y).

``getRestrictions()``
Get all restrictions.

Returns

   Restrictions [dict] Dictionary associating parameter name and restriction (see restrictions attribute).

``hasVariable(specifier)``
Determine whether the variable exists.

Parameters

   specifier [string or tuple] Defines the name of the variable to be checked by string or specifier.

Returns

   Flag [boolean] True if the variable exists.

``numberOfFreeParams()``
Get number of free parameters.

Returns

   n [int] The number of free parameters (determined by isFree).

``parameterSummary(toScreen=True, prefix='', sorting='none')``
Writes a summary of the parameters in text form.

Parameters

   toScreen [bool, optional, default = True] If True, the output is written to the screen.

   prefix [string, optional, default = ‘’] A prefix for each line (e.g., ‘#’).

   sorting [string, optional, {'none', 'ps'}] Determines the order in which the parameters are printed out. If ‘none’ is given (default), no particular order is imposed. If ‘ps’, Python’s sorting routine is used to impose an order.

Returns

   A list of strings containing the text.

``parameters()``
Obtain parameter names and values.

Returns

   Name-value [dict] A dictionary with the names and values of all parameters (``parName``:value, … ).
\texttt{relate} (\textit{dependentVar}, \textit{independentVars}, \textit{func}=\texttt{None}, **\texttt{kwargs})

Define a relation.

**Parameters**

- \textit{dependentVar} [string or tuple] The dependent variable given by string or specifier tuple.
- \textit{independentVars} [string or list of strings] The independent variables. You may also use specifier tuples to address the variables.
- \textit{func} [callable] The functional form of the relation. Must take the independent variables as arguments and returns the value of the dependent variable. If None is given, equality will be assumed.

\texttt{removeConditionalRestriction} (*\texttt{args})

Remove an existing conditional constraint.

**Parameters**

- \textit{id} [string] The identifier used to refer to the conditional constraint (returned by \texttt{addConditionalRestriction}).

\texttt{renameVariable} (\textit{oldName}, \textit{newName})

Change name of variable.

**Parameters**

- \textit{oldName} [string] Current variable name.
- \textit{newName} [string] New variable name.

**Notes**

Variable names and properties are not the same.

\texttt{restoreState} (\texttt{resource})

Restores parameter values from file or dictionary.

**Parameters**

- \textit{resource} [string or dictionary] If string, it is interpreted as filename of a pickle file holding the data dictionary. If dictionary, it uses the data saved in it; note that a valid data dictionary is returned by \texttt{saveState}.

\texttt{saveState} (*\texttt{args}, **\texttt{kwargs})

Save the state of the fitting object.

This method collects the parameter values, the applied restrictions, and the information whether parameters are free or frozen and saves them to the specified file (if given) using pickle. The saved state can be restored using the \texttt{restoreState} method.

**Note:** Unfortunately, “relations” cannot be saved.
Returns

**Saved data**  [dict] The dictionary containing the data saved to the file.

**setObjectiveFunction**(*miniFunc=’chisqr’)  
Define the objective function.

This function sets the *miniFunc* attribute, which is used to calculate the quantity to be minimized.

**Parameters**

*miniFunc*  [str {chisqr, cash79, sqrdiff} or callable] The objective function. If “chisqr”,
chi-square will be minimized. If “cash 79”, the Cash statistics (Cash 1979, ApJ
228, 939, Eq. 5) will be used. If “sqrdiff” is specified, Otherwise, a user-defined
function is assumed.

**setPenaltyFactor**(*penalFac*)  
Change the penalty factor.

**Parameters**

*penalFac*  [float] The penalty factor (default is 1e20).

**Notes**

May also be done by accessing the *penalty* property directly.

**setRestriction**(*restricts*)  
Define restrictions.

**Parameters**

*restricts*  [dictionary] A dictionary associating a variable (given as string or specifier
tuple) with a restriction of the form: [lower, upper]. Use “None” where no restric-
tions shall be applied.

**setRootName**(*root*, *rename=False*)  
Define the root name of the model.

**Parameters**

*root*  [string] A concise description of the model.

*rename*  [bool, optional, default=False] If true, all model variables will be renaming
using the root.

**showConditionalRestrictions**(**kwargs**)  
Show conditional restrictions.

**Parameters**

*toScreen*  [boolean, optional] If True (default), the output is written to stdout.

**Returns**

*Output*  [list of strings] The output as a list of strings.

**steppar**(*pars*, *ranges*, *extractFctVal=None*, *quiet=False*)  
Allows to step a parameter through a specified range.

This function steps the specified parameters through the given ranges. During each steps, all free param-
eters, except for those which are stepped, are fitted. The resulting contours allow to estimate confidence
intervals.
This command uses the fitting parameters specified on a call to the `fit` method. In particular, the same values for `x`, `y`, `yerr`, `minAlgo`, `miniFunc`, `fminPars`, and `fminArgs` are used.

**Note:** You need to have carried out a fit before you can use `steppar`.

**Parameters**

- **pars** [string or list of strings] The parameter(s) which are to be stepped.
- **ranges** [dictionary] A dictionary mapping parameter name to range specifier. The latter is a list containing [lower limit, upper limit, no. of steps, ‘lin’/’log’]. The fourth entry, which is optional, is a string specifying whether a constant linear step size (‘lin’) or a constant logarithmic step size (‘log’) shall be used.
- **quiet** [boolean, optional] If True, output will be suppressed.
- **extractFctVal** [callable, optional] A function specifying how the function value is extracted from the fit result. If standard settings are used, the default of None is adequate.

**Returns**

- **Parameter steps** [list] The return value is a list of lists. Each individual list contains the values of the stepped parameters as the first entries (same order as the input `pars` list), the following entry is the value of the objective function (e.g., chi square), and the last entry is a tuple containing the indices of the steps of the parameter values. This last entry can be useful to convert the result into an arrow to plot, e.g., contours.

**thaw**(specifiers)
Consider variables fixed.

**Parameters**

- **specifiers** [list of strings or tuples] The names of the variables to be fixed. Either given as string or specifier tuple.

**untie**(parName, forceFree=False)
Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters. The parameter parName is set to “freeze”.

**Parameters**

- **parName** [string] The name of the dependent variable which should become “unrelated”.
- **forceFree** [boolean] Set parName to “free” instead of “frozen” if set to True.

**updateModel**()
Recalculate the model using current settings.

**Notes**

Updates the `model` attribute of the class by calling `evaluate` using the `x` attribute, which is, e.g., assigned on call to a fit method.
Variable naming

class PyAstronomy.funcFit.ModelNameIdentBase(rootName="")

Managing the naming of model components.

This class handles the names of models or model components. Individual names or identifiers are composed of a “root name” and a “component counter”. The root name is supposed to be a concise string summarizing the type of model, while the component counter is used to distinguish between components with the same root name in composed models.

Parameters

  rootName [string, optional] A concise name for the model (default="").

Notes

The term “specifier” is used to indicate either a string containing a variable name or a tuple composed of (property, root name, component counter), which specifies a variable. In some cases, parts of the specifier tuple may be left out.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>composeVariableName</td>
<td>Combine property, root name, and counter into a</td>
<td>property[, rootName, counter]</td>
</tr>
<tr>
<td>convertSpecifier</td>
<td>Decompose specifier and compose variable name.</td>
<td>specifier</td>
</tr>
<tr>
<td>decomposeVariableName</td>
<td>Decomposes variable name into constituents.</td>
<td>name</td>
</tr>
<tr>
<td>getRoot()</td>
<td>Returns the root name.</td>
<td></td>
</tr>
<tr>
<td>identifier()</td>
<td>Return the appropriate identifier.</td>
<td></td>
</tr>
<tr>
<td>setRootName(root)</td>
<td>Define the root used to build the identifier.</td>
<td>root</td>
</tr>
<tr>
<td>specifierToName(input)</td>
<td>Convert specifier(s) to variable names.</td>
<td>input</td>
</tr>
</tbody>
</table>

composeVariableName (property, rootName=None, counter=None)

Combine property, root name, and counter into a variable name.

Parameters

  property [string] The property name.
  rootName [string, optional] A root name. If None, the instance’s own root name will be used.
  counter [int, optional] The component counter. If None, the instance’s counter will be used.

Returns

  Variable name [string] The resulting variable name.

Notes

The standard way used to compose variable names is: “property_rootname(counter)”
**convertSpecifier** (*specifier*)
Decompose specifier and compose variable name.

**Parameters**

**specifier** [string or tuple,] Either a string giving a variable name or a tuple specifying property, root name, and counter (the later two are optional).

**Returns**

**Decomposed specifier** [tuple] A tuple of (variable name, property, root name, component counter).

**decomposeVariableName** (*name*)
Decomposes variable name into constituents.

Expects a name of the from “property_root(c)” and returns the individual building blocks of the name, i.e., property, root name, component counter in that order. If one or more of these building blocks is not present, None will be returned instead.

**Parameters**

**name** [string] The variable name.

**getRoot** ()
Returns the root name.

**identifier** ()
Return the appropriate identifier.

**Returns**

**identifier** [string]

**Notes**

The “identifier” reads “root-name(c)”, where the later part, “(c)”, will only be present if the *component-Counter* property is larger than zero; in this case, ‘c’ is that number.

**setRootName** (*root*)
Define the root used to build the identifier.

**Parameters**

**root** [string] Concise description of the model.

**specifierToName** (*input*)
Convert specifier(s) to variable names.

**Returns**

**out** [Same type as input,] Converts all specifiers to variable names. The output is the same type as the input.

---

**Python decorators and MiniFunc**

In the following, we give a brief introduction to Python decorators and outline the use of the MiniFunc decorator.

**Decorators in Python** Loosely speaking, decorators can be used to add or modify the functionality of a callable object such as a function. This can for example be useful if a whole set of function shares common functionality or if a certain fraction of the required functionality is to be “hidden” from the user, as is the case here.
Applying a decorator, which is in Python marked by the `@` sign, turns one callable object into another one. In our case, it will be necessary to have a decorator with arguments. A minimalistic example may look like this:

```python
class Deco:
    def __init__(self, argument):
        self.argument = argument
    def __call__(self, f):
        print "Deco.__call__ is speaking!"
        def newFunc(y):
            print "First argument was: ", self.argument
            print "But now I received: ", y
            print "Evaluating function at given point: "
            f(y)
        return newFunc

def f1(x):
    print "x**2 = ", x**2
@Deco(13)
def f2(x):
    print "x**2 = ", x**2
print "------------------------------"
f1(2)
print "------------------------------"
f2(2)
```

Execution the above script results in the following output:

```
Deco.__call__ is speaking!
------------------------------
x**2 = 4
------------------------------
First argument was: 13
But now I received: 2
Evaluating function at given point:
x**2 = 4
```

What happens? At first sight, the functions `f1` and `f2` look exactly alike, yet the definition of `f2` is preceded by the `@Deco(13)` directive; obviously, a call to `f1` provides other output than a call of `f2`.

Above we defined a class called `Deco`. Instances of that class are callable objects, as they have a `__call__` method. The `@Deco(13)` directive preceding `f2` triggers the following process:

1) The constructor of Deco is called with a single argument (13).

2) The `__call__` method of Deco is called with a single argument, which is a function pointer pointing at `f2`.

3) The `__call__` method defines a new function, incorporating the functionality provided by `f2`, and returns it. This new function can combine the information given to the constructor and the arguments handed to the new function.

The whole process is, thus, replacing one callable object by another, possibly combining and extending the functionality.
The MiniFunc decorator can be used to apply user-defined minimization functions. In principle, such a function determines the quantity to be minimized (e.g., Chi square) for a given model and parameter vector. The implementation of the module requires to also carry out some other steps, which is done through the decorator, without making this visible for the user.

An example of a user defined objective function looks like:

```python
gf = GaussFit1d()
@MiniObj(gf)
def mini(odf, P):
    return sum(abs((odf.y - odf.model)/odf.yerr))
```

The first argument of the objective function is the fitting object, and the second is the parameter vector. Properties of the fitting objects such as `x`, `y`, or `model` may be accessed as usual. The return value is the quantity to be minimized.

**Note:** An example of the application of the decorator is given in the tutorial.

```python
class PyAstronomy.funcFit.MiniFunc(odf)
    This decorator can be applied to use self-defined objective functions.
    Applied to an objective function, it adds the functionality needed to evaluate the model given a certain parameter vector, so that the user does only have to take care about the quantity to be minimized.

    Parameters
    - **odf** [fitting object] The fitting object that is supposed to use the self-defined objective function.

    Methods
    ```
    __call__(f) Parameter:
    ```
    ```python
    def __call__(f):
        return f
    ```
    ```
    Parameter:
    ```
    ```python
    - **odf** - An instance of a fitting objects such as for example `GaussFit1d`.
    ```
    ```
    Methods
    ```
    __call__(f) Parameter:
    ```
    ```python
    API of sampleEMCEE
    ```
    PyAstronomy.funcFit.sampleEMCEE(fpns, fv0, lnprob=None, nwalkers=1, nsteps=1, scales=0.2, sampleArgs=None, dbfilename='chain.emcee', ps=None, emcp=None)
    ```
    MCMC sampling from specific density using the emcee package.
    This function may be used to use emcee to sample from any user-specified density, which does not have to be normalized. The resulting Markov Chains can be analyzed using the trace analysis package.

    Parameters
    ```
    - **fpns** [list of strings] Names of parameters for which Markov Chains are constructed.
    ```
    ```
    - **fv0** [dictionary] A dictionary mapping parameter name to starting value. This dictionary
    ```
    ```
    2.5. Model fitting
    ```
may contain any number of additional key-value pairs

**lnp** [callable] A function (or equivalent callable) which returns the (natural) logarithm of the (generally unnormalized) density. The first (and only mandatory) argument to the function is a dictionary holding the current parameter values for which the posterior density is to be evaluated. The function may take any number of additional keyword arguments, which can be specified by the `largs` parameter.

**largs** [dictionary, optional] A set of additional arguments passed to the `lnp` callable.

**nwalker** [int, optional] The number of walker to be used. By default, four times the number of free parameters is used.

**scales** [dictionary, optional] The scales argument can be used to control the initial distribution of the walkers. By default, all walkers are distributed around the location given by the current state of the object, i.e., the current parameter values. In each direction, the walkers are randomly distributed with a Gaussian distribution, whose default standard deviation is one. The scales argument can be used to control the width of the Gaussians used to distribute the walkers.

**sampleArgs** [dictionary, optional] Number controlling the sampling process. Use ‘burn’ (int) to specify the number of burn-in iterations (default is 0). Via ‘iters’ (int) the number of iterations after the burn-in can be specified (default 1000). The ‘process’ (int) key can be used to control the number of iterations after which the progress bar is updated (default is iters/100). Note that the ‘progressbar’ package must be installed to get a progress bar. Otherwise more mundane print statements will be used.

**dbfile** [string, optional] The result of the sampling, i.e., the chain(s), the corresponding values of the posterior, and the names of the free parameters are saved to the specified file (by default ‘chain.emcee’ is used). The traces stored there can be analyzed using the ‘TraceAnalysis’ class. Set this parameter to ‘None’ to avoid saving the results.

**ps** [tuple, optional] A tuple holding the current position and state of the sampler. This tuple is returned by this method. The `ps` argument can be used to continue sampling from the last state. Note that no burn-in will be carried out and the other arguments should be given as previously to continue sampling successfully.

**emcp** [dictionary, optional] Extra arguments handed to `EnsembleSampler` object.

Returns

**pos, state** [state of emcee sample] These information may be used to continue the sampling from previous position.

**Using an overbinned model in calculations**

Calculating the model at more points than actually needed can be very useful, e.g., if finite integration times are to be taken into account. The `turnIntoRebin` method turns every “normal” model into one, which uses rebinning. See the tutorial for an example of the application.

```python
PyAstronomy.funcFit.turnIntoRebin(CO)
```

Turn a “normal” fitting object into rebinning fitting object.

This function accepts a class object representing a model and returns another class object extended by the rebinning functionality.

**Parameters**

**CO** [A class object] The class object describing the model to use rebinning.

**Returns**
**Rebinned model**  [Fitting object] Another class object extended by the rebinning functionality.

The class object returned by the `turnIntoRebin` function is the received class object with a modified `evaluate` method and some extra functionality; technically, this is achieved by inheritance. The class object denoted by `_ModelRebinDocu` below inherits the functionality of the incoming class object and is returned.

```python
class PyAstronomy.funcFit._ModelRebinDocu(*args, **kwargs)
    Base class providing rebinning functionality.

    The model is evaluated at more points than actually needed. Several points are then averaged to obtain a “binned” model, which can, for example, account for finite integration times in observations.

    Attributes

    rebinTimes [array] Defined the abscissa values at which to evaluate the model to be rebinned afterwards.

    rebinIdent [dict] A dictionary associating bin number (in the unbinned model) with a list holding the bins in `rebinTimes`, which are to be averaged to obtain the binned model.

    Methods

    `evaluate(x)` Calculate the model.

    `setRebinArray_Ndt(time, N, dt)` Defines the overbinning parameters (`rebinTimes`, `rebinIdent`).

    `evaluate(x)` Calculate the model.

    Parameters

    x [array] The abscissa values.

    Returns

    model [array,] The binned model.

    Notes

    This function calculates the model at those time points specified by the `rebinTimes` property and saves the result in the class property `unbinnedModel`. Then it bins according to the definitions in `rebinIdent` and save the resulting model in the `binnedModel` property.

    `setRebinArray_Ndt(time, N, dt)` Defines the overbinning parameters (`rebinTimes`, `rebinIdent`).

    It is assumed that the time points given in the `time` array refer to the center of the time bins and every bin has length `dt`. The bins are then subdivided into `N` subintervals; the center of each such subinterval becomes a point in the overbinned time axis (`rebinTimes`).

    Parameters

    time [array] The time axis of the “observed” (not overbinned) transit light-curve.

    N [int] The number of point into which to subdivide each time bin of length `dt`.

    dt [float] The length of each time bin (on the original not oversampled time axis).
Simultaneous model fitting

The class `SyncFitContainer` is designed to provide a simple interface for fitting different models defined on different axes simultaneously. It is designed to mimic the behavior of the `OneDFit` class, but is not itself an object derived from `OneDFit`. An example of usage is given in the tutorial.

```python
class PyAstronomy.funcFit.SyncFitContainer
    Simultaneous model fitting.
    As an example, take a simultaneous measurement of a photometric planetary transit and the Rossiter-McLaughlin effect. Surely, both should be described by a subset of common parameters like the size of the planet and the large semi-major axis, but the models/measurements refer to quite different regimes: brightness and radial-velocity shift. This class can be used to carry out a fit of both simultaneously.

Attributes

- **pars** [Instance of Params] Manages the model parameters.
- **models** [dictionary] A dictionary of the form component-number model; saves the evaluated models.
- **penaltyFactor** [float] Factor used to scale the penalty imposed if parameter restrictions are violated.
- **_compos** [dictionary] A dictionary of the form component-number model-component. The component number uniquely identifies every model component.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addComponent(newCompo)</td>
<td>Add a new component to the container.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>componentes()</td>
<td>Returns:</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>evaluate(axes[, component])</td>
<td>Parameters:</td>
</tr>
<tr>
<td>fit(data[, yerr, X0, minAlgo, miniFunc])</td>
<td>Carries out the fit.</td>
</tr>
<tr>
<td>fitMCMC((data, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
</tbody>
</table>
```

Continued on next page
### Table 57 – continued from previous page

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>restoreState(resource)</code></td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td><code>saveState(*args, **kwargs)</code></td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td><code>setObjectiveFunction(miniFunc)</code></td>
<td>Define the objective function.</td>
</tr>
<tr>
<td><code>setRestriction(restricts)</code></td>
<td>Define restrictions.</td>
</tr>
<tr>
<td><code>showConditionalRestrictions(**kwargs)</code></td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><code>steppar(pars, ranges[, extractFctVal, quiet])</code></td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><code>thaw(specifiers)</code></td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><code>treatAsEqual(parameter)</code></td>
<td>Treat parameters as equal.</td>
</tr>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Evaluate all components.</td>
</tr>
</tbody>
</table>

#### `addComponent (newCompo)`

Add a new component to the container.

**Parameters:**

- `newCompo` - A funcFit model.

**Returns:** The component identifier.

#### `components ()`

**Returns:** A list holding the component names.

#### `evaluate (axes, component=None)`

**Parameters:**

- `axes` - If `component` is not given, a dictionary holding the x-axis for each component name. Otherwise, the x-axis for the specified component.
- `component` - string, optional, The name of the component to be evaluated.

The evaluated model(s) is saved in the `models` dictionary.

#### `fit (data, yerr=None, X0=None, minAlgo=None, miniFunc=None, *fminPars, **fminArgs)`

Carries out the fit.

In principle, any fit algorithm can be used. If none is specified, the default is `scipy.optimize.fmin` (Nelder-Mead Simplex). Another choice could for instance be `scipy.optimize.fmin_powell`. After the fit, the return value of the fitting method is stored in the class property `fitResult` and the `model` property is set to the best fit.

**Parameters:**

- `data` - Dictionary of the form: `{c:[x, y], . . .}`. Here `c` is the component number (starts with one, and `x, y` are the x-axis and associated values.
- `yerr` - array, optional: Error of data values. A dictionary of the from: `{c:yerr}`, where `c` is the component and `yerr` the array of error bars.
- `X0` - list, optional: The initial guess. If not provided, it will be assumed that self.pars already contains the initial guess.
- `minAlgo` - callable, The minimization algorithm. Default is `scipy.optimize.fmin`; other algorithms from scipy may be chosen. Alternatively, any callable object taking the function to
Carry out MCMC fit/error estimation. This member is designed to provide a flexible but easy to use interface to the capabilities of pymc. In the simplest case, it assumes a Poisson or Gaussian distribution of data points and uses continuous, uniform variables (all free fitting variables) with starting values defined by \( X_0 \), Limits given by \( Lims \), and step sizes given by \( Steps \) to sample from the posterior.

**Note:** The result (the Marchov-Chain/PyMC MCMC object) will be contained in the self.MCMC property; the output dictionary of MCMC.stats() (Mean, HPD interval etc.) is saved to self.basicStats.

**Parameters:**

- \( x \)- An array providing the x-values of the data points.
- \( y \)- An array providing the y-values of the data points. Note that for the MCMC process, it is essential to know the underlying distribution of the data points. \( \text{fitMCMC} \) assumes Poisson distributed data of \( yerr \) is not specified and Gaussian data if it is specified. If other distributions shall be used, the \( pyy \) parameter must contain a pymc random variable specifying it.
- \( yerr \)- array, optional, Error of data values. A dictionary of the from: \{c:yerr\}, where \( c \) is the component and yerr the array of error bars. If specified a Gaussian distribution will be assumed for the data points, otherwise a Poisson distribution is assumed.
- \( ppy \)- optional, Can be used to handle a PyMC variable containing the data. This can be useful if the distribution is neither Poisson nor Gaussian as otherwise assumed by this function.
- \( X_0 \)- A dictionary holding \{“parName”:value, \ldots\} specifying the start values. Note that parameters treated in pymcPars must not be part of this.
- \( Lims \)- A dictionary of type \{“ParName:[min,max], \ldots\} specifying the lower and upper limit of a variable. Note that parameters treated in pymcPars must not be part of this.
- \( Steps \)- A dictionary providing the step sizes for the MCMC sampler.
- \( pymcPars \)- optional, This variable is supposed to hold a dictionary of the form \{“parName”:PyMC-Variable, \ldots\}. \( pymcPars \) can be used to specify a nonuniform distribution for a parameter.
- \( potentials \)- optional, Can be used to provide a list of PyMC potentials, which may be needed to provide priors.
- \( dbfile \)- The name of the output file, which is to hold the MCMC chain produced during sampling.
- **sampleArgs \)- optional, Here additional keywords can be specified, which will be handed to the isample member of PyMC. Most notably it is useful to specify \( \text{iter, burn, and thin} \). For other possibilities see PyMC documentation.
setObjectiveFunction(miniFunc='chisqr')

Define the objective function.

This function sets the miniFunc attribute, which is used to calculate the quantity to be minimized.

**Parameters**

miniFunc [str {chisqr, cash79, sqrdiff} or callable] The objective function. If “chisqr”, chi-square will be minimized. If “cash 79”, the Cash statistics (Cash 1979, ApJ 228, 939, Eq. 5) will be used. If “sqrdiff” is specified, Otherwise, a user-defined function is assumed.

steppar(pars, ranges, extractFctVal=None, quiet=False)

Allows to step a parameter through a specified range.

This function steps the specified parameters through the given ranges. During each steps, all free parameters, except for those which are stepped, are fitted. The resulting contours allow to estimate confidence intervals.

This command uses the fitting parameters specified on a call to the fit method. In particular, the same values for x, y, yerr, minAlgo, miniFunc, fminPars, and fminArgs are used.

**Note:** You need to have carried out a fit before you can use steppar.

**Parameters**

pars [string or list of strings] The parameter(s) which are to be stepped.

ranges [dictionary] A dictionary mapping parameter name to range specifier. The latter is a list containing [lower limit, upper limit, no. of steps, ‘lin’/’log’]. The fourth entry, which is optional, is a string specifying whether a constant linear step size (‘lin’) or a constant logarithmic step size (‘log’) shall be used.

quiet [boolean, optional] If True, output will be suppressed.

extractFctVal [callable, optional] A function specifying how the function value is extracted from the fit result. If standard settings are used, the default of None is adequate.

**Returns**

Parameter steps [list] The return value is a list of lists. Each individual list contains the values of the stepped parameters as the first entries (same order as the input pars list), the following entry is the value of the objective function (e.g., chi square), and the last entry is a tuple containing the indices of the steps of the parameter values. This last entry can be useful to convert the result into an arrow to plot, e.g., contours.

treatAsEqual(parameter)

Treat parameters as equal.

**parameter - string or list of string**, If a string is given, all parameters with this “base name” (i.e., neglecting everything after an underscore) will be treated as equal. Otherwise the specified parameters will be treated as equal.

This method uses the relations known from OneDFit to treat parameters as equal. Dependent variables are thawed before the relation is applied, if they are not already free.

updateModel()

Evaluate all components. Updates the values in the models dictionary.
Analysis of Markov-Chains

The class `TraceAnalysis` provides some convenient methods to carry out an analysis of the Markov-Chains resulting from an MCMC “fit”.

Examples for using the class are given in the associated tutorial: *Analyze Markov-Chains using TraceAnalysis*

**class** `PyAstronomy.funcFit.TraceAnalysis(resource, db='pickle')`

This class provides a number of plotting methods. Note that **you still need to call *show()* from pylab to see the result.**

**Parameters**

- **resource** [string or pymc database object] If string, it assumed to be the filename of the Markov Chain file. Otherwise, it is supposed to be a pymc database object. If the filename is of the form “*.emcee”, it is assumed to be a trace produced by emcee.

**Attributes**

- **burn** [int] Applies a burn-in. All iterations earlier than burn will be neglected.
- **thin** [int] Applies thinning to each chain. Retains every k th sample, where k is an integer value.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>availableParameters()</code></td>
<td>Returns list of available parameter names.</td>
</tr>
<tr>
<td><code>availableTraces()</code></td>
<td>Returns a list of available PyMC Trace objects.</td>
</tr>
<tr>
<td><code>correlationMatrix([toScreen, method, ...])</code></td>
<td>Calculates the correlation or covariance matrix.</td>
</tr>
<tr>
<td><code>correlationTable([parsList, coeff, noPrint])</code></td>
<td>Calculate and show parameter correlations.</td>
</tr>
<tr>
<td><code>hpd(parm[, trace, cred])</code></td>
<td>Calculates highest probability density interval (HPD, minimum width BCI).</td>
</tr>
<tr>
<td><code>mean(parm)</code></td>
<td>Calculate mean.</td>
</tr>
<tr>
<td><code>median(parm)</code></td>
<td>Calculate median.</td>
</tr>
<tr>
<td><code>numberOfWalkers()</code></td>
<td>Get number of walkers in emcee chain.</td>
</tr>
<tr>
<td><code>parameterSet([prescription])</code></td>
<td>Finds parameter values for a particular prescription.</td>
</tr>
<tr>
<td><code>pearsonr(parm1, parm2)</code></td>
<td>Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.</td>
</tr>
<tr>
<td><code>plotCorr([parsList])</code></td>
<td>Produces correlation plots.</td>
</tr>
<tr>
<td><code>plotCorrEnh([parsList])</code></td>
<td>Produces enhanced correlation plots.</td>
</tr>
<tr>
<td><code>plotDeviance([parsList])</code></td>
<td>Plots value of deviance over parameter values encountered during sampling.</td>
</tr>
<tr>
<td><code>plotHist([parsList])</code></td>
<td>Plots distributions for a number of traces.</td>
</tr>
<tr>
<td><code>plotTrace(parm[, fmt])</code></td>
<td>Plots the trace.</td>
</tr>
<tr>
<td><code>plotTraceHist(parm)</code></td>
<td>Plots trace and histogram (distribution).</td>
</tr>
<tr>
<td><code>quantiles(parm[, qlist])</code></td>
<td>Quantiles for given trace.</td>
</tr>
<tr>
<td><code>selectWalkers(ws)</code></td>
<td>Select walkers for emcee chains.</td>
</tr>
<tr>
<td><code>selectedWalkers()</code></td>
<td>Get the list of selected walkers.</td>
</tr>
<tr>
<td><code>setBurn(burn)</code></td>
<td>Change value of “post burn-in”.</td>
</tr>
<tr>
<td><code>setThin(thin)</code></td>
<td>Change value of “post thinning”.</td>
</tr>
<tr>
<td><code>setToState(model[, state, verbose])</code></td>
<td>Set the parameter values to a certain state.</td>
</tr>
<tr>
<td><code>show()</code></td>
<td>Call <em>show()</em> from matplotlib to bring graphs to screen.</td>
</tr>
</tbody>
</table>

*Continued on next page*
Table 58 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spearmanr(parm1, parm2)</code></td>
<td>Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.</td>
</tr>
<tr>
<td><code>state()</code></td>
<td>Returns dictionary containing basic information on the sampling process.</td>
</tr>
<tr>
<td><code>std(parm)</code></td>
<td>Calculate standard deviation.</td>
</tr>
</tbody>
</table>

```python
plotHists()
```

**availableParameters()**
Returns list of available parameter names.

**availableTraces()**
Returns a list of available PyMC Trace objects.

**correlationMatrix**(toScreen=True, method='pearson', parList=None, covariance=False)
Calculates the correlation or covariance matrix.

### Parameters
- **parList** [list of strings, optional] The list of parameters used in the calculation. If not given, all available parameters will be used.
- **toScreen** [boolean, optional] If True, the result will be printed to stdout
- **method** [string, {'pearson', 'spearman'}] The correlation coefficient to be used.
- **covariance** [boolean, optional] If True, the covariance will be returned instead of the correlation. The default is False.

### Returns
- **Parlist** [list] Parameter names in the order used in the calculation.
- **Correlation matrix** [2d array] The correlation matrix
- **lines** [list of strings] Formatted version of the correlation matrix in the form of a list of strings.

**correlationTable**(parsList=None, coeff='pearson', noPrint=False)
Calculate and show parameter correlations

### Parameters
- **parsList** [list of strings, optional] A list of parameters for which to calculate the correlation. If None, all available parameters will be used.
- **coeff** [string, {'pearson', 'spearman'}, optional] The coefficient to be used. By default, Pearson’s correlation coefficient will be used.
- **noPrint** [boolean, optional] If True, the table output will be suppressed.

### Returns
- **Coefficients** [dictionary] Maps each tuple of two parameter names to the associated correlation coefficient.

**hpd**(parm, trace=None, cred=0.95)
Calculates highest probability density interval (HPD, minimum width BCI).

### Parameters
- **parm** [string] Name of parameter
cred [float, optional] Credibility level. Defaults to 0.95, i.e., the 95% HPD will be calculated.

trace [array, optional] If a trace is given, it will be used in the calculation instead of the trace for parm stored in the class. Note that the parm will be ignored in this case!

Returns
HPD [tuple] The lower and upper bound of the credibility interval.

mean (parm)
Calculate mean.
Parameters
parm [string] Name of parameter.

Returns
The mean [float]

median (parm)
Calculate median.
Parameters
parm [string] Name of parameter.

Returns
The median [float]

numberOfWalkers ()
Get number of walkers in emcee chain.

Returns
nWalkers [int] Number of available walkers

parameterSet (prescription='lowestDev')
Find parameter values for a particular prescription.
Parameters
prescription [string, {"lowestDev", "mean", "median"}] Which parameter set to find.
If 'lowestDev' is used, the method will return the parameter set pertaining to the lowest deviance solution. If 'mean' or 'median' are specified, the mean or median parameter values determined from the Markov Chains are returned.

Returns
Parameter set [dictionary] A dictionary mapping all parameter names to the value derived using the specified prescription.

Lowest deviance index [int] The index of the lowest deviance solution. Only returned if the prescription is 'lowestDev'

pearsonr (parm1, parm2)
Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.
Parameters
parm1, parm2 [string] The names of the two parameters used in the evaluation.

Returns
Pearson correlation coefficient [float]
p-value [float]

Notes

Uses SciPy’s `scipy.stats.pearsonr` to evaluate.

The SciPy documentation of scipy.stats.pearsonr:

The Pearson correlation coefficient measures the linear relationship between two data sets. Strictly speaking, Pearson’s correlation requires that each data set be normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases. The p-value roughly indicates the probability of an uncorrelated system producing data sets that have a Pearson correlation at least as extreme as the one computed from these data sets. The p-values are not entirely reliable but are probably reasonable for data sets larger than 500 or so.

plotCorr (parsList=None, **plotArgs)

Produces correlation plots.

Parameters

parsList [list of string, optional] If not given, all available traces are used. Otherwise a list of at least two parameters has to be specified.

plotArgs [dict, optional] Keyword arguments handed to plot procedure of pylab.

plotCorrEnh (parsList=None, **plotArgs)

Produces enhanced correlation plots.

Parameters

parsList [list of string, optional] If not given, all available traces are used. Otherwise a list of at least two parameters has to be specified.

plotArgs [dict, optional] Keyword arguments handed to plot procedures of pylab. The following keywords are available: contour, bins, cmap, origin, interpolation, colors

plotDeviance (parsList=None)

Plots value of deviance over parameter values encountered during sampling.

Parameters

parsList [string or list of strings, optional] Refers to a parameter name or a list of parameter names. If None, all available parameters are plotted.

plotHist (parsList=None)

Plots distributions for a number of traces.

Parameters

parsList [string or list of strings, optional] Refers to a parameter name or a list of parameter names. If None, all available parameters are plotted.

plotTrace (parm, fmt='b-')

Plots the trace.

Parameters

parm [string] The variable name.
fmt [string, optional] The matplotlib format string used to plot the trace. Default is ‘b-’.

plotTraceHist (parm)
Plots trace and histogram (distribution).

Parameters
parm [string] The variable name.

quantiles (parm, qlist=None)
Quantiles for given trace.

Parameters
parm [string] Name of parameter
qlist [list of floats (0-100), optional] Specifies which quantiles shall be calculated. The default is 2.5, 25, 50, 75, and 97.5 percent.

Returns
Quantiles [dictionary] For each quantile (in percent) the corresponding value.

selectWalkers (ws)
Select walkers for emcee chains.

Parameters
ws [list or array of integers] The walker to be considered in the analysis. Counting starts at zero.

selectedWalkers ()
Get the list of selected walkers.

Parameters
walkers [array] The selected walkers.

setBurn (burn)
Change value of “post burn-in”.
In the case of an emcee trace, the “post burn-in” is applied to the trace of all walkers.

Parameters
burn [int] The number of samples to be neglected.

setThin (thin)
Change value of “post thinning”.

Parameters
thin [int] Applies thinning to each chain. Retains every $k$ th sample, where $k$ is an integer value.

Notes
Use the “post thinning” to thin out your chains.

setToState (model, state=’best’, verbose=True)
Set the parameter values to a certain state.

Parameters
model - fitting object The fitting model object whose parameters will be updated.
**state** [“best”, “mean”, optional]

“best” [Set parameters to the “best fit” state as measured by deviance. This is the default.]

“mean” : Set parameters to mean value of trace.

**verbose** [bool, optional] If False, no output about what is done will be generated (default is True).

**show()**

Call `show()` from matplotlib to bring graphs to screen.

**spearmanr** (parm1, parm2)

Calculates a Spearman rank-order correlation coefficient and the p-value to test for non-correlation.

**Parameters**

parm1, parm2  [string] The names of the two parameters used in the evaluation.

**Returns**

*Spearman rank-order correlation coefficient*  [float]

*p-value*  [float]

**Notes**

Uses SciPy’s `scipy.stats.spearmanr` to evaluate.

The SciPy documentation of `scipy.stats.spearmanr`:

The Spearman correlation is a nonparametric measure of the monotonicity of the relationship between two data sets. Unlike the Pearson correlation, the Spearman correlation does not assume that both data sets are normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact monotonic relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases. The p-value roughly indicates the probability of an uncorrelated system producing data sets that have a Spearman correlation at least as extreme as the one computed from these data sets. The p-values are not entirely reliable but are probably reasonable for data sets larger than 500 or so.

**state()**

Returns dictionary containing basic information on the sampling process.

**std**(parm)

Calculate standard deviation.

**Parameters**

parm  [string] Name of parameter.

**Returns**

*The standard deviation*  [float]

**PyAstronomy.funcFit. hpd**(trace, cred)

Estimate the highest probability density interval.

This function determines the shortest, continuous interval containing the specified fraction (cred) of steps of the Markov chain. Note that multi-modal distribution may require further scrutiny.

**Parameters**

trace  [array] The steps of the Markov chain.
cred  [float] The probability mass to be included in the interval (between 0 and 1).

Returns

start, end  [float] The start and end points of the interval.

PyAstronomy.funcFit.quantiles(trace, qs)
Get quantiles for trace.

Parameters

trace  [array] The steps of the Markov chain.
qs  [list or array] The quantiles in percent.

Returns

Quantiles  [dictionary] For each quantile, the corresponding value.

Constructing n-dimensional coordinate mapping

If you have n-dimensional input data such as an image, funcFit needs an appropriate coordinate mapping.

PyAstronomy.funcFit.coordinateGrid(*args)
Construct a coordinate array for n-dimensional fitting.

For n-dimensional fitting, funcFit requires a mapping from array index (i,j,k,...) to coordinate (x,y,z,...). This mapping must be given in the form of an array with dimension (n1, n2, ..., nd), where ni is the length of the i-th coordinate axis and nd is the number of coordinate axes. Then, e.g. in the 2d case, [0,0,0] gives the x-coordinate of index [0,0] and [0,0,1] given the associated y-coordinate.

Parameters

args  [arrays] An arbitrary number of coordinate arrays.

Returns

Coordinate grid  [array] An array of dimension (n1, n2, ..., nd) where n1, n2, etc. is the length of the n-th coordinate axis and nd is the total number of dimensions.

2.5.2 The PyA model suite

The PyAstronomy model suite is a set of implementations of physical models like, for example, a transit light-curve model. The models are supposed to provide an interface to the funcFit package so that they can be used for fitting.

Note: We provide examples for the individual classes. For a more detailed introduction to the capabilities of the funcFit package, we refer to funcFit’s tutorial.

Contents:

XTran—Transit modeling

PyA’s XTran packages bundles several transit models, which differ in purpose and implementation.
Transit light-curves

Analytical transit profiles (Pal 2008)

The `palTrans` module implements the analytical transit light-curve model presented by Pal 2008\(^1\).

Contents:

The model classes

The `PalLC` and `PalLCCirc` classes encapsulate all relevant calculations; the latter considers only circular orbits, while the former, `PalLC`, uses the `KeplerEllipse` class from PyA’s astrolib to consider a Kepler orbit.

They inherit from funFit’s OneDFit class.

Circular orbit

class PyAstronomy.modelSuite.XTran.palTrans.PalLC

Calculate and fit analytical transit light-curves using the formulae provided by Pal 2008.

This class uses a circular planetary orbit.

Note: The evaluation of elliptical integrals is essential in calculating the transit model. While both the `mpmath` module and the `Boost` libraries implement those integrals, it is the Boost library, which evaluates them by far more quickly. Yet, the support for Boost has to be added manually.

Model parameters:

- \(p\) - Radius ratio between planet and star.
- \(a\) - Semi-major axis of planetary orbit [stellar radii].
- \(i\) - Inclination of orbit in degrees (90 deg is edge on view).
- `linLib` - Linear limb-darkening coefficient.
- `quadLimb` - Quadratic limb-darkening coefficient.
- `T0` - Time offset of transit center.
- `per` - Period of planetary orbit.
- `b` - Describes the flux ratio between a stellar companion and the main star (default is 0).

This class inherits the functionality of funcFit’s OneDFit object. You can, therefore, set and get the parameter using the brackets:

e.g., `pallc["p"] = 0.12345`

Warning: Time units have to be consistent.

Methods

---

\(^1\) Pal 2008, “Properties of analytic transit light-curve models”, 2008MNRAS.390..281P.
### MCMCautoParameters(ranges[, picky, ...])
Convenience function to generate parameters for MCMC fit.

### addConditionalRestriction(*args)
Define a conditional restriction.

### assignValue(specval)
Assign new values to variables.

### assignValues(specval)
Assign new values to variables.

### autoFitMCMC(x, y, ranges[, picky, stepsizex,...])
Convenience function to using auto-generated sampling parameters in MCMC.

### availableParameters()
Provides a list of existing parameters.

### delRestriction(parName)
Delete restriction

### description([parenthesis])
Returns a description of the model based on the names of the individual components.

### errorConfInterval(par[, dstat, statTol, ...])
Calculate confidence interval for a parameter.

### evaluate(time)
Calculate a light curve according to the analytical models given by Pal 2008.

### fit(x, y[, yerr, X0, minAlgo, mA, ...])
Carries out a fit.

### fitEMCEE([x, y, nwalker, priors, ...])
MCMC sampling using emcee package.

### fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])
Carry out MCMC fit/error estimation.

### freeParamNames()
Get the names of the free parameters.

### freeParameters()
Get names and values of free parameters.

### freeze(specifiers)
Consider variables free to float.

### frozenParameters()
Get names and values of frozen parameters.

### getRelationsOf(specifier)
Return relations of a variable.

### getRestrictions()
Get all restrictions.

### hasVariable(specifier)
Determine whether the variable exists.

### numberOfFreeParams()
Get number of free parameters.

### parameterSummary([toScreen, prefix, sorting])
Writes a summary of the parameters in text form.

### parameters()
Obtain parameter names and values.

### relate(dependentVar, independentVars[, func])
Define a relation.

### removeConditionalRestriction(*args)
Remove an existing conditional constraint.

### renameVariable(oldName, newName)
Change name of variable.

### restoreState(resource)
Restores parameter values from file or dictionary.

### saveState(*args, **kwargs)
Save the state of the fitting object.

### setObjectiveFunction([miniFunc])
Define the objective function.

### setPenaltyFactor(penalFac)
Change the penalty factor.

### setRestriction(restricts)
Define restrictions.

### setRootName(root[, rename])
Define the root name of the model.

### showConditionalRestrictions(**kwargs)
Show conditional restrictions.

### steppar(pars, ranges[, extractFctVal, quiet])
Allows to step a parameter through a specified range.

### thaw(specifiers)
Consider variables fixed.

### untie(parName[, forceFree])
Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.

### updateModel()
Recalculate the model using current settings.

### whichEllInts()
Says which module is used for the evaluation of the elliptical functions.

---

**evaluate**(time)
Calculate a light curve according to the analytical models given by Pal 2008.

**Parameters**

- **time** [array] An array of time points at which the light curve shall be calculated.

.. note:: time = 0 -> Planet is exactly in the line of sight (phase = 0).
Returns

**Model** [array] The analytical light curve is stored in the property *lightcurve*.

**whichEllInts**()
Says which module is used for the evaluation of the elliptical functions.

---

**Note:** The support for BOOST has to be added manually!

---

Returns

**Identifier** [string] Either ‘boost’ or ‘mpmath’ depending on whether the *BOOST* libraries are used or the *mpmath* python module.

---

Full Keplerian orbit

**class** `PyAstronomy.modelSuite.XTran.palTrans.PalLCKep(ke=None, collisionCheck=False)`

Analytical transit light-curves using the formulae provided by Pal 2008.

More information on the Keplerian orbit can be found here: *Calculate a Keplerian (two body) orbit*

---

**Note:** The evaluation of elliptical integrals is essential in calculating the transit model. While both the *mpmath* module and the *Boost* libraries implement those integrals, it is the Boost library, which evaluates them by far more quickly. Yet, the support for Boost has to be added manually.

---

**Model parameters:**

- *p* - Radius ratio between planet and star.
- *a* - Semi-major axis of planetary orbit [stellar radii].
- *w* - Longitude of periastron [deg].
- *Omega* - Longitude of the ascending node [deg].
- *e* - Orbital eccentricity.
- *i* - Inclination of orbit in degrees (90 deg is *edge on* view).
- *linLib* - Linear limb-darkening coefficient.
- *quadLimb* - Quadratic limb-darkening coefficient.
- *tau* - Time of periastron passage.
- *per* - Period of planetary orbit.
- *b* - Describes the flux ratio between a stellar companion and the main star (default is 0).

This class inherits the functionality of funcFit’s OneDFit object. You can, therefore, set and get the parameter using the brackets:

```
e.g., pallc[“p”] = 0.12345
```

---

**Warning:** Time units have to be consistent.

---

Parameters

2.5. Model fitting
ke [None, optional] If an instance of KeplerEllipse is given, the parameters a, i, tau, per, w, Omega, and e will be adopted from it.

collisionCheck [boolean, optional] If set True, it will be checked whether the two bodies collide on the current orbit.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(time)</td>
<td>Calculate a light curve according to the analytical models given by Pal 2008.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAa, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE((x, y, xerr, nwalker, priors, ...))</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toList, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>
Table 60 – continued from previous page

| whichEllInts() | Says which module is used for the evaluation of the elliptical functions. |

evaluate(time)

Calculate a light curve according to the analytical models given by Pal 2008.

**Parameters**

- **time** [array] An array of time points at which the light curve shall be calculated.

**Returns**

- **Model** [array] The analytical light curve is stored in the property *lightcurve*.

**Transit model with finite integration time**

If the instrumental integration times are significant, the observed transit light-curve can show an apparent distortion, which is the consequence of integrating over a fraction of the transit (e.g., Kipping). To take this effect into account, the *PalFitRebin* class can be used, which calculates the model on a finer grid and than averages to obtain the actual light curve:

```python
PyAstronomy.modelSuite.XTran.palTrans.PallC_Rebin
alias of PyAstronomy.funcFit.modelRebin.turnIntoRebin.<locals>._ModelRebin
```

**Adding Boost support**

The Boost libraries are a set of high-quality peer-reviewed c++ libraries written and maintained by excellent programmers.

Boost offers both a convenient interface to create c++/c written Python modules and an implementation of the elliptical integrals of the third kind.

Boost support is mainly an issue of speed. A major part of the evaluation of the transit model is evaluating elliptic integrals. This can conveniently be done, using the *mpmath* module, which is written in pure Python. Yet, the evaluation is much faster, if the Boost library is applied.

**How can Boost support be activated?**

To enable boost support, unfortunately, a little bit of manual work is needed. This may require some knowledge about compiling c++ code and the Boost libraries in particular. In the *palTrans* directory of the source distribution of PyAstronomy, you will find the *ellint* subdirectory, which contains two files ell.cpp and a makefile_template.

First you have to adapt the makefile_template to contain the library paths appropriate for your system; then rename the makefile_template and call in “makefile”. Compiling the ell.cpp file using the makefile, produces a shared object library *ell.so*, if successful. This library file has, finally, to be copied to the installation directory of PyAstronomy.modelSuite.palTrans so that it can be imported by this module.

You can use the *whichEllInts()* method of the PaLLC class to check whether Boost support is present.

**Examples**

The following examples should give a reasonably impression of how the transit model class can be used.

---

1 Kipping 2010, “Binning is sinning: morphological light-curve distortions due to finite integration time”, 2010MNRAS.408.1758K

---

2.5. Model fitting
Calculating a model

In this first example, we demonstrate how to calculate a model transit light-curve.

```python
from PyAstronomy.modelSuite import palTrans
import matplotlib.pyplot as plt
import numpy as np

# Create a PalLC instance
plc = palTrans.PalLC()

# Set parameter values
plc['p'] = 0.1  # Planet radius / Stellar radius
plc['per'] = 1.0  # Orbital period
plc['a'] = 2.0  # Large semi major axis [R_S]
plc['i'] = 90.0  # Orbital inclination [deg]
# Specify limb darkening
# (quadratic limb-darkening law)
plc['linLimb'] = 0.4
plc['quadLimb'] = 0.2
# Specify T0 (central time of transit)
plc['T0'] = -0.1
# Specify binary contribution
plc['b'] = 0.0

# Check the parameters
plc.parameterSummary()

# Create a time axis
time = np.arange(1000)/1000.0 - 0.5

# Calculate the light curve using above set
# model parameters
lightcurve = plc.evaluate(time)

# Plot the result
plt.plot(time, lightcurve, 'bp')
plt.show()
```

Fitting a transit

The leading half of this example is very similar to the first one; we only use a smaller number of point for the time axis. The second half demonstrates how the fitting interface is invoked (more examples using simpler models can are shown in the `funcFit tutorial`).

**Warning:** If you are using `mpmath` for evaluation of the elliptical integrals, the calculations will be quite slow.

```python
from __future__ import print_function, division
from PyAstronomy.modelSuite import palTrans
import matplotlib.pyplot as plt
import numpy as np

# Create a PalLC instance
```
plc = palTrans.PalLC()

# Set parameter values
plc['p'] = 0.1  # Planet radius / Stellar radius
plc['per'] = 1.0  # Orbital period
plc['a'] = 7.5  # Large semi major axis [R_S]
plc['i'] = 90.0  # Orbital inclination [deg]
# Specify limb darkening
# (quadratic limb-darkening law)
plc['linLimb'] = 0.4
plc['quadLimb'] = 0.2
# Specify T0 (central time of transit)
plc['T0'] = -0.1
# Specify binary contribution
plc['b'] = 0.0

# Check the parameters
print("Input parameters: ")
plc.parameterSummary()

# Create a time axis
time = np.arange(100)/100.0 * 0.2 - 0.2

# Calculate the light curve using above set
# model parameters
lc = plc.evaluate(time)

# Save the result and add some noise
flux = lc + np.random.normal(0.0, 0.002, time.size)

# Now lets try to recover what we put in
# Choose some "guess" parameters
plc['p'] = 0.1  # Planet radius / Stellar radius
plc['per'] = 1.0  # Orbital period
plc['a'] = 7.5  # Large semi major axis [R_S]
plc['i'] = 90.0  # Orbital inclination [deg]
# Specify limb darkening
# (quadratic limb-darkening law)
plc['linLimb'] = 0.4
plc['quadLimb'] = 0.2
# Specify T0 (central time of transit)
plc['T0'] = -0.08
# Specify binary contribution
plc['b'] = 0.0

# Assume we want to fit "p", "a", "i", and "T0"
plc.thaw(['T0', 'i'])

# Before we start fitting, check how the elliptical integrals
# are evaluated (mpmath or Boost)
print("Which elliptical integrals are used?: ", plc.whichEllInts())

# Carry out the fit
plc.fit(time, flux, yerr=np.ones(time.size)*0.002)
print("Fit parameters: ")
plc.parameterSummary()
Obtain a model taking finite integration time into account

This example shows how to use the `PalLC_Rebin` class to take finite integration times and the resulting light-curve distortion into account. This example is very similar to the first one.

```python
from PyAstronomy.modelSuite import palTrans
import matplotlib.pyplot as plt
import numpy as np

# Create a PalLC_Rebin instance
plc = palTrans.PalLC_Rebin()

# Set parameter values
plc['p'] = 0.1  # Planet radius / Stellar radius
plc['per'] = 1.0  # Orbital period
plc['a'] = 2.0  # Large semi major axis [R_S]
plc['i'] = 90.0  # Orbital inclination [deg]
# Specify limb darkening
# (quadratic limb-darkening law)
plc['linLimb'] = 0.8
plc['quadLimb'] = 0.2
# Specify T0 (central time of transit)
plc['T0'] = -0.1
# Specify binary contribution
plc['b'] = 0.0

# Check the parameters
plc.parameterSummary()

# Create a time axis
time = np.arange(50)/50.0 - 0.51
# Specify oversampling parameters.
# Here use 10 points per observed bin.
plc.setRebinArray_Ndt(time, 10, time[1]-time[0])

# Calculate the light curve using above set
# model parameters
lc = plc.evaluate(time)

# Plot the result (both the overbinned and final
# model light-curves)
plt.plot(plc.rebinTimes, plc.unbinnedModel, 'b.-')
plt.plot(time, plc.binnedModel, 'rd--')
plt.legend(['Overbinned LC', 'Averaged LC'])
plt.show()
```
Analytical transit model (Mandel & Agol 2002)

The forTrans module provides easy access to the FORTRAN routines by Mandel & Agol 2002\(^1\), which can be used to calculate transit light curves. While the FORTRAN code ensures fast computation of the light curve, PyAstronomy supplies funcFit’s fitting framework.

**Note:** The code contained within the files occultnl.f and occultquad.f is used with kind permission of the authors. It is described in\(^1\); further explanations are given in Eastman et al. 2013 (PASP 125, 83).

The MandelAgolLC class

The MandelAgolLC model can be used to calculate transit light curves with quadratic or non-linear limb darkening based on a circular or general Keplerian orbit.

```python
class PyAstronomy.modelSuite.XTran.forTrans.mandelAgol.MandelAgolLC(orbit='circular', ld='quad', collCheck=True)
```

Analytical transit light-curves using the formulae provided by Mandel & Agol 2002.

**Note:** The computation of transit light curves is done using the external occultquad FORTRAN library.

This library can be installed, e.g., via

`pip install PyAstronomy_ext`

It can also be compiled manually using SciPy’s f2py wrapper (http://www.scipy.org/F2py). Simply go to the forTrans directory of the source distribution of PyAstronomy, then invoke

```
f2py -c occultquad.pyf occultquad.f
f2py -c occultnl.pyf occultnl.f
```

**Model parameters**

The set of parameters specifying this model depends on: the type of orbit chosen (circular or keplerian) and the type of limb darkening chosen (quadratic or non-linear).

More information on the Keplerian orbit can be found here: *Calculate a Keplerian (two body) orbit*

*Orbital model parameters (circular orbit):*

- \(p\) - Radius ratio between planet and star.
- \(a\) - Semi-major axis of planetary orbit [stellar radii].
- \(i\) - Inclination of orbit in degrees (90 deg is *edge on* view).
- \(T0\) - Time offset of transit center.
- \(per\) - Period of planetary orbit.
- \(b\) - Describes the flux ratio between a stellar companion and the main star (default is 0).

*Orbital model parameters (Keplerian orbit):*

- \(p\) - Radius ratio between planet and star.
- \(a\) - Semi-major axis of planetary orbit [stellar radii].

---

• **$i$** - Inclination of orbit in degrees (90 deg is edge on view).
• **$per$** - Period of planetary orbit.
• **$b$** - Describes the flux ratio between a stellar companion and the main star (default is 0).
• **$tau$** - Time of periapsis passage.
• **$Omega$** - Longitude of the ascending node [deg].
• **$w$** - Argument of periapsis [deg]. Note that the longitude if periapsis is given by $Omega+w$.
• **$e$** - Orbital eccentricity (0-1).

**Limb darkening parameters (quadratic):**

• **linLib** - Linear limb-darkening coefficient.
• **quadLimb** - Quadratic limb-darkening coefficient.

**Limb darkening parameters (non-linear):**

• **$a1$** - Non-Linear limb-darkening coefficient.
• **$a2$** - Non-Linear limb-darkening coefficient.
• **$a3$** - Non-Linear limb-darkening coefficient.
• **$a4$** - Non-Linear limb-darkening coefficient.

**Limb-darkening laws**

The quadratic limb-darkening law is given by:

$$\frac{I(\mu)}{I(1)} = 1 - \text{linLib} \times (1 - \mu) - \text{quadLimb} \times (1 - \mu)^2$$

The non-linear limb-darkening law is given by:

$$\frac{I(\mu)}{I(1)} = 1 - \sum_{n=1}^{4} a_n (1 - \mu^{n/2})$$

**Warning:** Time units have to be consistent.

**Parameters**

- **orbit** [string, {“circular”, “keplerian”}, optional] Determines whether a circular or full keplerian planetary orbit is used in the calculations. The default is a circular orbit.
- **ld** [string, {“quad”, “nl”}] Determines whether quadratic or non-linear limb darkening is used. The default is quadratic limb darkening.
- **collCheck** [boolean, optional] If True (default), the model will check whether there is a physical collision between the star and the planet on evaluating the model and raises an exception when there is one.

**Methods**

- **MCMCautoParameters(ranges[, picky, ...])** Convenience function to generate parameters for MCMC fit.
- **addConditionalRestriction(*args)** Define a conditional restriction.
Example: Calculate model light curve (circular orbit, quadratic limb darkening)

```python
# Import some unrelated modules
import numpy as np
import matplotlib.pyplot as plt
# ... and now the forTrans module
from PyAstronomy.modelSuite import forTrans as ft

# Create MandelAgolLC object with
circular orbit and quadratic limb darkening
```
Example: Calculate model light curve (keplerian orbit, quadratic limb darkening)

```python
# Import some unrelated modules
import numpy as np
import matplotlib.pyplot as plt
# ... and now the forTrans module
from PyAstronomy.modelSuite import forTrans as ft

# Create MandelAgolLC object with
# keplerian orbit and quadratic limb darkening
ma = ft.MandelAgolLC(orbit="keplerian", ld="quad")

# See the available parameters and their current values
ma.parameterSummary()

# Set parameters
ma["per"] = 0.2
ma["i"] = 88.76
ma["a"] = 6.5
ma["p"] = 0.16
ma["linLimb"] = 0.47
ma["quadLimb"] = 0.24
ma["b"] = 0.
ma["e"] = 0.75
ma["w"] = 127.
ma["Omega"] = 3.9

# Choose some time axis
time = np.linspace(0, 0.5, 1000)

# ... and calculate model
y = ma.evaluate(time)

# Let's see what happened ...
plt.plot(time, y, 'b.')</nplt.show()
```

(continues on next page)
Example: Comparing limb-darkening laws

```python
# Import some modules
import numpy as np
import matplotlib.pyplot as plt
# ... and now the forTrans module
from PyAstronomy.modelSuite import forTrans as ft

# First, let's compute a transit model using
# quadratic limb-darkening prescription.
ma = ft.MandelAgolLC(ld="quad")

# Set parameters. The LD coefficients are taken
# from Claret 2011 for a solar-metallicity star
# with Teff=6000 K and logg=4.5.
ma["per"] = 0.2
ma["i"] = 90.0
ma["a"] = 6.5
ma["T0"] = 0.5
ma["p"] = 0.16
ma["linLimb"] = 0.0479
ma["quadLimb"] = 0.2716
ma["b"] = 0.

# Choose some time axis
time = np.linspace(0, 0.2, 1000)

# ... and calculate model
yQLD = ma.evaluate(time)

# Now, let's compute a transit model with
# non-linear limb-darkening prescription
# for the same stellar parameters.
maNL = ft.MandelAgolLC(ld="nl")
maNL["per"] = 0.2
maNL["i"] = 90.0
maNL["a"] = 6.5
maNL["T0"] = 0.5
maNL["p"] = 0.16
maNL["a1"] = 0.5335
maNL["a2"] = 0.0793
maNL["a3"] = -0.3466
maNL["a4"] = 0.1609
maNL["b"] = 0.
```

(continues on next page)
# Let's compare both models...
plt.plot(time, yQLD, '-', label="Quadratic LD")
plt.plot(time, yNLLD, 'd', label="Non-linear LD")
plt.legend()
plt.show()

Non-linear limb-darkening - The MandelAgolNLLC class

Warning: The mandelAgolNL is considered deprecated. Its functionality has been absorbed in the MandelAgolLC model.

class PyAstronomy.modelSuite.XTran.forTrans.mandelAgolNL.MandelAgolNLLC:
    Calculate and fit analytical transit light-curves using the formulae provided by Mandel & Agol 2002. This model uses the non-linear limb-darkening prescription by Claret 2011:

\[
\frac{I(\mu)}{I(1)} = 1 - \sum_{n=1}^{4} a_n(1 - \mu^{n/2})
\]

Note: The computation of transit light curves is done using the external occultnl FORTRAN library. This library must be compiled manually using SciPy’s f2py wrapper (http://www.scipy.org/F2py). Simply go to the forTrans directory of the source distribution of PyAstronomy, then invoke

f2py -c occultnl.pyf occultnl.f

Model parameters:
- \( p \) - Radius ratio between planet and star.
- \( a \) - Semi-major axis of planetary orbit [stellar radii].
- \( i \) - Inclination of orbit in degrees (90 deg is edge on view).
- \( a_1 \) - Non-Linear limb-darkening coefficient.
- \( a_2 \) - Non-Linear limb-darkening coefficient.
- \( a_3 \) - Non-Linear limb-darkening coefficient.
- \( a_4 \) - Non-Linear limb-darkening coefficient.
- \( T_0 \) - Time offset of transit center.
- \( per \) - Period of planetary orbit.
- \( b \) - Describes the flux ratio between a stellar companion and the main star (default is 0).

Methods
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(time)</td>
<td>Calculate a light curve according to the analytical models given by Mandel &amp; Agol 2002.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

**Thin shell transit-model**

Transit light-curves usually have one minimum and a “U” shape. This can be different, e.g., if optically thin chromospheric emission-lines are considered. In this case, most emission may come from the stellar limb resulting in a “W”-shaped transit-profile.

The model has been presented by Schlawin et al.¹.

The model class

class PyAstronomy.modelSuite.XTran.limBrightTrans.LimBrightTrans

Planetary transit light-curves for spherical shell model.

This class implements a model calculating the light curve of a planet transiting an optically thin spherical shell of negligible thickness (e.g., a stellar chromosphere).

The model provided by Schlawin et al. 2010 assumes that the thickness of the shell is much smaller than the size of the planet. The shell is optically thin and thus provides natural limb-brightening. The obscured part of the stellar surface is calculated based on computing the volume of the intersection of a sphere with a cylinder and then taking a partial derivative with respect to the radius of the sphere to find its surface area.

The code closely follows the IDL procedure located at http://www.astro.washington.edu/agol/.

Fit parameters:

- \( p \) - Rp/Rs (ratio of planetary and stellar radius)
- \( a \) - Semi-major axis of planetary orbit [stellar radii].
- \( per \) - Orbital period [d]
- \( T0 \) - Central transit time
- \( i \) - Inclination of orbit [rad]

By default all parameters remain frozen.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description(parenthesis)</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(time)</td>
<td>Calculate a light curve according to the analytical model</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAa, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 63 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameterSummary(itolScreen, prefix, sorting)</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction(*miniFunc)</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

**evaluate**(time)

Calculate a light curve according to the analytical model by Schlawin et al. 2010.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>[array] An array of time points at which the light curve shall be calculated</td>
</tr>
</tbody>
</table>

**Returns**

<table>
<thead>
<tr>
<th>Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>[array] The analytical light curve is stored in the property lightcurve.</td>
</tr>
</tbody>
</table>

**Notes**

<table>
<thead>
<tr>
<th>Note</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>time = 0 -&gt; Planet is exactly in the line of sight (phase = 0).</td>
<td></td>
</tr>
</tbody>
</table>

Example code - Calculate model light curve

```python
# Import some unrelated modules
import matplotlib.pylab as mpl
from numpy import arange
# ... and now the LimBrightTrans module
from PyAstronomy.modelSuite import LimBrightTrans

# Create LimBrightTransit object
lbt = LimBrightTrans()
# Set parameters
lbt["p"] = 0.08
lbt["a"] = 6.70
lbt["i"] = 87.84
lbt["T0"] = 4.0
lbt["per"] = 10.
```

(continues on next page)
# Choose some time axis and calculate model

time = arange(3., 5., 0.001)
y = lbt.evaluate(time)

# Let's see what happened...
mpl.ylabel("Relative Flux")
mpl.xlabel("Time")
mpl.plot(time, y, 'r-')
mpl.show()

Rossiter McLaughlin curves

The Rossiter-McLaughlin effect (Ohta et al. 2005)

The Rossiter McLaughlin (RMcL) effect is an apparent RV shift of the stellar spectrum during planetary transits. As the planet moves across the stellar disk, it eclipses different surface sections with different radial velocities caused by stellar rotation. Here, we provide an implementation of the analytical radial-velocity model presented by Ohta et al. 2005\(^1\).

Contents:

Rossiter-McLaughlin model class

The class \texttt{RmcL} implements the analytical model presented by Ohta et al. 2005. For an elliptical orbit, the class \texttt{RmcLell} can be used.

- \texttt{RmcL (circular orbit)}
- \texttt{RmcLell (elliptical orbit)}

\texttt{RmcL (circular orbit)}

\begin{verbatim}
class PyAstronomy.modelSuite.RmcL:
    Analytical Rossiter-McLaughlin effect.
    
    This class implements the analytical model radial velocity (RV) curves for the Rossiter-McLaughlin effect given by Ohta et. al 2005.
    
    \textit{Fit parameters}:
    
    \begin{itemize}
    \item epsilon - linear limb dark
    \item gamma - Rp/Rs (ratio of planetary and stellar radius)
    \item P - Orbital period [d]
    \item T0 - Central transit time
    \item i - Inclination of orbit [rad]
    \item Is - Inclination of stellar rotation axis [rad]
    \item Omega - Angular rotation velocity (star) [rad/s]
    \end{itemize}
\end{verbatim}

• lambda - Sky-projected angle between stellar rotation axis and normal of orbit plane [rad]
• a - Semi major axis [stellar radii]

By default all parameters remain frozen.

**Note:** According to the input parameter units, the units of the model RV curve are **stellar-radii per second**.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(xOrig)</td>
<td>Calculates and returns RV curve according to current model parameters.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 64 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
</tr>
<tr>
<td>W2</td>
</tr>
<tr>
<td>W3</td>
</tr>
<tr>
<td>W4</td>
</tr>
<tr>
<td>Xp</td>
</tr>
<tr>
<td>XpVec</td>
</tr>
<tr>
<td>Zp</td>
</tr>
<tr>
<td>etap</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>planetDistance</td>
</tr>
<tr>
<td>rho</td>
</tr>
<tr>
<td>rhoFromVec</td>
</tr>
<tr>
<td>trueAnomaly</td>
</tr>
<tr>
<td>x0</td>
</tr>
<tr>
<td>xc</td>
</tr>
<tr>
<td>z0</td>
</tr>
<tr>
<td>zeta</td>
</tr>
</tbody>
</table>

`evaluate (xOrig)`
Calculates and returns RV curve according to current model parameters.

**Note:** The units of the model RV curve are **stellar-radii per second**.

**Parameters**

- **xOrig** [array] The time stamps at which to calculate the model RV curve. Note that the orbit period and central transit time are used to convert time into “true anomaly”.

**RmcLell (elliptical orbit)**

**class PyAstronomy.modelSuite.RmcLell**
Analytical Rossiter-McLaughlin effect.

This class implements the analytical model radial velocity (RV) curves for the Rossiter-McLaughlin effect given by *Ohta et. al 2005*.

**Fit parameters:**

- epsilon - Linear limb darkening coefficient
- gamma - Rp/Rs (ratio of planetary and stellar radius)
- P - Orbital period [d]
- tau - Time of periastron passage [same as orbital period]
- i - Inclination of orbit [rad]
- Is - Inclination of stellar rotation axis [rad]
• Omega - Angular rotation velocity (star) [rad/s]
• \(\lambda\) - Sky-projected angle between stellar rotation axis and normal of orbit plane [rad]
• a - Semi major axis [stellar radii]
• \(w\) - Argument of periapsis [rad]
• e - Eccentricity

**Note:** In the case of zero eccentricity, a value of -90 deg for the argument of periastron (\(w\)) makes the time of periastron (\(\tau\)) numerically identical with the central transit time of the circular case (T0).

**Note:** According to the input parameter units, the units of the model RV curve are **stellar-radii per second**.

### Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MCMCautoParameters(ranges[, picky, ...])</code></td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td><code>addConditionalRestriction(*args)</code></td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td><code>assignValue(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>assignValues(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>autoFitMCMC(x, y, ranges[, picky, steps, ...])</code></td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td><code>availableParameters()</code></td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td><code>delRestriction(parName)</code></td>
<td>Delete restriction.</td>
</tr>
<tr>
<td><code>description([parenthesis])</code></td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td><code>errorConfInterval(par[, dstat, statTol, ...])</code></td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td><code>evaluate(xOrig)</code></td>
<td>Calculates and returns RV curve according to current model parameters.</td>
</tr>
<tr>
<td><code>fit(x, y, X0, minAlgo, mA, ...)</code></td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td><code>fitEMCEE([x, y, err, nwalker, priors, ...])</code></td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td><code>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</code></td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td><code>freeParamNames()</code></td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td><code>freeParameters()</code></td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td><code>freeze(specifiers)</code></td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td><code>frozenParameters()</code></td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td><code>getRelationsOf(specifier)</code></td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td><code>getRestrictions()</code></td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td><code>hasVariable(specifier)</code></td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td><code>numberOfFreeParams()</code></td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td><code>parameterSummary([toScreen, prefix, sorting])</code></td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td><code>parameters()</code></td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td><code>relate(dependentVar, independentVars[, func])</code></td>
<td>Define a relation.</td>
</tr>
<tr>
<td><code>removeConditionalRestriction(*args)</code></td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td><code>renameVariable(oldName, newName)</code></td>
<td>Change name of variable.</td>
</tr>
<tr>
<td><code>restoreState(resource)</code></td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td><code>saveState(*args, **kwargs)</code></td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td><code>setObjectiveFunction([miniFunc])</code></td>
<td>Define the objective function.</td>
</tr>
</tbody>
</table>
Table 65 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setPenaltyFactor(penalFac)</code></td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td><code>setRestriction(restricts)</code></td>
<td>Define restrictions.</td>
</tr>
<tr>
<td><code>setRootName(root[, rename])</code></td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td><code>showConditionalRestrictions(**kwargs)</code></td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><code>steppar(pars, ranges[, extractFctVal, quiet])</code></td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><code>thaw(specifiers)</code></td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

Parameters

- `xOrig` [array] The time stamps at which to calculate the model RV curve. Note that the orbit period and central transit time are used to convert time into “true anomaly”.

Examples

In the following, we provide examples demonstrating the use of the Rossiter McLaughin model.

Note: For a more detailed introduction on the fitting see the `funcFit` tutorial.

Calculating models

The following example demonstrates how to calculate models.

Circular orbit

```python
# Import some unrelated modules
from numpy import arange, pi
import matplotlib.pyplot as plt
# ... and the model suite
from PyAstronomy import modelSuite as ms
# Create Rossiter-McLaughlin object
```

Note: The units of the model RV curve are **stellar-radii per second**. 
Elliptical orbit

```python
# Import some unrelated modules
from numpy import arange, pi
import matplotlib.pylab as plt
# ... and the model suite
from PyAstronomy import modelSuite as ms

# Create Rossiter-McLaughlin object (circular orbit)
rmcl = ms.RmcL()
# and one for an elliptical orbit
rmel = ms.RmcLell()

# Assign parameter values
rmcl.assignValue({"a": 6.7, "lambda": 7.2/180.0*pi, "epsilon": 0.5,
                   "P": 1.74, "T0": 0.2, "i": 87.8/180.*pi,
                   "Is": 90.0/180.0*pi, "Omega": 1.609e-5, "gamma": 0.2})
rmel.assignValue({"a": 6.7, "lambda": 7.2/180.0*pi, "epsilon": 0.5,
                   "P": 1.74, "tau": 0.2, "i": 87.8/180.*pi, "w": -90/180.*pi,
                   "e": 0.05, "Is": 90.0/180.0*pi, "Omega": 1.609e-5, "gamma": 0.2})

# Choose some time axis and calculate model
time = arange(100)/100.0 * 0.2 + 0.1
rvc = rmcl.evaluate(time)
rve = rmel.evaluate(time)

# Let's see what happened...
plt.ylabel("Radial velocity [stellar-radii/s]")
plt.xlabel("Time [d]")
plt.plot(time, rvc, 'b.-', label="circular")
plt.plot(time, rve, 'r.-', label="elliptical")
plt.legend()
plt.show()
```

Fitting a model

This example is an extension of the first. It demonstrates how a model fit can be carried out.
Circular orbit—radial-velocity shift

The radial velocity is the radial component of the velocity of a source relative to an observer and is usually inferred spectroscopically. Light from an object being part of a binary system and orbiting the common center of mass will be subject to the Doppler effect. Given a circular orbit, its radial velocity shift will show sinusoidal variations determined by the object’s orbital elements.

class PyAstronomy.modelSuite.radVel.SinRadVel
    Spectroscopic radial-velocity (RV) shift due to circular orbital motion.

Fit parameters:

• P - float, Orbital period [d]
• T0 - float, Central transit time [d]
• K - float, radial velocity semi-amplitude [km/s]
• rv0 - float, constant offset in radial velocity [km/s]
By default all parameters remain frozen.

The convention is that at phase zero, also the orbital radial velocity is zero. With increasing phase the object becomes bluer first.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MCMCautoParameters(ranges[, picky, ...])</code></td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td><code>addConditionalRestriction(*args)</code></td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td><code>assignValue(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>assignValues(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</code></td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td><code>availableParameters()</code></td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td><code>delRestriction(parName)</code></td>
<td>Delete restriction.</td>
</tr>
<tr>
<td><code>description([parenthesis])</code></td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td><code>errorConfInterval(par[, dstat, statTol, ...])</code></td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td><code>evaluate(x)</code></td>
<td>Calculates and returns radial velocity shift according to current model parameters.</td>
</tr>
<tr>
<td><code>fit(x, y[, yerr, X0, minAlgo, mA, ...])</code></td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td><code>fitEMCEE([x, y, yerr, nwalker, priors, ...])</code></td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td><code>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</code></td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td><code>freeParamNames()</code></td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td><code>freeParameters()</code></td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td><code>freeze(specifiers)</code></td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td><code>frozenParameters()</code></td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td><code>getRelationsOf(specifier)</code></td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td><code>getRestrictions()</code></td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td><code>hasVariable(specifier)</code></td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td><code>numberOfFreeParams()</code></td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td><code>parameterSummary([toScreen, prefix, sorting])</code></td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td><code>parameters()</code></td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td><code>relate(dependentVar, independentVars[, func])</code></td>
<td>Define a relation.</td>
</tr>
<tr>
<td><code>removeConditionalRestriction(*args)</code></td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td><code>renameVariable(oldName, newName)</code></td>
<td>Change name of variable.</td>
</tr>
<tr>
<td><code>restoreState(resource)</code></td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td><code>saveState(*args, **kwargs)</code></td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td><code>setObjectiveFunction([miniFunc])</code></td>
<td>Define the objective function.</td>
</tr>
<tr>
<td><code>setPenaltyFactor(penalFac)</code></td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td><code>setRestriction(restricts)</code></td>
<td>Define restrictions.</td>
</tr>
<tr>
<td><code>setRootName(root[, rename])</code></td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td><code>showConditionalRestrictions(**kwargs)</code></td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><code>steppar(pars, ranges[, extractFctVal, quiet])</code></td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><code>thaw(specifiers)</code></td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>
**evaluate**(*x*)

Calculates and returns radial velocity shift according to current model parameters.

**Parameters**

*x* [array] The time stamps at which to calculate the model RV curve.

---

**Example code - Fit RV curve**

```python
from __future__ import print_function, division
# Import some unrelated modules
from numpy import arange, random, ones
import matplotlib.pylab as plt
# ... and now the radVel module
from PyAstronomy.modelSuite import radVel as rv

# Create Radial Velocity object
r = rv.SinRadVel()
# Set parameters
r.assignValue({"P": 1.8, "T0": 0.25, "K": 0.5, "rv0": 10.0})
# Choose some time axis and calculate model
time = arange(100)/100.0 * 3.0 - 1.5
y = r.evaluate(time)

# Create some faked data by adding noise
rvData = y + random.normal(0.0, 0.05, y.size)

# Randomize starting parameters for fit
for p, v in r.parameters().items():
    r[p] = v + (random.random() - 0.5) * v
# Show starting values
print("Starting values for fit:")
r.parameterSummary()

# Thaw all parameters
r.thaw(list(r.parameters().keys()))
# Start the fit
r.fit(time, rvData, yerr=ones(y.size)*0.05)

# Show fit results
print("Fitted values:")
r.parameterSummary()

# Let's see what happened...
plt.ylabel("Radial velocity [km/s]")
plt.xlabel("Radial velocity [d]")
plt.errorbar(time, rvData, yerr=ones(y.size)*0.05, fmt='b.')
plt.plot(time, y, 'r-')
plt.show()
```

---

**The Kepler ellipse models**

The Kepler ellipse models provide a fitting framework for the Kepler orbit. In particular, the models provide the mapping from time to Cartesian x, y, and z coordinates or velocities. Also a representation of the usual radial velocity curve is available. Inevitably, the models are somewhat redundant. The calculations are based on algorithms implemented in *PyAstronomy’s Astrolib*. 
KeplerEllipseModel: Location or velocity

The KeplerEllipseModel model allows to model positions and velocities along a Keplerian orbit. On creation of the model, one can choose which of the coordinate axes are of interest to the user. The coordinates or velocities will then be returned as a one-dimensional array as explained the `evaluate` method of the class.

Example: Evaluating and fitting model using MCMC

```python
from __future__ import print_function, division
from PyAstronomy.modelSuite import KeplerEllipseModel
import numpy as np
import matplotlib.pyplot as plt

# Create a model class instance
# In this case, we are only interested
# in the x- and z-components of the orbit
# solution.
kem = KeplerEllipseModel(relevantAxes="xz")

# Setting some guess parameters
kem["a"] = 7.8
kem["per"] = 12.3
kem["e"] = 0.07
kem["tau"] = 0.745
kem["Omega"] = 143.
kem["w"] = 0.2
kem["i"] = 92.0

# Evaluate the model
time = np.linspace(0, kem["per"], 20)
model = kem.evaluate(time)
# Note that the model has twice the number of points
# compared to the time axis. This is because it contains
# the data for two axes
print("Used " + str(len(time)) + " time points")
print("-> length of model: ", len(model))

# Isolating the model for the x-axis, i.e.,
# every second data point starting from the
# beginning.
xmodel = model[0::2]
# Isolating the model for the y-axis
```

(continues on next page)
ymodel = model[1::2]

# Use the model to obtain mock data
# by introducing some scatter
data = model + np.random.normal(0., 0.5, model.size)
# Plot the resulting "data"
plt.title("Kepler Ellipse Model --- Example")
plt.errorbar(data[0::2], data[1::2], xerr=np.ones(20)*0.5,
             yerr=np.ones(20)*0.5, fmt="bp")

# Use MCMC to sample from the posterior
# Specify free parameters
kem.thaw(["a", "per", "e", "tau", "Omega", "w", "i"])
# Specify starting values
X0 = {}
steps = {}
for p in kem.freeParameters():
    X0[p] = kem[p]

lims = {"a": [5., 10.], "per": [10., 15.], "e": [0., 1.], "tau": [0.5, 1.],
        "Omega": [0., 360.], "w": [-5., 5.], "i": [90., 95.]}

kem.fitMCMC(time, data, X0, lims, steps, yerr=np.ones(len(data))*0.5,
             iter=500, dbfile="kemExample.tmp")

# Plot the lowest deviance model
ldmodel = kem.evaluate(np.linspace(0, kem["per"], 200))
plt.plot(ldmodel[0::2], ldmodel[1::2], 'r--')
plt.show()

KeplerEllipseModel: API

class PyAstronomy.modelSuite.KeplerEllipseModel (relevantAxes='xyz', mode='pos')
A model of a Keplerian orbit.

This class uses the KeplerEllipse from the PyA's pyasl to calculate a Keplerian orbit. It may be used to fit complete 3d position or velocity information on the orbit; any individual axes may also be selected.

The constructor allows to specify relevant axes, which are those axes considered in the calculation. The actual (technical) model is, however, only one-dimensional. The values returned by evaluate have the order a1, b1, c1, a2, b2, c3, ... Where a, b, and c represent the first, second, and third axis and the number specifies the data point. Note that in this case, the resulting model has not the same number of points as the time axis.

Fit parameters
• a - The semi-major axis (same units as the data)
• per - The period (same time units as data)
• e - The eccentricity
• tau - Time of periapsis passage (same time units as data)
• Omega - Longitude of the ascending node [deg]
• w - Argument of periapsis [deg]
• $i$ - Inclination angle [deg]

**Parameters**

relevantAxes [string] A string containing any combination of x, y, and z. The string specifies the axes (and their order) to be considered in the calculations.

mode [string, {“pos”, “vel”}] Determines whether the output is positions or velocities. In this case, the units are determined by the units of major axis and time (e.g., AU per day).

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(t)</td>
<td>Calculates and returns model according to the current parameter values.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
</tbody>
</table>
untie(parName[, forceFree])

Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.

updateModel()

Recalculate the model using current settings.

evaluate(t)

Calculates and returns model according to the current parameter values.

Although more than one axis may be relevant the output will be one dimensional. If, e.g., the relevant axes are x and y, the order of the output will be x0, y0, x1, y1, ... .

Parameters

- t [array] Times at which to evaluate the model.

Radial velocity curves: KeplerRVModel

Radial velocity curves are among the most important tools to study planetary and stellar systems. The KeplerRVModel provides an implementation of a radial velocity curve model for one or more planets (or stars). Note, however, that mutual interactions between the constituents of the system are not taken into account.

Example: Best-fit RV model and error analysis using KeplerRVModel

The example demonstrates how to fit a radial velocity curve and posterior-based error estimation (and parameter estimation) using emcee.

```python
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy.modelSuite import KeplerRVModel
from PyAstronomy import funcFit as fuf

# Generate artificial data ...
jd = np.arange(100)
rv = 1.5 * np.sin(jd / 37.6 * 2.*np.pi)
# ... with some error
rverr = 0.5
rv += np.random.normal(0, rverr, len(jd))
rverr = np.ones(len(rv)) * 0.5

# Get RV model with one planet (mp) and a potential constant offset
# in RV (deg = 0)
krvm = KeplerRVModel(mp=1, deg=0)
# To obtain some useful estimate of the minimum mass of the companion,
# we must specify the mass of the star (in terms of solar masses)
krvm['mstar'] = 0.5

# Let us have a look at the available parameters.
# Note that not all are meant for fitting in this model (MA and a)!
# There is also not much use in fitting 'mstar'. It may, however, be
# used in combination with a prior to take into account its uncertainty in
# the estimates.
krvm.parameterSummary(sorting="ps")
```

(continues on next page)
# We specify some guess parameters.
krvm["per1"] = 37.0
krvm["K1"] = 1.0
krvm["e1"] = 0.0
krvm["tau1"] = 17.0
krvm["w1"] = 180.

# Let us fit all of these but period ...
krmthaw(["K1", "tau1", "w1", "e1", "c0"])
# ... and now also the period
krmthaw(["per1"])
# and then get the best-fit model
kmo = krvm.evaluate(jd)

# What about chi-square and RMS?
chi = np.sum((rv - krvm.model)**2 / rverr**2)
# Reduced chi-square
rchi = chi / (len(rv) - len(krvm.freeParameters()))
print("chi-square and reduced chi-square: \$%6.3f, \$%6.3f\" % (chi, rchi))
rms = np.std(rv - krvm.model)
print("RMS: ", rms)
pl.title("RV data (blue) and model (red)")
plt.errorbar(jd, rv, yerr=rverr, fmt='b+')
plt.plot(jd, krvm.model, 'r-')
plt.show()

# Now let us do some posterior-based error analysis using MCMC

# Say, we want 20 burn-in iterations and, thereafter,
# 50 further iterations (per walker).
sampleArgs = {"iters": 50, "burn": 100}

# Specify a bounded uniform prior on the eccentricity. Note that restrictions are not
# automatically converted into priors (they may not ne uniform). Potentially further:
# e.g., on per1 may be required to prevent wandering into 'forbidden territory'.
priors = {"e1": fuf.FuFPrior("limuniform", upper=1, lower=0)}

# Start the sampling (ps could be used to continue the sampling)
ps = krvm.fitEMCEE(jd, rv, yerr=rverr, sampleArgs=sampleArgs, scales="e": 0.05),
       dbfile="chain1.emcee",
       priors=priors)

# Have a look at the posterior
ta = fuf.TraceAnalysis("chain1.emcee")

# What about the deviance (-2 log(Likelihood))
ta.plotTraceHist("deviance")
ta.show()

# Expectation value and highest probability density interval for eccentricity
print("Expectation value for eccentricity: ", ta.mean("e1"))
print("90% HPD for eccentricity: ", ta.hpd("e1", cred=0.9))
ta.show()
ta.show()

KeplerRVModel: API

class PyAstronomy.modelSuite.KeplerRVModel(mp=1, deg=0, msun=1.988547e+30, mJ=1.8986e+27, au=149597870700.0)

A model of a Keplerian orbit.

This class uses the KeplerEllipse from the PyA's pyasl to calculate radial velocities for a Keplerian orbit. All calculations are based on the negligible companion mass hypothesis (i.e., this model has been implemented with planetary systems in mind).

Note: Any planet with zero period will be ignored.

Note: ? is a placeholder for an integer larger zero, indicating the number of the planet (the total number is controlled by the mp keyword).

Fit parameters
• per? - The period (same time units as data)
• e? - The eccentricity
• tau? - Time of periapsis passage (same time units as data)
• w? - Argument of periapsis [deg]
• K? - Semi-amplitude of radial velocity (same units as data)
• mstar - Stellar mass in solar masses. This parameter is usually not fitted. It may be used to take into account the uncertainty on stellar mass in Bayesian (MCMC) analysis.

Derived parameters (not to be fitted)
• a? - The semi-major axis in AU
• MA? - Mean anomaly corresponding to time of first data point [deg]
• msini? - Minimum mass (msini) in Jupiter masses

Parameters

mp [int, optional] The number of planets considered in the model. Default is one. Note that signals are added, i.e., no interaction is taken into account.
deg [int, optional] Default is zero (i.e., a constant). The degree of a polynomial used to represent a systematic (non-periodic) evolution in the data.
msun [float, optional] Solar mass [kg]
mJ [float, optional] Jupiter mass [kg]
au [float, optional] Astronomical unit [m]

Methods
<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCMCautoParameters(ranges[, picky, ...])</td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td>addConditionalRestriction(*args)</td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td>assignValue(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>assignValues(specval)</td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description((parenthesis))</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAA, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE(x, y, yerr, nwalker, priors, ...)</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

### Line list based Gaussian spectral model

**class** PyAstronomy.modelSuite.LLGauss(lineList, uniSig=None, modelBinsize=0.005, useFastRB=True, verbose=False, onlyAbs=True)

A spectral model based on Gaussian profiles and a line list. This class provides a simple spectral model based on a number of Gaussian lines, whose strength may be fitted individually. Note that the EW of the lines is given by: \( A(n)^\text{lineScale} \), where \( A(n) \) is the area of the n-th
Gaussian. The scale parameter does not influence the EW of the Gaussians.

**Note:** The unit of the EWs given in the lineList needs to be the same as the wavelength units.

**Fit parameters:**

- lineScale - A common scaling of the area of all lines.
- scale - A scaling of the entire spectrum.
- eps - Linear limb-darkening coefficient.
- vrad - Radial velocity [km/s].
- vsini - Projected stellar rotational velocity [km/s].
- A{n} - The amplitudes (area) parameters of the individual Gaussians.
- sig{n} - The standard deviations of the individual Gaussian.
- mu{n} - The position of the individual Gaussians.

**Parameters**

- onlyAbs [boolean, optional] If True (default), restrictions will be applied, which prevent emission lines in the spectrum.
- lineList [array] An array with either two or three columns. The first column given the position of the lines, the second gives the EW of the lines, and the third—if present—gives the depth of the lines. The depth is the maximal depression of the continuum, e.g., a value of 0.96 means that the center of the line of 4% below the continuum. If the depth is given, the width of the individual Gaussians is determined from it, unless the uniSig is specified.
- uniSig [float, optional] Use “unified sigma”, i.e., the same width for all lines. Note that this flag overrules the “depth” column in the lineList, if it has been specified.
- modelBinsize [float, optional] Internally, the model should be calculated on a finer grid than the actual spectrum. This parameter specifies the used bin size, which is 0.005 by default.
- useFastRB [boolean, optional] Use the “fast” rotational broadening algorithm. This algorithm uses a wavelength-independent broadening kernel, which is considerably faster than considering the wavelength dependence. Setting this flag to False is necessary if you use very long wavelength ranges; by default it is True.
- verbose [boolean, optional] If True, the class print the current parameters during the evaluation.

**Methods**

- MCMCautoParameters(ranges[, picky, ...]) Convenience function to generate parameters for MCMC fit.
- addConditionalRestriction(*args) Define a conditional restriction.
- assignValue(specval) Assign new values to variables.
- assignValues(specval) Assign new values to variables.
- autoFitMCMC(x, y, ranges[, picky, stepsize, ...]) Convenience function to using auto-generated sampling parameters in MCMC.

Continued on next page
Table 69 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>availableParameters()</td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td>delRestriction(parName)</td>
<td>Delete restriction</td>
</tr>
<tr>
<td>description([parenthesis])</td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td>errorConfInterval(par[, dstat, statTol, ...])</td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td>evaluate(x)</td>
<td>Calculates the model for current parameters.</td>
</tr>
<tr>
<td>fit(x, y[, yerr, X0, minAlgo, mAa, ...])</td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td>fitEMCEE([x, y, yerr, nwalker, priors, ...])</td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td>fitMCMC(x, y, X0, Limits, Steps[, yerr, ...])</td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td>freeParamNames()</td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td>freeParameters()</td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td>freeze(specifiers)</td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td>getRelationsOf(specifier)</td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td>getRestrictions()</td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td>hasVariable(specifier)</td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td>numberOfFreeParams()</td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td>numberOfLines()</td>
<td>Get the number of lines in the model.</td>
</tr>
<tr>
<td>parameterSummary([toScreen, prefix, sorting])</td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td>parameters()</td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td>relate(dependentVar, independentVars[, func])</td>
<td>Define a relation.</td>
</tr>
<tr>
<td>removeConditionalRestriction(*args)</td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td>renameVariable(oldName, newName)</td>
<td>Change name of variable.</td>
</tr>
<tr>
<td>restoreState(resource)</td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td>saveState(*args, **kwargs)</td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td>setObjectiveFunction([miniFunc])</td>
<td>Define the objective function.</td>
</tr>
<tr>
<td>setPenaltyFactor(penalFac)</td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td>setRestriction(restricts)</td>
<td>Define restrictions.</td>
</tr>
<tr>
<td>setRootName(root[, rename])</td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td>showConditionalRestrictions(**kwargs)</td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td>steppar(pars, ranges[, extractFctVal, quiet])</td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td>thaw(specifiers)</td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td>thawLineStrengths([wlmin, wlmax])</td>
<td>Thaw line strengths.</td>
</tr>
<tr>
<td>thawLineWidths([wlmin, wlmax])</td>
<td>Thaw line widths.</td>
</tr>
<tr>
<td>untie(parName[, forceFree])</td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td>updateModel()</td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>

### evaluate (x)

Calculates the model for current parameters.

The “model” is calculated on a wavelength axis with binning specified by the `modelBinsize` parameter.

The line positions are Doppler shifted and the resulting model is rotationally broadened. Finally, the entire model is multiplied by the `scale` parameter to account for a global normalization.

**Parameters**

- x [array] The wavelengths at which to calculate the model.

**Returns**

- model [array] The model evaluated at the specified positions.
**numberOfLines()**

Get the number of lines in the model.

**Returns**

**Number of lines** [int] Number of Gaussian in the model.

**thawLineStrengths (wlmin=None, wlmax=None)**

Thaw line strengths.

Thaws parameters of the from A{n}, where n is the number of the Gaussian component. By default all such parameters will be thawed. The selection may, however, be influenced by specifying `wlmin` and `wlmax`.

**Parameters**

- **wlmin** [float, optional] If specified, only the strength of lines at wavelengths larger than this limits will be thawed.
- **wlmax** [float, optional] If specified, only the strength of lines at wavelengths below this limit will be thawed.

**Returns**

**Thawed parameters** [list] A list of thawed parameter names.

**thawLineWidths (wlmin=None, wlmax=None)**

Thaw line widths.

Thaws parameters of the from sig{n}, where n is the number of the Gaussian component. By default all such parameters will be thawed. The selection may, however, be influenced by specifying `wlmin` and `wlmax`.

**Parameters**

- **wlmin** [float, optional] If specified, only the strength of lines at wavelengths larger than this limits will be thawed.
- **wlmax** [float, optional] If specified, only the strength of lines at wavelengths below this limit will be thawed.

**Returns**

**Thawed parameters** [list] A list of thawed parameter names.

**Example: Evaluation and fitting**

```py
from PyAstronomy import modelSuite as ms
import numpy as np
import matplotlib.pylab as plt

# Create our line list with 4 line
lineList = np.zeros((4, 3))
# Assign wavelengths (in A)
lineList[0, 0] = 5002.37
lineList[1, 0] = 5005.9
lineList[2, 0] = 5007.52
lineList[3, 0] = 5007.64
# Assign EWs (in A)
lineList[0, 1] = 0.01
lineList[1, 1] = 0.05
```

(continues on next page)
lineList[2, 1] = 0.009
lineList[3, 1] = 0.12
# Assign depths (0-1)
lineList[0, 2] = 0.97
lineList[1, 2] = 0.9
lineList[2, 2] = 0.99
lineList[3, 2] = 0.35

wvl = np.arange(5000., 5010., 0.01)

# Get an instance of the LLGauss class
llg = ms.LLGauss(lineList)
# Have a look at the model parameters
llg.parameterSummary()
# Evaluate the model
ml = llg.evaluate(wvl)
# Now apply rotational broadening [km/s]
# with limb-darkening of 0.6
llg["vsini"] = 61.0
llg["eps"] = 0.6
# and evaluate again
mvsini = llg.evaluate(wvl)
# Next, apply a Doppler shift [km/s]
llg["vrad"] = -32.7
# and evaluate
mvrad = llg.evaluate(wvl)

# Plot the results
plt.subplot(2, 1, 1)
plt.plot(wvl, m1, 'b.-')
plt.plot(wvl, mvsini, 'g.-')
plt.plot(wvl, mvrad, 'y.-')

# Now use the model for fitting
# We need "data" ...
data = llg.evaluate(wvl)
# ... with noise
data += np.random.normal(0.0, 0.01, len(data))
# Lets modify the strengths of the Gaussians
# and get it back.
for i in range(llg.numberOfLines()):
    llg["A"+str(i+1)] += np.random.normal(0.0, 0.1)
# Use all line strengths for fitting
llg.thawLineStrengths()
# and fit
llg.fit(wvl, data)
# Plot the result
plt.subplot(2, 1, 2)
plt.errorbar(wvl, data, yerr=np.ones(len(wvl))*0.01, fmt='bp')
plt.plot(wvl, llg.evaluate(wvl), 'r--')
plt.show()
This class provides a convenient parameterization of the Voigt profile, as it is frequently used in astronomy. In particular, the line is parameterized in terms of wavelength, Doppler parameter, damping width, and oscillator strength.

The velocity dispersion is the standard deviation of the velocity distribution. For zero damping width (gamma), the resulting model line is a Gaussian with a standard deviation of b/sqrt(2) in velocity units.

Instrumental resolution can be applied via the parameter R. The instrumental profile is assumed to be a Gaussian with FWHM of w0/R. The additional broadening is implemented by using an internal, effective Doppler parameter. The case R=0 corresponds to infinite instrumental resolution (i.e., no additional broadening).

**Fit parameters:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>w0</td>
<td>Wavelength of the transition</td>
</tr>
<tr>
<td>b</td>
<td>Doppler parameter (corresponds to sqrt(2) times the velocity dispersion).</td>
</tr>
<tr>
<td>gamma</td>
<td>Damping width (full width at half maximum of the Lorentzian)</td>
</tr>
<tr>
<td>f</td>
<td>Oscillator strength (unitless)</td>
</tr>
<tr>
<td>R</td>
<td>Instrumental resolution</td>
</tr>
</tbody>
</table>

**Methods**

- `FWHM()` Estimate FWHM
- `MCMCAutoParameters(ranges[, picky, ...])` Convenience function to generate parameters for MCMC fit.
- `addConditionalRestriction(*args)` Define a conditional restriction.
- `assignValue(specval)` Assign new values to variables.
- `assignValues(specval)` Assign new values to variables.
- `autoFitMCMC(x, y, ranges[, picky, stepsize, ...])` Convenience function to using auto-generated sampling parameters in MCMC.
- `availableParameters()` Provides a list of existing parameters.
- `b1()` Doppler width in cm
- `delRestriction(parName)` Delete restriction
- `description([parenthesis])` Returns a description of the model based on the names of the individual components.
- `errorConfInterval(par[, dstat, statTol, ...])` Calculate confidence interval for a parameter.
- `evaluate(x)` Evaluate the absorption-line profile.
- `fit(x, y[, yerr, X0, minAlgo, mAA, ...])` Carries out a fit.
- `fitEMCEE([x, y, yerr, nwalker, priors, ...])` MCMC sampling using emcee package.
- `fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])` Carry out MCMC fit/error estimation.
- `freeParamNames()` Get the names of the free parameters.
- `freeParameters()` Get names and values of free parameters.
- `freeze(specifiers)` Consider variables free to float.
- `frozenParameters()` Get names and values of frozen parameters.
- `getRelationsOf(specifier)` Return relations of a variable.
- `getRestrictions()` Get all restrictions.
- `hasVariable(specifier)` Determine whether the variable exists.
- `numberOfFreeParams()` Get number of free parameters.
- `parameterSummary([toScreen, prefix, sorting])` Writes a summary of the parameters in text form.
- `parameters()` Obtain parameter names and values.
- `relate(dependentVar, independentVars[, func])` Define a relation.
- `removeConditionalRestriction(*args)` Remove an existing conditional constraint.

Continued on next page...
renameVariable(oldName, newName)  Change name of variable.
restoreState(resource)  Restores parameter values from file or dictionary.
saveState(*args, **kwargs)  Save the state of the fitting object.
setObjectiveFunction(miniFunc)  Define the objective function.
setPenaltyFactor(penalFac)  Change the penalty factor.
setRestriction(restricts)  Define restrictions.
setRootName(root[, rename])  Define the root name of the model.
showConditionalRestrictions(**kwargs)  Show conditional restrictions.
steppar(pars, ranges[, extractFctVal, quiet])  Allows to step a parameter through a specified range.
thaw(specifiers)  Consider variables fixed.
untie(parName[, forceFree])  Remove all relations of parameter parName, i.e., the parameter is not dependent on other parameters.
updateModel()  Recalculate the model using current settings.

FWHM()  
Estimate FWHM
Applies same approximation is Voigt1d

Returns

FWHM  [float] FWHM of line profile in wavelength units [Å]

bl()  
Doppler width in cm

evaluate(x)  
Evaluate the absorption-line profile.

Parameters

x  [array of floats] Contains the wavelength values in Angstrom.

Returns

Model  [array of floats] Return the cross-section in cm².

Example

```python
from PyAstronomy import modelSuite as ms
import numpy as np
import matplotlib.pylab as plt

# Obtain an object of type VoigtAstroP ...
v = ms.VoigtAstroP()
# ... and set some parameters
v["b"] = 87.7
v["f"] = 0.5
v["w0"] = 1214.0
# Damping constant [cm]
v["gamma"] = 2e-9

# Generate wavelength axis ...
wvl = np.linspace(1212., 1216., 200)
# ... and evaluate model
m = v.evaluate(wvl)
```

(continues on next page)
# Plot result
plt.plot(wvl, m, 'b.-')
plt.show()

Example: Adding instrumental resolution

```python
from PyAstronomy import modelSuite as ms
import numpy as np
import matplotlib.pylab as plt

# Obtain an object of type VoigtAstroP ...
v = ms.VoigtAstroP()
# ... and set some parameters
v['b'] = 40.7
v['f'] = 0.5
v['w0'] = 1214.0
# Damping constant [cm]
v['gamma'] = 2e-9

# Generate wavelength axis ...
wvl = np.linspace(1212., 1216., 200)
# ... and evaluate model
m = v.evaluate(wvl)

# Add (Gaussian) instrumental broadening with resolution 5000
v['R'] = 5000
mr = v.evaluate(wvl)

# Plot result
plt.plot(wvl, m, 'b.-', label="R = inf")
plt.plot(wvl, mr, 'r.-', label="R = 5000")
plt.legend()
plt.show()
```

Hydrogen Lyman-alpha line-profile

```python
class PyAstronomy.modelSuite.LyaTransmission(N=0.0, b=10.0, D_fraction=1.5e-05)
    Lyman alpha transmission profile including Deuterium absorption.

    The transmission is given by
    
    $e^{-\sigma N}$,

    where N is the column density and $\sigma$ is the wavelength-dependent cross-section.

    **Fit Parameters:**

    | Parameter | Description | Unit |
    |-----------|-------------|------|
    | N         | Hydrogen column density | /cm^2 |
    | b         | Doppler parameter | km/s |
    | Dfrac     | Deuterium fraction | –     |
```

Parameters
**N** [float, optional] Hydrogen column density [/cm^2]. The default is 0.

**b** [float, optional] The Doppler parameter (corresponds to sqrt(2) times the velocity dispersion) to model thermal line width [km/s]. The default is 10 km/s.

**D_fraction** [float, optional] Fractional abundance of Deuterium with respect to Hydrogen. The default is 1.5e-5.

### Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MCMCautoParameters(ranges[, picky, ...])</code></td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td><code>addConditionalRestriction(*args)</code></td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td><code>assignValue(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>assignValues(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</code></td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td><code>availableParameters()</code></td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td><code>delRestriction(parName)</code></td>
<td>Delete restriction.</td>
</tr>
<tr>
<td><code>description([parenthesis])</code></td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td><code>errorConfInterval(par[, dstat, statTol, ...])</code></td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td><code>evaluate(x)</code></td>
<td>Evaluate the transmission profile.</td>
</tr>
<tr>
<td><code>fit(x, y[, yerr, X0, minAlgo, mAA, ...])</code></td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td><code>fitEMCEE([x, y, yerr, nwalker, priors, ...])</code></td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td><code>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</code></td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td><code>freeParamNames()</code></td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td><code>freeParameters()</code></td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td><code>freeze(specifiers)</code></td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td><code>frozenParameters()</code></td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td><code>getRelationsOf(specifier)</code></td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td><code>getRestrictions()</code></td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td><code>hasVariable(specifier)</code></td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td><code>numberOfFreeParams()</code></td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td><code>parameterSummary([toScreen, prefix, sorting])</code></td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td><code>parameters()</code></td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td><code>relate(dependentVar, independentVars[, func])</code></td>
<td>Define a relation.</td>
</tr>
<tr>
<td><code>removeConditionalRestriction(*args)</code></td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td><code>renameVariable(oldName, newName)</code></td>
<td>Change name of variable.</td>
</tr>
<tr>
<td><code>restoreState(resource)</code></td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td><code>saveState(*args, **kwargs)</code></td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td><code>setObjectiveFunction([miniFunc])</code></td>
<td>Define the objective function.</td>
</tr>
<tr>
<td><code>setPenaltyFactor(penalFac)</code></td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td><code>setRestriction(restricts)</code></td>
<td>Define restrictions.</td>
</tr>
<tr>
<td><code>setRootName(root[, rename])</code></td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td><code>showConditionalRestrictions(**kwargs)</code></td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><code>steppar(pars, ranges[, extractFctVal, quiet])</code></td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><code>thaw(specifiers)</code></td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>
evaluate \((x)\)
Evaluate the transmission profile.

**Parameters**

- \(x\) [array of floats] Contains the wavelength values in Angstrom.

**Returns**

- **Model** [array of floats] The line profile.

**Example**

```python
from PyAstronomy import modelSuite as ms
import numpy as np
import matplotlib.pyplot as plt

la = ms.LyaTransmission()
    # Set some parameters
la["N"] = 5e17
la["b"] = 12.2
la["Dfrac"] = 1.9e-5

    # Set up wavelength axis ...
wvl = np.linspace(1214., 1217., 1000)
    # ... and evaluate model
m = la.evaluate(wvl)

    # Plot the result
plt.plot(wvl, m, 'b.-')
plt.show()
```

**Rotational broadening profile**

**class** PyAstronomy.modelSuite.RotBroadProfile
Implements rotational broadening with linear limb-darkening.

**Fit Parameters:**

- \(xmax\) - Maximal extent of the profile
- \(eps\) - Linear limb-darkening coefficient
- \(A\) - Area under the profile (negative for absorption)
- \(off\) - An offset applied to the profile
- \(lin\) - Gradient of a linear term to adjust the ‘continuum’
- \(mu\) - Center of the profile (same units as \(xmax\))
- \(gsig\) - The standard deviation of a Gaussian with which the rotational profile is convoluted, e.g., to model instrumental resolution.

The profile is given by:

\[
G(x) = A \left( c_1 \sqrt{1 - (x/x_{max})^2} + c_2 (1 - (x/x_{max})^2) \right) + off + lin \cdot x
\]

with the constants given by:

\[
c_1 = \frac{2(1 - \epsilon)}{\pi x_{max}(1 - \epsilon/3)} \quad c_2 = \frac{\epsilon}{2x_{max}(1 - \epsilon/3)}
\]
Here, $x$ can either denote a velocity or a wavelength shift. Thus, $x_{\text{max}}$ can be given in km/s or Angstrom depending on the input; see, e.g., “Stellar Photospheres” by D.F. Gray for a derivation.

Note that the profile is normalized, i.e., the area amounts to one. This may, however, depend on the sampling. If the profile is undersampled, errors can become significant.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MCMCautoParameters(ranges[, picky, ...])</code></td>
<td>Convenience function to generate parameters for MCMC fit.</td>
</tr>
<tr>
<td><code>addConditionalRestriction(*args)</code></td>
<td>Define a conditional restriction.</td>
</tr>
<tr>
<td><code>assignValue(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>assignValues(specval)</code></td>
<td>Assign new values to variables.</td>
</tr>
<tr>
<td><code>autoFitMCMC(x, y, ranges[, picky, stepsize, ...])</code></td>
<td>Convenience function to using auto-generated sampling parameters in MCMC.</td>
</tr>
<tr>
<td><code>availableParameters()</code></td>
<td>Provides a list of existing parameters.</td>
</tr>
<tr>
<td><code>delRestriction(parName)</code></td>
<td>Delete restriction</td>
</tr>
<tr>
<td><code>description([parenthesis])</code></td>
<td>Returns a description of the model based on the names of the individual components.</td>
</tr>
<tr>
<td><code>errorConfInterval(par[, dstat, statTol, ...])</code></td>
<td>Calculate confidence interval for a parameter.</td>
</tr>
<tr>
<td><code>evaluate(v)</code></td>
<td>Calculates the rotational broadening profile according to current parameter values.</td>
</tr>
<tr>
<td><code>fit(x, y[, yerr, X0, minAlgo, mA, ...])</code></td>
<td>Carries out a fit.</td>
</tr>
<tr>
<td><code>fitEMCEE([x, y, nwalker, priors, ...])</code></td>
<td>MCMC sampling using emcee package.</td>
</tr>
<tr>
<td><code>fitMCMC(x, y, X0, Lims, Steps[, yerr, ...])</code></td>
<td>Carry out MCMC fit/error estimation.</td>
</tr>
<tr>
<td><code>freeParamNames()</code></td>
<td>Get the names of the free parameters.</td>
</tr>
<tr>
<td><code>freeParameters()</code></td>
<td>Get names and values of free parameters.</td>
</tr>
<tr>
<td><code>freeze(specifiers)</code></td>
<td>Consider variables free to float.</td>
</tr>
<tr>
<td><code>frozenParameters()</code></td>
<td>Get names and values of frozen parameters.</td>
</tr>
<tr>
<td><code>getRelationsOf(specifier)</code></td>
<td>Return relations of a variable.</td>
</tr>
<tr>
<td><code>getRestrictions()</code></td>
<td>Get all restrictions.</td>
</tr>
<tr>
<td><code>hasVariable(specifier)</code></td>
<td>Determine whether the variable exists.</td>
</tr>
<tr>
<td><code>numberOfFreeParams()</code></td>
<td>Get number of free parameters.</td>
</tr>
<tr>
<td><code>parameterSummary([toScreen, prefix, sorting])</code></td>
<td>Writes a summary of the parameters in text form.</td>
</tr>
<tr>
<td><code>parameters()</code></td>
<td>Obtain parameter names and values.</td>
</tr>
<tr>
<td><code>relate(dependentVar, independentVars[, func])</code></td>
<td>Define a relation.</td>
</tr>
<tr>
<td><code>removeConditionalRestriction(*args)</code></td>
<td>Remove an existing conditional constraint.</td>
</tr>
<tr>
<td><code>renameVariable(oldName, newName)</code></td>
<td>Change name of variable.</td>
</tr>
<tr>
<td><code>restoreState(resource)</code></td>
<td>Restores parameter values from file or dictionary.</td>
</tr>
<tr>
<td><code>saveState(*args, **kwargs)</code></td>
<td>Save the state of the fitting object.</td>
</tr>
<tr>
<td><code>setObjectiveFunction([miniFunc])</code></td>
<td>Define the objective function.</td>
</tr>
<tr>
<td><code>setPenaltyFactor(penalFac)</code></td>
<td>Change the penalty factor.</td>
</tr>
<tr>
<td><code>setRestriction(restricts)</code></td>
<td>Define restrictions.</td>
</tr>
<tr>
<td><code>setRootName(root[, rename])</code></td>
<td>Define the root name of the model.</td>
</tr>
<tr>
<td><code>showConditionalRestrictions(**kwargs)</code></td>
<td>Show conditional restrictions.</td>
</tr>
<tr>
<td><code>steppar(pars, ranges[, extractFctVal, quiet])</code></td>
<td>Allows to step a parameter through a specified range.</td>
</tr>
<tr>
<td><code>thaw(specifiers)</code></td>
<td>Consider variables fixed.</td>
</tr>
<tr>
<td><code>untie(parName[, forceFree])</code></td>
<td>Remove all relations of parameter parName, i.e., the parameter is not dependend on other parameters.</td>
</tr>
<tr>
<td><code>updateModel()</code></td>
<td>Recalculate the model using current settings.</td>
</tr>
</tbody>
</table>
evaluate\((v)\)
Calculates the rotational broadening profile according to current parameter values.

Parameters: x : array
Wavelength or velocity.

Example of usage

```python
import numpy as np
import matplotlib.pyplot as plt
from PyAstronomy import modelSuite as ms

# Get an instance of the model ...
x = ms.RotBroadProfile()
# ... and define some starting value
x["xmax"] = 60.0
x["A"] = 1.0
x["eps"] = 0.8
x["off"] = 0.0

# Define a radial velocity axis
vv = np.linspace(-90., 90., 200)

# Construct some "data" and ...
data = x.evaluate(vv)
# ... add noise
data += np.random.normal(0.0, 1e-3, data.size)

# Fit the model using A, xmax, and eps as free parameters ...
x.thaw(["A", "xmax", "eps"])
x.fit(vv, data)
# ... and show the resulting parameter values.
x.parameterSummary()

# Plot the data and the model
plt.plot(vv, data, 'bp')
plt.plot(vv, x.model, 'r--')
plt.show()
```
CHAPTER 3

Spotlights
4.1 PyAstronomy core (pyaC)

The pyaC package (pyaCore) is the place to collect functionality commonly required among PyAstronomy’s subpackages.

Moreover, pyaC’s documentation is the place were general guidelines for PyAstronomy are formulated.

4.1.1 Functionality

PyAstronomy exceptions and warnings

Occasionally, it is necessary to raise an exception or, at least, show a warning. In the ideal case, this should inform the user exactly about what went wrong (if anything) and provide her/him with a solution to the problem.

PyA contains an exception template, which is the base class of all more specialized PyA exceptions. The template defines a set of information, which can be given to support the user to solve the problem. An exception class may also be used to show a warning by handing it to the warn function.

Contents:

What makes a PyA exception? The PyA exception template

The PyA exception template is designed to provide information about several issues, which, if the exception is not caught right away, should help the user or developer to figure out the problem and, hopefully, resolve it.

The questions to be addressed in the exception template are:

- **What happened?**
  
  This point is the only mandatory field. The user should be provided with information about what actually happened.
• Where did it happen?
The location where the exception occurred. For example, the module, class, and/or function.

• Why did it happen?
The reason or potential reasons for the exception to be raised. For example: The latitude may be too large, because you are using degrees instead of radian.

• How can it be solved?
If possible, provide a solution or a suggestion to solve the problem. For example: Set the deg flag to True, if you use degrees.

• Additional information available?
For example: See web page abcde.com for further information on the algorithm.

Additionally, the template takes an argument specifying the type of the error to give a rough classification of the exception.

In many cases, not all of the fields can be addressed. The first point (what happened) is the only mandatory point; all others may be specified optionally.

The template implementation

class PyAstronomy.pyaC.pyaErrors.PyaErrTemplate(what, errType, where=None, why=None, solution=None, addInfo=None, tbfe=None)

The PyA error class template.

Parameters

what [string] What has happened?
errType [string] Description of the type of error.
where [string] Where did it happen?
why [string] Why did it happen?
solution [string or list of strings] How can the problem be solved?
addInfo [string] Whatever additional information may be available.
tbfe [Exception] Saves trace back from a previously raised exception.

Of the parameters, only the first (what) and second (errType) are mandatory; the latter

should be provided by every more specialized exception class.

.. note:: The template should never be raised directly, but specialized derived classes
should be used.

Attributes

args

Methods

addTB(e) Add trace back from another exception.

Continued on next page
Creating a custom exception class

All PyA exception classes should derive from the exception template class (PyErrTemplate). The name of the exception raised is the first point where the user learns something about the nature of the error. Therefore, the classes should be named carefully.

A simple error class may look like this:

```python
class MyNewPyAError(PyaErrTemplate):
    def __init__(self, what, **keys):
        PyaErrTemplate.__init__(self, what, "My own error type", **keys)
```

The keys are then those given to the template, specifying it in more detail. Every exception should contain documentation, which must, at least, address the question: *When to be raised?*

Show a warning

PyA exception classes may also be used to show warnings. To do, call the `warn` function.

```python
PyAstronomy.pyaC.pyaErrors.warn(w)
```

Parameters:

- `w` - *Something which can be converted into a string*. This may especially be a PyA exception, which is treated as a warning here.

Warnings are printed to stdout.

List of PyA exceptions

Below you find a list of all implemented PyA exceptions. If possible, the exceptions are sorted by category.

**PyA value exceptions**

- `exception pyaC.pyaErrors.pyaValErrs.PyAFloatingPointError(what, **keys)`
  - When to be raised?
    - Whenever a floating point error is caught.

- `exception pyaC.pyaErrors.pyaValErrs.PyANameClash(what, **keys)`
  - When to be raised?
    - Whenever an unexpected doubling of names (e.g., key values in a dictionary) is encountered.

- `exception pyaC.pyaErrors.pyaValErrs.PyAValError(what, **keys)`
  - When to be raised?
    - Whenever an unexpected value is encountered.
PyA uncategorized exceptions

exception pyaC.pyaErrors.pyaOtherErrors.PyAAlgorithm Failure (what, **keys)
    When to be raised?
    Whenever an algorithm fails in accomplishing its purpose.

exception pyaC.pyaErrors.pyaOtherErrors.PyADeprecationError (what, **keys)
    When to be raised?
    Whenever a deprecated part of code is called.

exception pyaC.pyaErrors.pyaOtherErrors.PyADownloadError (what, **keys)
    When to be raised?
    A download could not successfully be carried out.

exception pyaC.pyaErrors.pyaOtherErrors.PyAFileError (fn, mode, **keys)
    When to be raised?
    Whenever a file-related error is detected.
    Parameters
    fn [string] Name of the file in question.
    mode [string, {no, ne}]
        • “no”: File could not be opened.
        • “ne”: File does not exist

exception pyaC.pyaErrors.pyaOtherErrors.PyAImportFailure (what, **keys)
    When to be raised?
    When an import fails. This may not be fatal.

exception pyaC.pyaErrors.pyaOtherErrors.PyANetworkError (what, **keys)
    When to be raised?
    This exception should be raised when an action through the network has failed (e.g., a download).

exception pyaC.pyaErrors.pyaOtherErrors.PyAImplemented (what, **keys)
    When to be raised?
    This exception should be raised when the function/method called has not been implemented. Such a situation
    often occurs when a member is to be implemented in a derived class (abstract base class concept in c++).

exception pyaC.pyaErrors.pyaOtherErrors.PyAOrderError (what, **keys)
    When to be raised?
    Whenever operations seem to be carried out in the wrong order.

exception pyaC.pyaErrors.pyaOtherErrors.PyAParameterConflict (what, **keys)
    When to be raised?
    This exception should be raised when conflicting/mutually exclusive parameters are received.

exception pyaC.pyaErrors.pyaOtherErrors.PyARequiredImport (what, **keys)
    When to be raised?
    If a definitely needed package (e.g., numpy) cannot be imported.

exception pyaC.pyaErrors.pyaOtherErrors.PyAUnclassifiedError (what, **keys)
    When to be raised?
    If an error occurred that cannot or shall not be specified further.
PyA permanent — Configuration and Data

The pyaPermanent package is a core module of PyAstronomy. Its purpose is to manage content, which shall be permanently available to PyA, but may be created or added dynamically; examples comprise configuration and newly downloaded data.

What gets stored where?

To store the data, PyA requires a directory (in the following the “data directory”) to which it has write access. Such a directory can be assigned (and created) at the first call to a method, which demands access to it. By default, PyA will suggest to use a subdirectory of your home directory, but this choice can be modified. If an appropriate directory has been assigned to PyA, a single file in your home directory will be created (named “.pyaConfigWhere”), which contains a single line, which points PyA to its data directory; unfortunately, this cannot be avoided, because PyA needs one point, which it can find without any additional information.

In PyA’s data directory, the file “pyaConfig.cfg” holds the “root configuration” for PyA. Other modules may add own content, usually in subdirectories.

How to use it?

Of course, you can directly access the files in PyA’s data path. However, PyA provides some useful classes, which offer some additional convenience and maybe “bug security”. PyA packages will use these to access the data directory:

PyA’s root configuration

class PyAstronomy.pyaC.pyaPermanent.PyAConfig
    Provide access to permanent PyA configuration.

    Permanent configuration and data created by/for PyAstronomy are stored in a data directory appointed to PyAstronomy. The directory name is stored in a file named “.pyaConfigWhere”, which is located in the user’s home directory (defined by environment variable home). This file contains only a single line, which is exactly the name of the PyA data directory.

    In the PyA data directory, this class creates a stub configuration file named “pyaConfig.cfg”, if the file does not already exist.

    Attributes

    dpath [string] The name of PyA’s data directory (defined by the content of the file “.pyaConfigWhere” in the home directory). None if no valid directory could be found or created.

    configWhere [string] The full name of the “.pyaConfigWhere” file.

    Methods

    accessRootConfig() Provides access to PyA’s root configuration file.

    getDataRoot() Access the root path to PyA’s permanent data and configuration.

    remove_option(section, option) Remove option

    remove_section(section) Remove section

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>saveConfigToFile()</code></td>
<td>Save current state of configuration to file ‘pyaConfig.cfg’.</td>
</tr>
<tr>
<td><code>set(section, option, value)</code></td>
<td>Add an entry to the global configuration file.</td>
</tr>
</tbody>
</table>

**accessRootConfig()**

Provides access to PyA’s root configuration file.

**Returns**

*Configuration* [ConfigParser object] The content of the root configuration file “pyaConfig.cfg”.

**getDatosRoot()**

Access the root path to PyA’s permanent data and configuration.

**Returns**

*Path* [string] The root path as defined in the file “.pyaConfigWhere” in the home directory.

**remove_option**(section, option)** remove_section**(section)

Remove option or section.

**saveConfigToFile()**

Save current state of configuration to file ‘pyaConfig.cfg’.

**set**(section, option, value)

Add an entry to the global configuration file.

If the specified section does not already exist, it is created.

**Example session**

When PyAConfig is called without a configured data path, it will ask for a location where PyA is allowed to store its data and configuration. This will look something like this:

```
>>> from PyAstronomy.pyaC import pyaPermanent
>>> pc = pyaPermanent.PyAConfig()
Please provide a directory where PyA can store data (may already exist):
Press enter to accept default.
Path (default = /pathToHome/PyAData): another/Path/I/Like
PyA data path configured successfully. Using path:
another/Path/I/Like
```

**Note:** The data path is not a subdirectory of the path you specify, but exactly the path you specify.

**The PyA File System**

Well, “file system” is not really true. It is only a tool to manage access to PyA’s data directory.
class PyAstronomy.pyaC.pyaPermanent.PyAFS

Manage access to PyA's data directory.

This class provides a convenient interface to create, read, and modify files in PyA's data directory.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>composeFilename(rfn)</td>
<td>Returns full filename.</td>
</tr>
<tr>
<td>createSubfolder(folder)</td>
<td>Create subfolders (relative to data root directory).</td>
</tr>
<tr>
<td>downloadToFile(url, fn[, clobber, verbose, ...])</td>
<td>Download content from URL.</td>
</tr>
<tr>
<td>fileExists(fn)</td>
<td>Check whether file exists</td>
</tr>
<tr>
<td>getFullPath(rpath)</td>
<td>Return full path for relative path</td>
</tr>
<tr>
<td>globglob(rpath, ge)</td>
<td>Apply glob.glob on relative path</td>
</tr>
<tr>
<td>removeFile(fn)</td>
<td>Remove a file from PyA's data directory.</td>
</tr>
<tr>
<td>requestFile(relName[, mode, openMethod])</td>
<td>Obtain a file object within the PyA data path.</td>
</tr>
<tr>
<td>touchFile(relName)</td>
<td>Create an empty file in PyA's data directory.</td>
</tr>
</tbody>
</table>

**composeFilename (rfn)**

Returns full filename.

Parameters

rfn [string] Relative file name (in PyA data path).

Returns

filename [string] Absolute filename.

**createSubfolder (folder)**

Create subfolders (relative to data root directory).

Ignores already existing folders. This method can handle recursive directory creation.

**downloadToFile**

Download content from URL.

Parameters

url [string] The location of the content.

fn [string] The relative or absolute name of the file to which the download shall be saved.

clobber [boolean, optional] If True, an existing file will be overwritten.

verbose [boolean, optional] If True, information on the download will be printed to the screen.

openMethod [callable] The method used to open the file to write to (default is open, other choices may be gzip.open or io.ipen)

context [ssl context] SSL context parameter handed to urlopen.

**fileExists (fn)**

Check whether file exists

Parameters

fn [string] The filename either relative to PyA's data path or absolute.

Returns
Flag [boolean] True if file exists and False otherwise.

**getFullPath** *(rpath)*
Return full path for relative path

**Parameters**

- **rpath** [string or list of strings] Relative path in PyA data directory or individual constituents of path

**Returns**

- **Path** [string] Absolute path

**globglob** *(rpath, ge)*
Apply glob.glob on relative path

**Parameters**

- **rpath** [string] Relative path
- **ge** [string] Glob expression (e.g., .)

**Returns**

- **Fns** [list of strings] Glob results (relative paths)

**requestFile** *(relName, mode='r', openMethod=<built-in function open>, *args, **kwargs)*
Obtain a file object within the PyA data path.

This method opens and creates files within PyA's data path. If the directory in which a file shall be created does not yet exist, it will be created.

**Parameters**

- **relName** [string] The filename. Usually, it will be given relative to the PyA data path, but it can also be an absolute filename.
- **mode** [string, optional] The opening mode (e.g., “r” or “w”). This flag is given to the `openMethod`. The default is “r” for “read”.
- **openMethod** [method, optional] The method used to create the file object. The default is Python’s built-in `open` method. Another example could be `gzip.open`.

**Returns**

- The file object.

**touchFile** *(relName)*
Create an empty file in PyA's data directory.

**Parameters**

- **relName** [string] The filename. Usually, it will be given relative to the PyA data path, but it can also be an absolute filename.
Realizing an update cycle

```python
class PyAstronomy.pyaC.pyaPermanent.PyAUpdateCycle(fn, section, updateCycle=7):
    A simple data update cycle.
    This class provides a simple way to realize an update cycle for data, which need to be re-initialized periodically.
    Parameters
    fn [string] The name of the file (within PyAFS), which is used to store the settings.
    section [string] The section within the config file used to store the settings.
    updateCycle [float or int] The number of days defining the update cycle.
```

Methods

```python
changeDownloadCycle(c)
    Change the time after which the data are updated.
    By default, the data will be updated if they are older than the given update cycle. This method allows you to change that cycle.
    Parameters
    c [float or None] The new update cycle in days. If None is provided, updating is switched off.

dataAge()
    Determine the “age” of the data.
    Returns
    age [float] The time since last data update in days. None, if no age can be determined, e.g., if data have never been downloaded.

needsUpdate()
    Determine whether data need to be updated.
    Returns
    Update flag [boolean] True if data need update and False otherwise.
```

Math tools for PyAstronomy

A collection of mathematical or numerical tools with potential application throughout PyAstronomy.

Compute numerical derivatives

```python
PyAstronomy.pyaC.diffCFD(x, y, derivative, errOrd=2)
    Calculate the numerical derivative using central finite distances.
    Parameters
    x [array] The abscissa values.
```
y [array] The ordinate values.

**derivative** [int] The derivative to calculate. Use, e.g., 1 to obtain the first derivative.

**errOrd** [int] The order of the error term. The default is 2. Other possibilities are 4, 6, and 8.

Note the finite differences may not be available for all combinations.

**Returns**

**indices** [array of int] The indices with respect to the input abscissa to which the derivative applies.

**der** [array] The derivative.

**Example**

```python
from PyAstronomy import pyaC
import matplotlib.pylab as plt
import numpy as np

x = np.linspace(-10, 10, 1000)

# Computer polynomial and its derivatives
# (quasi analytically)
y = [np.poly1d((0.03, -0.31, 0.4, 0.35, 1.4))]
for i in range(4):
    y.append(y[-1].deriv())

# Compute derivates numerically and compare to analytic solution
erro = 2
for i in range(1, 5):
    indi, der = pyaC.diffCFD(x, np.polyval(y[0], x), i, erro)
    plt.plot(x[indi], np.polyval(y[i], x[indi]), 'b.')
    plt.plot(x[indi], der, 'r--')
plt.show()
```

**Numerical integration**

**Trapezoid rule with interpolated boundaries**

**PyAstronomy**.**pyaC.ibtrapz**(x, y, x0, x1, iaout=False)

Use the trapezoid rule to integrate and interpolate boundary values.

Can be used for integration on tabled data, where the integration boundaries are not located on the grid. The values to use at the boundaries are determined using linear interpolation.

**Parameters**

**x,y** [arrays] The data.

**x0, x1** [float] The integration boundaries.

**iaout** [boolean, optional] If True, the arrays used internally for integration are also returned. The default is False.

**Returns**

**Integral** [float] The value of the resulting integral.
**xi, yi** [arrays, optional] Internally used arrays for integration including the values derived at the boundaries. Only returned if `iaout` is set to True.

**Example**

```python
from __future__ import print_function
from PyAstronomy.pyaC import mtools
import numpy as np

x = np.arange(-2., 2.01, 0.1)
y = x**3 + 1.7
x0 = -1.375
x1 = +1.943

# Analytical value of integral
analyt = 0.25*(x1**4 - x0**4) + 1.7*(x1-x0)

print("Analytical value: ", analyt)
print("ibtrapz: ", mtools.ibtrapz(x, y, x0, x1))
```

**Find zero crossings in discrete data set**

**PyAstronomy.pyaC.zerocross1d**(x, y, getIndices=False)

Find the zero crossing points in 1d data.

Find the zero crossing events in a discrete data set. Linear interpolation is used to determine the actual locations of the zero crossing between two data points showing a change in sign. Data points which are zero are counted in as zero crossings if a sign change occurs across them. Note that the first and last data point will not be considered whether or not they are zero.

**Parameters**

- **x, y** [arrays] Ordinate and abscissa data values.
- **getIndices** [boolean, optional] If True, also the indicies of the points preceding the zero crossing event will be returned. Default is False.

**Returns**

- **xvals** [array] The locations of the zero crossing events determined by linear interpolation on the data.
- **indices** [array, optional] The indices of the points preceding the zero crossing events. Only returned if `getIndices` is set True.

**Example**

```python
import numpy as np
import matplotlib.pylab as plt
from PyAstronomy import pyaC

# Generate some 'data'
x = np.arange(100.)**2
y = np.sin(x)
```

(continues on next page)
# Set the last data point to zero.
# It will not be counted as a zero crossing!
y[-1] = 0

# Set point to zero. This will be counted as a zero crossing
y[10] = 0.0

# Get coordinates and indices of zero crossings
xc, xi = pyaC.zerocross1d(x, y, getIndices=True)

# Plot the data
plt.plot(x, y, 'b.-')
# Add black points where the zero line is crossed
plt.plot(xc, np.zeros(len(xc)), 'kp')
# Add green points at data points preceding an actual zero crossing.
plt.plot(x[xi], y[xi], 'gp')
plt.show()

Conversions and calculations

Convert degree into rad and vice versa

PyAstronomy.pyaC.degtorad(d)
Convert degrees into radians.

Parameters

d [float or array] Angle in degrees.

Returns

Angle [float or array] The angle converted into radians.

PyAstronomy.pyaC.radtodeg(r)
Convert radians into degrees.

Parameters

d [float or array] Angle in radians.

Returns

Angle [float or array] The angle converted into degrees.

Compute ratio of factorials

PyAstronomy.pyaC.farat(x, y)
Compute ratio of factorials.

Computes x!/y! via \( \ln(x!) - \ln(y!) \) to avoid numerical overflow.

Parameters

x [int, float] The factorial of x is the numerator
y [int, float] The factorial of y is the denominator
Returns

**Ratio**  [float] The ratio $x!/y!$ (not the logarithm).

**Needful things**

A collection of helpers, which have no place elsewhere.

**Nested loop iterator**

```python
class PyAstronomy.pyaC.NestedLoop(limits, lowerLimits=None)
```

Implements an iteration over a nested loop.

Iterates over nested loops. First, increases the first counter, then increments the second and so on.

**Parameters**

- **limits**  [list of int] The upper limits for the loops.
- **lowerLimits**  [list of int, optional] The lower limits of the loops. The default is zero.

**Example**

```python
from __future__ import print_function
from PyAstronomy import pyaC

nl = pyaC.NestedLoop([4,2,5], lowerLimits=[0,0,-5])

for indices in nl:
    print(indices)
```

**Invert index selection**

```python
PyAstronomy.pyaC.invertIndexSelection(a, indi)
```

Invert index selection in one-dimensional array.

Say, e.g., a numpy.where operation produced an array of indices (`indi`), which you needed for one reason, but now, for another reason, you need all elements of the array, which were not selected by the operation. This is the situation handled by this function.

**Parameters**

- **a**  [int or array] Either the length of the array to which `indi` refers to or the array (one-dimensional) itself.
- **indi**  [array] An array of indices selected from `a`.

**Returns**

**Inverse selection**  [array] An array containing the indices of all array elements not referenced by `indi`. 
Example

```python
from __future__ import print_function
import numpy as np
from PyAstronomy import pyc as pc

# Create "data" values and select some
x = np.exp(np.arange(20.0)/20.0)
indi = np.where(np.logical_and(x > 1.4, x < 1.7))

print("Selected indices and values:")
print(" indices: ", indi)
print(" values : ", x[indi])

indiInv = pc.invertIndexSelection(x, indi)

print()
print("Inverted selection:")
print(" indices: ", indiInv)
print(" values : ", x[indiInv])

# Check that the same result is obtained by simply
# passing the length of the array 'x'
indiInvLen = pc.invertIndexSelection(len(x), indi)

print()
print("Are indiInv and indiInvLen are the same? ")
print(" ", np.all(indiInvLen == indiInv))
```

Fuzzy word matching

**PyAstronomy.pyc.fuzzyMatch**(inkey, wordList, caseSensitive=True, n=3, cutoff=0.6, raises=False)

Find exact and approximate matches for input in word list.

Uses `get_close_matches` from Python’s difflib to search for the best matches between the input keyword and a list of words.

**Parameters**

- **inkey** [string] The input keyword.
- **wordList** [list of strings] List of words with possible matches for inkey.
- **caseSensitive** [boolean, optional] If True (default), the search will be case sensitive.
- **n** [int, optional] Number of potential matches returned by `get_close_matches`.
- **cutoff** [float, optional] Number between 0 and 1 indicating the degree of similarity between inkey and the entries from wordList. The lower the number, the more dissimilar the potential matches may be. (cutoff parameter from `get_close_matches`).
- **raises** [boolean, optional] If True, a `PyAValError` giving a summary of the failure will be raised if no exact match is found. The default is false.

**Returns**

- **Matches** [dictionary] If found, contains the exact match (key “em”) found in the list (in lower case if caseSensitive is True) and a list of close matches (key “cm”), which the
user may have meant to specify. If an exact match is found, also its index in wordList is provided.

Example

```python
from __future__ import print_function
from PyAstronomy import pyaC

wordList = ['one', 'two', 'three', 'four', 'o-ne']

r = pyaC.fuzzyMatch('One', wordList)
print("Exact match: \{em:\}, close match(es): \{cm:\}".format(**r))

r = pyaC.fuzzyMatch('One', wordList, cutoff=0.4)
print("Exact match: \{em:\}, close match(es): \{cm:\}".format(**r))

r = pyaC.fuzzyMatch('One', wordList, caseSensitive=False)
print("Exact match: \{em:\}, close match(es): \{cm:\}".format(**r))
```

Matrix (2d) output

PyAstronomy.pyaC.matrix2doutput (m, oformat=’% 6.2e’, colsep=’ | ’, rowNames=None, colNames=None, transpose=False, toScreen=True)

Format a matrix in readable form and write it to screen.

The column is specified by the second index, e.g., the first entry in the second column is given by m[0,1]. The first entry in the third row is, consequently, given by m[2,0].

**Parameters**

- **m** [2-dimensional array] The data to be formatted.
- **oformat** [string or list of strings, optional] The output format. If string, the same format string will be used for all columns. If a list of strings is given, the associated specifier will be used for each individual column.
- **colsep** [string, optional] The separator used between columns.
- **rowNames** [list of strings, optional] The names of the rows.
- **colNames** [list of strings optional] The names of the columns.
- **transpose** [boolean, optional] If True, the input matrix will be transposed. In effect, this exchanges the roles of columns and rows. Note, however, that the role of colNames and rowNames remains unaltered. The default is False.
- **toScreen** [boolean, optional] If True (default), the result will be written to screen.

**Returns**

- **Formatted matrix** [list of strings] The formatted output is a list of strings, which might be written to screen.

Example
from __future__ import print_function
from PyAstronomy import pyaC as PC
import numpy as np

m = np.zeros((4, 3))
colNames = ["first", "second", "third"]
rowNames = ["1", "2", "3", "4"]

for j in range(4):
    for i in range(3):
        m[j, i] = (i+1) * 10**(j+1) * np.sin(i*j)

PC.matrix2doutput(m, colNames=colNames, rowNames=rowNames)
print()
PC.matrix2doutput(m, rowNames=rowNames)
print()
PC.matrix2doutput(m, colsep=" ")
print()
PC.matrix2doutput(m, oformat="% 12.5f", colNames=colNames, rowNames=rowNames)
print()
PC.matrix2doutput(m, oformat=["% 12.5f", "% 6.3f", "% e"], colNames=colNames)

Simple Input-Output file

class PyAstronomy.pyaC.SimIOF(origin, *args, **kwargs)
    Simple Input/Output file.

    If a file is opened for writing, the origin and date will be added at the top of the file. If it is opened for reading, the
    properties will be read, converted into float of possible, and stored in the args attribute. If the float-conversion
    fails, the value is kept as a string.

    Parameters

    origin [str] Identifies the script/program writing the file.
    args [tuple] Passed to the constructor of a file object.
    kwargs [dictionary] Passed to the constructor of a file object.

    Methods

    addColInfo(cns[, oneLine]) Add enumerated column names to file.
    addProp(name, value, fmt=None) Add a property to the file.
    close() Close file.
    write(*args, **kwargs) Write to file object

    addColInfo (cns, oneLine=True)
        Add enumerated column names to file.

        Parameters

        cns [list of strings] List of names
        oneLine [boolean, optional] If True, all definitions will be written into a single line.
        Otherwise each definition is written on its own line.

    addProp (name, value, fmt=None)
Add a property to the file.

**Parameters**

- **name** [string or list of strings] Name of the property
- **value** [any, or list of any type] Value of that property
- **fmt** [string, list, optional] The format-string used to represent the property. If a single string is given, it is applied to all properties. Otherwise, one format string for every property should be given.

```python
from __future__ import print_function
from PyAstronomy import pyaC as PC
import numpy as np

f = PC.SimIOF("origin", "test.tmp", 'w')

a = 5.6
b = 8.7
c = 5

f.addProp("a", a, fmt="% 4.4e")
f.addProp(["b", "c"], [b, c])

for x in range(10):
    f.write(str(x) + "\n")

f.close()

g = PC.SimIOF(\"origin\", \"test.tmp\", \'r\')

# See the properties assigned
print(g.args)
# Use loadtxt to load the data
print(np.loadtxt(g))
```

### 4.1.2 Rules and guidelines

**PyAstronomy versioning rules**

In the following, the version tag of PyAstronomy is described. The version tag consists of three numbers and a possible extension, specifying the status (beta) of the version.

The entire PyA package is characterized by a single version number.

As an example, such a version number could look like:

- 0.1.2
- 0.1.2-beta.

The version tag, thus, consists of four parts being:

- **First number**: An integer specifying the major release number.
• Second number: An integer specifying the minor release number. A change in this number indicates that an API change can have occurred (but not necessarily).

• Third number: An integer specifying the fix release number. A change in this number indicates that no changes to the public API, possibly breaking running code, have been introduced; though, the internal working of one or more submodules may have changed.

• -beta: Indicates that a version is currently developed on the basis of the stated version. No guarantee for nothing.

The PyA guide for porting IDL code

The “Interactive Data Language” (IDL) has been an influential tool in the astronomical community. Some essential code is written in IDL, and sometimes it may be desirable to port it to Python.

Some typical aspects of IDL cannot be directly translated into Python constructs. Therefore, we provide a number of guidelines here, which can be used in porting IDL code to Python:

• Case sensitivity IDL is case-insensitive. We recommend to use lower case for all function and variable names.

• IDL keywords IDL functions may take keyword arguments. If the purpose of the keyword is only to be present or not, we recommend to use a boolean keyword that defaults to False in Python. If the keyword argument is supposed to convey information to the function, we recommend to use a Python keyword that defaults to the standard value used (in IDL) when when the keyword is not given.

• Arrays Python provides no built-in array support. IDL arrays are represented by numpy arrays. Were IDL uses an array as return value, it may not be good practice to do the same in Python. In this case, it remains up to the developer to choose a reasonable solution and document it.

• Functions supporting both single float and array arguments We regard array support in Python as optional. In most cases performance is not an issue for the functions under consideration.
CHAPTER 5

Indices and tables

- genindex
- modindex
- search
Bibliography


[vdK] van der Klis, Fourier Techniques In X-Ray Timing


p
pyaC.pyaErrors.pyaOtherErrors, 356
pyaC.pyaErrors.pyaValErrs, 355
PyAstronomy.pyasl.asl.astroTimeLegacy, 102
Symbols

_ModelRebinDocu (class in PyAstronomy.funcFit), 295
_cheby () (PyAstronomy.pysl.SpecTypeDeJager method), 36
_checkBand() (PyAstronomy.pysl.Ramirez2005 method), 30
_checkJP() (PyAstronomy.pysl.BSBase method), 54
_checkN() (PyAstronomy.pysl.BSBase method), 54
_checkST() (PyAstronomy.pysl.Ramirez2005 method), 30
_ciddor() (PyAstronomy.pysl.RefractiveIndexAV method), 74
_convertBandName() (PyAstronomy.pysl.Ramirez2005 method), 30
_decomposeSPT() (PyAstronomy.pysl.SpecTypeDeJager method), 36
_edlen53() (PyAstronomy.pysl.RefractiveIndexAV method), 74
_extractTableData() (PyAstronomy.pysl.Ramirez2005 method), 30
_indexSpecType() (PyAstronomy.pysl.SpecTypeDeJager method), 36
_initSimplex() (PyAstronomy.funcFit.NelderMead method), 269
_peckReeder() (PyAstronomy.pysl.RefractiveIndexAV method), 74
_readData() (PyAstronomy.pysl.Ramirez2005 method), 30
_readTab23() (PyAstronomy.pysl.Ramirez2005 method), 30
_readTab45() (PyAstronomy.pysl.Ramirez2005 method), 31
_resolveMetallicityIndex() (PyAstronomy.pysl.Ramirez2005 method), 31
_resolve_b() (PyAstronomy.pysl.SpecTypeDeJager method), 36
_resolve_s() (PyAstronomy.pysl.SpecTypeDeJager method), 37
_step() (PyAstronomy.funcFit.NelderMead method), 269
_stopNM1965() (PyAstronomy.funcFit.NelderMead method), 269

A

absMagToPower() (in module PyAstronomy.pysl), 18
absModuleToDist() (in module PyAstronomy.pysl), 18
abundance() (PyAstronomy.pysl.AbundancePatterns method), 25
AbundancePatterns (class in PyAstronomy.pysl), 24
accessRootConfig() (PyAstronomy.pyaC.pyaPermanent.PyAConfig method), 358
addColInfo() (PyAstronomy.pyaC.SimIOF method), 368
addComponent() (PyAstronomy.funcFit.SyncFitContainer method), 297
addConditionalRestriction() (PyAstronomy.funcFit.OneDFit method), 280
addConditionalRestriction() (PyAstronomy.funcFit.Params method), 272
addKeplerPassband() (PyAstronomy.pysl.TransmissionCurves method), 99
addPassband() (PyAstronomy.pysl.TransmissionCurves method), 99
addProp() (PyAstronomy.pyaC.SimIOF method), 368
addSpitzerIRACPassbands() (PyAstronomy.pysl.TransmissionCurves method), 100
addTESSPassband() (PyAstronomy.pysl.TransmissionCurves method), 100
airmassPP() (in module PyAstronomy.pysl), 140
airmassSpherical() (in module PyAstronomy.pyasl), 141
airtovac() (in module PyAstronomy.pyasl), 76
airtovac() (PyAstronomy.pyasl.RefractiveIndexAV method), 74
airtovac2() (in module PyAstronomy.pyasl), 72
aitoff() (in module PyAstronomy.pyasl), 115
applyConditionalRestrictions() (PyAstronomy.funcFit.Params method), 272
applyProperMotion() (PyAstronomy.pyasl.RefractiveIndexAV method), 115
assignValue() (in module PyAstronomy.pyasl), 144
assignValue() (PyAstronomy.funcFit.OneDFit method), 280
assignValues() (PyAstronomy.funcFit.Params method), 273
AtomicNo (class in PyAstronomy.pyasl), 23
autoFitMC(MC) (PyAstronomy.funcFit.OneDFit method), 280
availableBands() (PyAstronomy.pyasl.TransmissionCurves method), 100
availableBands() (PyAstronomy.pyasl.ExoplanetEU method), 172
availableColumns() (PyAstronomy.pyasl.ExoplanetEU method), 172
availableColumns() (PyAstronomy.pyasl.ExoplanetsOrg method), 174
availableColumns() (PyAstronomy.pyasl.MMSETSTable method), 182
availableColumns() (PyAstronomy.pyasl.nasaExoplanetArchive method), 165
availableGrads() (PyAstronomy.pyasl.KuruczModels method), 179
availableLoggs() (PyAstronomy.pyasl.KuruczMT method), 180
availableParameters() (PyAstronomy.funcFit.OneDFit method), 281
availableParameters() (PyAstronomy.funcFit.Params method), 273
availableParameters() (PyAstronomy.funcFit.TraceAnalysis method), 301
availablePatterns() (PyAstronomy.pyasl.AbundancePatterns method), 25
availableTeffs() (PyAstronomy.pyasl.KuruczMT method), 180
availableTraces() (PyAstronomy.funcFit.TraceAnalysis method), 301

B
BallesterosBV_T (class in PyAstronomy.pyasl), 33
Baraffe98Tracks (class in PyAstronomy.pyasl), 162

C
CauchyLorentzld (class in PyAstronomy.funcFit), 251
changeDownloadCycle() (PyAstronomy.pyaC.pyaPermanent.PyUpdateCycle method), 361
changeDownloadCycle() (PyAstronomy.pyaC.pyaPermanent.PyUpdateCycle method), 361
changeDownloadCycle() (PyAstronomy.pyasl.ExoplanetEU method), 172
changeDownloadCycle() (PyAstronomy.pyasl.ExoplanetEU2 method), 169
changeDownloadCycle() (PyAstronomy.pyasl.ExoplanetsOrg method), 174
changeDownloadCycle() (PyAstronomy.pyasl.nasaExoplanetArchive method), 165
chromoAgeRHK() (in module PyAstronomy.pyasl), 34
Circle2d (class in PyAstronomy.funcFit), 264
close() (PyAstronomy.pyaC.SimIOF method), 269
c_o_aberration() (in module PyAstronomy.pyasl), 122
c_o_nutation() (in module PyAstronomy.pyasl), 120
c_o_refract() (in module PyAstronomy.pyasl), 125
c_o_refract_forward() (in module PyAstronomy.pyasl), 124
colorToTeff() (PyAstronomy.pyasl.Ramirez2005 method), 31
colorToTeff_nop() (PyAstronomy.pyasl.Ramirez2005 method), 31
components() (PyAstronomy.funcFit.SyncFitContainer method), 297
composeFilename() (PyAstronomy.pyaC.pyaPermanent.PyAFS method), 359
composeVariableName() (PyAstronomy.funcFit.ModelNameIdentBase method),
ConstantFit1d (class in PyAstronomy.funcFit), 260
cr0vertDampingConstant () (in module PyAstronomy.pyasl), 94
cr0vertSpecif!er() (PyAstronomy.funcFit.ModelNameIdentBase method), 290
cr0nverveWith() (PyAstronomy.pyasl.TransmissionCurves method), 100
cordinateGrid() (in module PyAstronomy.funcFit), 306
cordsDegToSexa() (in module PyAstronomy.pyasl), 143
cordsSexaToDeg() (in module PyAstronomy.pyasl), 143
correlationMatrix() (PyAstronomy.funcFit.TraceAnalysis method), 301
correlationTable() (PyAstronomy.funcFit.TraceAnalysis method), 301
cr3ateSubfolder() (PyAstronomy.pyaC.pyaPermanent.PyAFS method), 359
crosscorrRV() (in module PyAstronomy.pyasl), 83
dataAge() (PyAstronomy.pyaC.pyaPermanent.PyAUpdateCycle method), 361
dataAge() (PyAstronomy.pyasl.ExoplanetEU method), 172
dataAge() (PyAstronomy.pyasl.ExoplanetEU2 method), 170
dataAge() (PyAstronomy.pyasl.ExoplanetsOrg method), 174
dataAge() (PyAstronomy.pyasl.NasaExoplanetArchive method), 165
daycnv() (in module PyAstronomy.pyasl.asl.astroTimeLegacy), 104
ddx_rochepot_d1() (in module PyAstronomy.pyasl), 40
ddy_rochepot_d1() (in module PyAstronomy.pyasl), 40
ddz_rochepot_d1() (in module PyAstronomy.pyasl), 40
decimalYear() (in module PyAstronomy.pyasl), 114
decimalYearGregorianDate() (in module PyAstronomy.pyasl), 114
decompose() (PyAstronomy.pyasl.SVD method), 68
decomposeVariableName() (PyAstronomy.funcFit.ModelNameIdentBase method), 291
degToDMS() (in module PyAstronomy.pyasl), 144
degToHMS() (in module PyAstronomy.pyasl), 143
degotrad() (in module PyAstronomy.pyaC), 364
delRestriction() (PyAstronomy.funcFit.OneDFit method), 281
delRestriction() (PyAstronomy.funcFit.Params method), 273
description() (PyAstronomy.funcFit.OneDFit method), 281
diffCFD() (in module PyAstronomy.pyaC), 361
dmsToDeg() (in module PyAstronomy.pyasl), 144
dopplerShift() (in module PyAstronomy.pyasl), 85
downloadData() (PyAstronomy.pyasl.ExoplanetsOrg method), 175
downloadData() (PyAstronomy.pyasl.NasaExoplanetArchive method), 165
downloadData() (PyAstronomy.pyasl.SWEETCat method), 177
downloadToFile() (PyAstronomy.pyaC.pyaPermanent.PyAFS method), 359
eq2hor() (in module PyAstronomy.pyasl), 130
errorConfInterval() (PyAstronomy.funcFit.OneDFit method), 282
estimateStdMAD() (PyAstronomy.pyasl.BSBase method), 54
evalComponent() (PyAstronomy.funcFit.MultiVoigt1d method), 255
evaluate() (PyAstronomy.funcFit._ModelRebinDocu method), 295
evaluate() (PyAstronomy.funcFit.MultiVoigt1d method), 255
evaluate() (PyAstronomy.funcFit.SyncFitContainer method), 297
evaluate() (PyAstronomy.modelSuite.KeplerEllipseModel method), 336
evaluate() (PyAstronomy.modelSuite.LLGauss method), 341
evaluate() (PyAstronomy.modelSuite.LyaTransmission method), 348
evaluate() (PyAstronomy.modelSuite.radVel.SinRadVel method), 332
evaluate() (PyAstronomy.modelSuite.RmcL method), 326
evaluate() (PyAstronomy.modelSuite.RmcLell method), 328
evaluate() (PyAstronomy.modelSuite.RotBroadProfile method), 350

Index 379
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluate()</td>
<td>PyAstronomy.modelSuite.VoigtAstroP method, 345</td>
</tr>
<tr>
<td>evaluate()</td>
<td>PyAstronomy.modelSuite.XTran.limBrightTrans.LimBrightTrans method, 323</td>
</tr>
<tr>
<td>evaluate()</td>
<td>PyAstronomy.modelSuite.XTran.palTrans.PalLC method, 308</td>
</tr>
<tr>
<td>evaluate()</td>
<td>PyAstronomy.modelSuite.XTran.palTrans.PalLC* method, 311</td>
</tr>
<tr>
<td>ExoplanetEU</td>
<td>(class in PyAstronomy.pyasl), 171</td>
</tr>
<tr>
<td>ExoplanetEU2</td>
<td>(class in PyAstronomy.pyasl), 173</td>
</tr>
<tr>
<td>expCorrRN()</td>
<td>(in module PyAstronomy.pyasl), 161</td>
</tr>
<tr>
<td>ExpDecayFit1d</td>
<td>(class in PyAstronomy.funcFit), 257</td>
</tr>
<tr>
<td>FAP()</td>
<td>(PyAstronomy.pyTiming.pyPeriod.Fourier method), 199</td>
</tr>
<tr>
<td>FAP()</td>
<td>(PyAstronomy.pyTiming.pyPeriod.Gls method), 194</td>
</tr>
<tr>
<td>FAP()</td>
<td>(PyAstronomy.pyTiming.pyPeriod.LombScargle method), 201</td>
</tr>
<tr>
<td>farat()</td>
<td>(in module PyAstronomy.pyasl), 198</td>
</tr>
<tr>
<td>fastRotBroad()</td>
<td>(in module PyAstronomy.pyasl), 364</td>
</tr>
<tr>
<td>FHK()</td>
<td>(PyAstronomy.pyasl.SMW_RHK method), 21</td>
</tr>
<tr>
<td>fileExists()</td>
<td>(PyAstronomy.pyaslp.yaC.pyaPermanent.PyAFS method), 359</td>
</tr>
<tr>
<td>findModels()</td>
<td>(PyAstronomy.pyaslp.BSEqSamp method), 58</td>
</tr>
<tr>
<td>FirstIonizationPot</td>
<td>(class in PyAstronomy.pyasl), 181</td>
</tr>
<tr>
<td>fit()</td>
<td>(PyAstronomy.funcFit.FuFNM method), 268</td>
</tr>
<tr>
<td>fit()</td>
<td>(PyAstronomy.funcFit.FitterBase method), 266</td>
</tr>
<tr>
<td>fit()</td>
<td>(PyAstronomy.funcFit.NelderMead method), 269</td>
</tr>
<tr>
<td>fit()</td>
<td>(PyAstronomy.funcFit.OneDFit method), 282</td>
</tr>
<tr>
<td>fit()</td>
<td>(PyAstronomy.funcFit.ScipyFMIN method), 267</td>
</tr>
<tr>
<td>fitEMCEE()</td>
<td>(PyAstronomy.funcFit.OneDFit method), 297</td>
</tr>
<tr>
<td>fitMCMC()</td>
<td>(PyAstronomy.funcFit.OneDFit method), 283</td>
</tr>
<tr>
<td>fitMCMC()</td>
<td>(PyAstronomy.funcFit.SyncFitContainer method), 284</td>
</tr>
<tr>
<td>flux2photons()</td>
<td>(in module PyAstronomy.pyasl), 95</td>
</tr>
<tr>
<td>foldAt()</td>
<td>(in module PyAstronomy.pyasl), 43</td>
</tr>
<tr>
<td>forceUpdate()</td>
<td>(PyAstronomy.pyaslp.ExoplanetEU method), 173</td>
</tr>
<tr>
<td>Fourier</td>
<td>(class in PyAstronomy.pyTiming.pyPeriod), 199</td>
</tr>
<tr>
<td>freeze()</td>
<td>(PyAstronomy.funcFit.OneDFit method), 285</td>
</tr>
<tr>
<td>frozenParameters()</td>
<td>(PyAstronomy.funcFit.Params method), 273</td>
</tr>
<tr>
<td>ftest()</td>
<td>(in module PyAstronomy.pyasl), 159</td>
</tr>
<tr>
<td>FWHM()</td>
<td>(PyAstronomy.modelSuite.VoigtAstroP method), 345</td>
</tr>
<tr>
<td>gal_uvw()</td>
<td>(in module PyAstronomy.pyasl), 145</td>
</tr>
<tr>
<td>GaussFit1d</td>
<td>(class in PyAstronomy.funcFit), 249</td>
</tr>
<tr>
<td>GaussFit2d</td>
<td>(class in PyAstronomy.funcFit), 261</td>
</tr>
<tr>
<td>generalizedESD()</td>
<td>(in module PyAstronomy.pyasl), 61</td>
</tr>
<tr>
<td>get_ak()</td>
<td>(PyAstronomy.pyasl.BSEqSamp method), 58</td>
</tr>
<tr>
<td>get_juldate()</td>
<td>(in module PyAstronomy.pyasl.asl.astroTimeLegacy), 105</td>
</tr>
<tr>
<td>get_lagrange_1()</td>
<td>(in module PyAstronomy.pyasl), 41</td>
</tr>
<tr>
<td>get_lagrange_2()</td>
<td>(in module PyAstronomy.pyasl), 42</td>
</tr>
<tr>
<td>get_lagrange_3()</td>
<td>(in module PyAstronomy.pyasl), 42</td>
</tr>
<tr>
<td>get_lagrange_4()</td>
<td>(in module PyAstronomy.pyasl), 42</td>
</tr>
<tr>
<td>get_lagrange_5()</td>
<td>(in module PyAstronomy.pyasl), 42</td>
</tr>
<tr>
<td>get_nb()</td>
<td>(PyAstronomy.pyasl.BSEqSamp method), 58</td>
</tr>
<tr>
<td>get_rhok()</td>
<td>(PyAstronomy.pyasl.BSEqSamp method), 60</td>
</tr>
<tr>
<td>get_allData()</td>
<td>(PyAstronomy.pyasl.ExoplanetEU method), 173</td>
</tr>
<tr>
<td>get_allData()</td>
<td>(PyAstronomy.pyasl.ExoplanetsOrg method), 175</td>
</tr>
<tr>
<td>get_allData()</td>
<td>(PyAstronomy.pyasl.NasaExoplanetArchive method), 165</td>
</tr>
</tbody>
</table>

---

PyAstronomy Documentation, Release 0.15.2

Index

380
Index

**IFitterBase** (class in *PyAstronomy.funcFit*), 266

```python
info() (PyAstronomy.pyTiming.pyPeriod.Gls method), 195
```

```python
ingressDuration() (in module PyAstronomy.pyas1), 149
```

```python
ingressDuration_Rs() (in module PyAstronomy.pyas1), 150
```

```python
instrBroadGaussFast() (in module PyAstronomy.pyas1), 87
```

```python
intep() (in module PyAstronomy.pyas1), 65
```

```python
inverseAitoff() (in module PyAstronomy.pyas1), 115
```

```python
invertIndexSelection() (in module PyAstronomy.pyaC), 365
```

```python
isInTransit() (in module PyAstronomy.pyas1), 151
```

```python
jdcnv() (in module PyAstronomy.pyas1.asl.astroTimeLegacy), 107
```

```python
juldate() (in module PyAstronomy.pyas1.asl.astroTimeLegacy), 107
```

**K**

```python
KeplerEllipseModel (class in PyAstronomy.modelSuite), 334
```

```python
KeplerRVModel (class in PyAstronomy.modelSuite), 338
```

```python
KuruczModels (class in PyAstronomy.pyas1), 178
```

```python
KuruczMT (class in PyAstronomy.pyas1), 179
```

**L**

```python
lambertPhaseFunction() (in module PyAstronomy.pyas1), 158
```

```python
LimBrightTrans (class in PyAstronomy.modelSuite.XTran.limBrightTrans), 322
```

```python
listGridLinks() (PyAstronomy.pyas1.KuruczModels method), 179
```

```python
listObservatories() (in module PyAstronomy.pyas1), 141
```

```python
LLGauss (class in PyAstronomy.modelSuite), 339
```

```python
localTime() (in module PyAstronomy.pyas1.asl), 113
```

```python
log10ccfNoyes() (PyAstronomy.pyas1.SMW_RHK method), 22
```

```python
log10ccfRutten() (PyAstronomy.pyas1.SMW_RHK method), 22
```

```python
log10xvbv() (PyAstronomy.pyas1.Pizzolato2003 method), 26
```

```python
log10xbolbvh (PyAstronomy.pyas1.Pizzolato2003 method), 26
```

```python
log10xbolmass() (PyAstronomy.pyas1.Pizzolato2003 method), 27
```

```python
log10xmass() (PyAstronomy.pyas1.Pizzolato2003 method), 27
```

```python
logRpholNoyes() (PyAstronomy.pyas1.SMW_RHK method), 22
```

```python
LombScarige (class in PyAstronomy.pyTiming.pyPeriod), 201
```

```python
lumAndTeff() (PyAstronomy.pyas1.SpecTypeDeJager method), 37
```

```python
Lyatransmission (class in PyAstronomy.modelSuite), 346
```

**M**

```python
magToFluxDensity_bessel98() (in module PyAstronomy.pyas1), 19
```

```python
MandelAgolLC (class in PyAstronomy.modelSuite.XTran.forTrans.mandelAgol), 315
```

```python
MandelAgolNLLC (class in PyAstronomy.modelSuite.XTran.forTrans.mandelAgolNL), 320
```

```python
matrix2doutput() (in module PyAstronomy.pyas1), 367
```

```python
MCMCautoParameters() (PyAstronomy.funcFit.OneDFit method), 279
```

```python
mean() (PyAstronomy.funcFit.TraceAnalysis method), 302
```

```python
meanUnbStd() (PyAstronomy.pyas1.BSBase method), 55
```

```python
median() (PyAstronomy.funcFit.TraceAnalysis method), 302
```

```python
metallicity() (PyAstronomy.pyas1.KuruczMT method), 180
```

```python
MiniFunc (class in PyAstronomy.funcFit), 293
```

```python
MMSCETSTable (class in PyAstronomy.pyas1), 181
```

```python
modelAvailable() (PyAstronomy.pyas1.KuruczMT method), 180
```

```python
ModelNameIdentBase (class in PyAstronomy.funcFit), 290
```

```python
moonphase() (in module PyAstronomy.pyas1), 135
```

```python
moonpos() (in module PyAstronomy.pyas1), 135
```

```python
MultiGauss2d (class in PyAstronomy.funcFit), 263
```

```python
MultiVoigt1d (class in PyAstronomy.funcFit), 254
```

**N**

```python
NasaExoplanetArchive (class in PyAstronomy.pyas1), 164
```

```python
needsUpdate() (PyAstronomy.pyaC.pyaPermanent.PyAUpdateCycle method), 361
```

```python
needsUpdate() (PyAstronomy.pyas1.ExoplanetEU method), 173
```

```python
needsUpdate() (PyAstronomy.pyas1.ExoplanetEU2 method), 170
```

```python
needsUpdate() (PyAstronomy.pyas1.ExoplanetsOrg method), 175
```
needsUpdate() (PyAstronomy.pyasl.NasaExoplanetArchive method), 165
NelderMead (class in PyAstronomy.funcFit), 268
NestedLoop (class in PyAstronomy.pyaC), 365
numberOfFreeParams() (PyAstronomy.funcFit.OneDFit method), 286
numberOfFreeParams() (PyAstronomy.funcFit.Params method), 274
numberOfLines() (PyAstronomy.modelSuite.LLGauss method), 341
numberOfWalkers() (PyAstronomy.funcFit.TraceAnalysis method), 302
nutate() (in module PyAstronomy.pyasl), 119

O
observatory() (in module PyAstronomy.pyasl), 141
OneDFit (class in PyAstronomy.funcFit), 277

P
PalLC (class in PyAstronomy.modelSuite.XTran.palTrans), 307
PalLC_Rebin (in module PyAstronomy.modelSuite.XTran.palTrans), 311
PalLCKep (class in PyAstronomy.modelSuite.XTran.palTrans), 309
parameters() (PyAstronomy.funcFit.OneDFit method), 286
parameters() (PyAstronomy.funcFit.Params method), 274
parameterSet() (PyAstronomy.funcFit.TraceAnalysis method), 302
parameterSummary() (PyAstronomy.funcFit.OneDFit method), 286
parameterSummary() (PyAstronomy.funcFit.Params method), 274
Params (class in PyAstronomy.funcFit), 270
pattern() (PyAstronomy.pyasl.AbundancePatterns method), 25
pdmEquiBin() (PyAstronomy.pyTiming.pyPDM.PyPDM method), 205
pdmEquiBinCover() (PyAstronomy.pyTiming.pyPDM.PyPDM method), 206
pearsonr() (PyAstronomy.funcFit.TraceAnalysis method), 302
PeriodBase (class in PyAstronomy.pyTiming.pyPeriod), 197
phase() (PyAstronomy.pyTiming.pyPDM.PyPDM method), 206
photons2flux() (in module PyAstronomy.pyasl), 95
Pizzolato2003 (class in PyAstronomy.pyasl), 26
planck() (in module PyAstronomy.pyasl), 82
plot() (PyAstronomy.pyTiming.pyPeriod.Fourier method), 200
plot() (PyAstronomy.pyTiming.pyPeriod.Gls method), 195
plot() (PyAstronomy.pyTiming.pyPeriod.LombScargle method), 202
plot() (PyAstronomy.pyTiming.pyPeriod.PeriodBase method), 198
plotCorr() (PyAstronomy.funcFit.TraceAnalysis method), 303
plotCorrEnh() (PyAstronomy.funcFit.TraceAnalysis method), 303
plotDeviance() (PyAstronomy.funcFit.TraceAnalysis method), 304
plotFIP() (in module PyAstronomy.pyasl), 62
plotHist() (PyAstronomy.funcFit.TraceAnalysis method), 303
plotTrace() (PyAstronomy.funcFit.TraceAnalysis method), 303
plotTraceHist() (PyAstronomy.funcFit.TraceAnalysis method), 303
pnorm() (PyAstronomy.pyTiming.pyPeriod.Gls method), 195
pointDistGESD() (in module PyAstronomy.pyasl), 62
PolyFit1d (class in PyAstronomy.funcFit), 258
polyResOutlier() (in module PyAstronomy.pyasl), 63
positionAngle() (in module PyAstronomy.pyasl), 138
powerLevel() (PyAstronomy.pyTiming.pyPeriod.Fourier method), 200
powerLevel() (PyAstronomy.pyTiming.pyPeriod.Gls method), 195
powerLevel() (PyAstronomy.pyTiming.pyPeriod.LombScargle method), 202
powerLevel() (PyAstronomy.pyTiming.pyPeriod.PeriodBase method), 198
precess() (in module PyAstronomy.pyasl.asl.astroTimeLegacy), 109
precess_xyz() (in module PyAstronomy.pyasl.asl.astroTimeLegacy), 110
premat() (in module PyAstronomy.pyasl.asl.astroTimeLegacy), 111
Prob() (PyAstronomy.pyTiming.pyPeriod.Fourier method), 200
prob() (PyAstronomy.pyTiming.pyPeriod.Gls method), 195
Prob() (PyAstronomy.pyTiming.pyPeriod.LombScargle method), 201
prob() (PyAstronomy.pyTiming.pyPeriod.LombScargle
probInv() (PyAstronomy.pyTiming.pyPeriod.Gls method), 196
ProbInv() (PyAstronomy.pyTiming.pyPeriod.LombScargle method), 202
probInv() (PyAstronomy.pyTiming.pyPeriod.LombScargle method), 202
purgeKuruczData() (in module PyAstronomy.pyasl), 178
PyAAlgorithmFailure, 356
pyaC.pyaErrors.pyaOtherErrors (module), 356
pyaC.pyaErrors.pyaValErrs (module), 355
PyAConfig (class in PyAstronomy.pyaC.pyaPermanent), 357
PyADeprecationError, 356
PyADownloadError, 356
PyAErrTemplate (class in PyAstronomy.pyaC.pyaErrors), 354
PyAFileError, 356
PyAFloatingPointError, 355
PyAFS (class in PyAstronomy.pyaC.pyaPermanent), 358
PyAImportFailure, 356
PyANamespaceClash, 355
PyANetworkError, 356
PyANotImplemented, 356
PyAOrderError, 356
PyAParParameterConflict, 356
PyARequiredImport, 356
PyAstronomy.pysal.asl.astroTimeLegacy (module), 102
PyAUpdateCycle (class in PyAstronomy.pyaC.pyaPermanent), 361
PyAValError, 355
PyPDM (class in PyAstronomy.pyTiming.pyPDM), 205
quadExtreme() (in module PyAstronomy.pysal), 66
quantiles() (in module PyAstronomy.funcFit), 306
quantiles() (PyAstronomy.funcFit.TraceAnalysis method), 304
remove_option() (PyAstronomy.pyaC.pyaPermanent.PyAConfig method), 358
remove_section() (PyAstronomy.pyaC.pya Permanent.PyAConfig method), 358
removeConditionalRestriction() (PyAstronomy.funcFit.OneDFit method), 287
removeConditionalRestriction() (PyAstronomy.funcFit.Params method), 275
removeFile() (PyAstronomy.pyaC.pyaPermanent.PyAFS method), 360
renameParameter() (PyAstronomy.funcFit.Params method), 275
renameVariable() (PyAstronomy.funcFit.OneDFit method), 287
requestFile() (PyAstronomy.pyaC.pyaPermanent.PyAFS method), 360
requestModelGrid() (PyAstronomy.pysal.KuruczModels method), 179
restoreState() (PyAstronomy.funcFit.OneDFit method), 287
restoreState() (PyAstronomy.funcFit.Params method), 275
returnNyquist() (PyAstronomy.pyTiming.pyPeriod.TimeSeries method), 197
RmcL (class in PyAstronomy.modelSuite), 324
RmcLell (class in PyAstronomy.modelSuite), 326
roche_lobe_radius_eggleton() (in module PyAstronomy.pysal), 40
roche_voi_MC() (in module PyAstronomy.pysal), 41
roche_yz_extent() (in module PyAstronomy.pysal), 41
rochepot() (in module PyAstronomy.pysal), 39
rochepot_dl() (in module PyAstronomy.pysal), 39
rotBroad() (in module PyAstronomy.pysal), 88
RotBroadProfile (class in PyAstronomy.modelSuite), 348
S
SampCorr (class in PyAstronomy.pysal), 60
sampleEMCEE() (in module PyAstronomy.funcFit), 293
saveConfigToFile() (PyAstronomy.pyaC.pyaPermanent.PyAConfig method), 358
saveState() (PyAstronomy.funcFit.OneDFit method), 287
saveState() (PyAstronomy.funcFit.Params method), 275
Scanner (class in PyAstronomy.pyTiming.pyPDM), 205
teffToColor() (PyAstronomy.pyasl.Ramirez2005 method), 32

teffToColor_nop() (PyAstronomy.pyasl.Ramirez2005 method), 32

tempFromthermalBroadeningWidth() (in module PyAstronomy.pyasl), 91

thaw() (PyAstronomy.funcFit.OneDFit method), 289

thaw() (PyAstronomy.funcFit.Params method), 276

thawLineStrengths() (PyAstronomy.modelSuite.LLGauss method), 342

thawLineWidths() (PyAstronomy.modelSuite.LLGauss method), 342

thermalBroadeningWidth() (in module PyAstronomy.pyasl), 91

TimeSeries (class in PyAstronomy.pyTiming.pyPeriod), 197

toFile() (PyAstronomy.pyTiming.pyPeriod.Gls method), 196

touchFile() (PyAstronomy.pyaC.pyaPermanent.PyAFS method), 360

TraceAnalysis (class in PyAstronomy.funcFit), 300

transit_T1_T4_ell() (in module PyAstronomy.pyasl), 151

transitDuration() (in module PyAstronomy.pyasl), 148

transitDuration_Rs() (in module PyAstronomy.pyasl), 149

transitTimes() (in module PyAstronomy.pyasl), 155

transitVisibilityPlot() (in module PyAstronomy.pyasl), 157

TransmissionCurves (class in PyAstronomy.pyasl), 99

treatAsEqual() (PyAstronomy.funcFit.SyncFitContainer method), 299

turnIntoRebin() (in module PyAstronomy.funcFit), 294

twilightName() (in module PyAstronomy.pyasl), 137

U

unred() (in module PyAstronomy.pyasl), 92

untie() (PyAstronomy.funcFit.OneDFit method), 289

untie() (PyAstronomy.funcFit.Params method), 276

updateModel() (PyAstronomy.funcFit.OneDFit method), 289

updateModel() (PyAstronomy.funcFit.SyncFitContainer method), 299

V

vactoair() (in module PyAstronomy.pyasl), 77

vactoair() (PyAstronomy.pyasl.RefractiveIndexAV method), 74

vactoair2() (in module PyAstronomy.pyasl), 73

variance() (PyAstronomy.pyasl.BSBase method), 56

Voigt1d (class in PyAstronomy.funcFit), 252

VoigtAstroP (class in PyAstronomy.modelSuite), 343

W

warn() (in module PyAstronomy.pyaC.pyaErrors), 355

weekday() (in module PyAstronomy.pyasl.astroTimeLegacy), 112

whichEllInts() (PyAstronomy.modelSuite.XTran.palTrans.PalLC method), 309

write() (PyAstronomy.pyaC.SimIOF method), 369

writeldFitsSpec() (in module PyAstronomy.pyasl), 97

X

xyz() (in module PyAstronomy.pyasl.astroTimeLegacy), 112

Z

zerocross1d() (in module PyAstronomy.pyaC), 363