pyspeckit Documentation

Release 1.0.2.dev232

Adam Ginsburg

Sep 16, 2023

CONTENTS

An extensible spectroscopic analysis toolkit for astronomy.

If you're just getting started, see the *examples page*.

To cite pyspeckit, use https://ui.adsabs.harvard.edu/abs/2022AJ....163..291G/abstract and http://adsabs.harvard.edu/abs/2011ascl.soft09001G.

Part I

Downloads

- v1.0.1
- latest commit from github (same as above, or also *Install pyspeckit via GitHub*)
- latest commit from bitbucket (see *Installation and Requirements*)
- pypi entry.

Supported file types and their formats:

- *FITS*
- *Plain Text*
- *hdf5*

Part II

Guides / Getting Started

If you're already a python user, go straight to the *examples page* to get a quick start. For simple gaussian fitting, *this example* is a good starting point.

- ***Guide for GILDAS-CLASS users***

  A simple getting started guide aimed at Gildas-CLASS users

- ***Guide for IRAF users***

  Intended for users of IRAF's `splot` interactive fitting routine.

Part III

Classes and API

At the core, PySpecKit runs on a 'Spectroscopic Object' class called `Spectrum`. Therefore everything interesting about PySpecKit can be learned by digging into the properties of this class.

- *spectrum* can read a variety of individual spectra types
  - `Spectrum` The Spectrum class, which is the core of pyspeckit. The `__init__` procedure opens a spectrum file.
  - `Spectra` A group of `Spectrum` s. Generally for when you have multiple wavelength observations you want to stitch together (e.g., two filterbanks on a heterodyne system, or the red/blue spectra from a multi-band spectrometer like the Double Imaging Spectrograph)
  - `ObsBlock` An Observation Block - multiple spectra of different objects or different times covering the same wavelength range
- *Cubes* is used to deal with data cubes and has functionality similar to GAIA and ds9.
  - `Cube` A Cube of Spectra. Has features to collapse the cube along the spectral axis and fit spectra to each element of the cube. Is meant to replicate Starlink's GAIA in some ways, but with less emphasis on speed and much greater emphasis on spectral line fitting.

Part IV

Features

- *Baseline Fitting* describes baseline & continuum fitting.
- *Model Fitting* describes the general process of model fitting.
- *Measurements* is a toolkit for performing EQW, column, and other measurements…
- *Units* contains the all-important `SpectroscopicAxis` class that is used to deal with coordinate transformations
- *Registration* describes the extensible qualities of pyspeckit

CHAPTER

ONE

INSTALLATION AND REQUIREMENTS

Hint: You can *pip_install* pyspeckit to get the latest release version:

```
pip install pyspeckit
```

PySpecKit requires at least the basic scientific packages:

- numpy
- matplotlib
- mpfit is included
- scipy is optional. It is only required for RADEX grid interpolation and certain types of optimization
- python2.7 or ordereddict for model parameter storage

You'll most likely want at least one of the following packages to enable file reading

- astropy >=0.4
- atpy (which depends on asciitable [github link])
- hdf5

If you have pip (see https://pypi.org/project/pyspeckit), you can install with:

```
pip install pyspeckit
```

Or the most recent version with either of these commands:

```
pip install https://github.com/pyspeckit/pyspeckit/archive/master.zip
```

You can acquire the code with this clone command (see also *Install pyspeckit via GitHub*):

```
git clone git@github.com:pyspeckit/pyspeckit.git pyspeckit
cd pyspeckit
python setup.py install
```

Or you can Download the latest zip version, then extract and install using the standard python method (but the pip install version of this is easier):

```
wget --no-check-certificate https://github.com/pyspeckit/pyspeckit/archive/refs/heads/master.
↪zip
unzip master.zip
cd pyspeckit-pyspeckit-[commit]
python setup.py install
```

You can also check out the source code

Note: If you use `easy_install pyspeckit` with the Enthought Python Distribution, you will most likely get a SandboxViolation error. You can get around this by using `python setup.py install` or `pip install pyspeckit`.

1.1 Install pyspeckit via GitHub

1.1.1 Quick

You can pip install the development version of pyspeckit:

```
pip install https://github.com/pyspeckit/pyspeckit/archive/master.zip
```

1.1.2 Installing a branch

If there's a bugfix branch, e.g., from a pull request, you can install it in a similar way. Just replace the branch name:

```
pip install https://github.com/pyspeckit/pyspeckit/archive/{branchname}.zip
```

Or, if the contribution is from a different user:

```
pip install https://github.com/{username}/pyspeckit/archive/{branchname}.zip
```

Where in both cases you need to replace `{{branchname}}` and `{username}`.

1.1.3 More detailed

If you want to help develop pyspeckit, follow these instructions:

Logged into your github account, go to https://github.com/pyspeckit/pyspeckit and click "Fork" in upper right.

Copy the URL of your pyspeckit fork https://github.com/*yourusername*/pyspeckit

On the command line type:

```
git clone https://github.com/yourusername/pyspeckit
```

(it will put it in a directory called pyspeckit in your working directory):

```
cd pyspeckit

git remote add upstream https://github.com/pyspeckit/pyspeckit
```

To get the most up to date version, type:

```
git pull upstream master
```

Update your personal "fork" to match `upstream/master`:

```
git push origin master
```

and enter your username and password if it asks.

Still in the `pyspeckit/` directory, type:

```
pip install -e .
```

You're good to go!

1.1.4 To make changes and generate a Pull request:

Create a new branch: `git checkout -b name_of_your_new_branch` This will automatically switch you to this new branch. Type `git branch` to see all the branches. The active one will be highlighted and have an asterisk next to it. To switch to an existing branch, type `git checkout name_of_branch`

After you make a change inside your local fork on your machine, type `git add changed_file` where changed_file is the name of the file(s) you edited.

Time to commit your change and add a little note about your change `git commit -m details` `details` should be a description of the changes you made, inside quotes

Push the change to GitHub: `git push origin name_of_branch` where name_of_branch is the branch you've been active in during this process.

If you want to contribute your changes to https://github.com/pyspeckit/pyspeckit, create a "pull request". In https://github.com/yourusername/pyspeckit, navigate to your branch where you pushed you want to merge with https://github.com/pyspeckit/pyspeckit and click "Pull request"

CHAPTER

TWO

MODELS

See *Parameters* for information on how to restrict/modify model parameters.

The generic SpectralModel class is a wrapper for model functions. A model should take in an X-axis and some number of parameters. In order to declare a SpectralModel, you give SpectralModel the function name and the number of parameters it requires. The rest of the options are optional, though parnames & shortvarnames are strongly recommended. If you do not specify fitunits, your fitting code must deal with units internally.

Here are some examples of how to make your own fitters:

```
hill5_fitter = model.SpectralModel(hill5_model, 5,
        parnames=['tau', 'v_lsr',  'v_infall',  'sigma', 'tpeak'],
        parlimited=[(True,False),(False,False),(True,False),(True,False), (True,False)],
        parlimits=[(0,0), (0,0), (0,0), (0,0), (0,0)],
        # specify the parameter names (TeX is OK)
        shortvarnames=("\\tau","v_{lsr}","v_{infall}","\\sigma","T_{peak}"),
        fitunits='Hz' )

gaussfitter = model.SpectralModel(gaussian, 3,
        parnames=['amplitude','shift','width'],
        parlimited=[(False,False),(False,False),(True,False)],
        parlimits=[(0,0), (0,0), (0,0)],
        shortvarnames=('A',r'\Delta x',r'\sigma'))
```

Then you can register these fitters.

2.1 Fitting

Once you have a model defined, you can fit it using the `pyspeckit.Spectrum.specfit` module. Documents on fitting have not been prepared yet, but you can learn most of the tricks by looking at the various fitting examples and the *Parameters* documentation.

See also *Model Fitting*.

2.2 Related Documents

2.2.1 Parameters

Model parameters are very flexible, and can be accessed and modified in many parallel ways.

The `parinfo` class is built on top of lmfit-py's parameters for compatibility with lmfit-py, but it builds on that. The code for the parameter overloading is in parinfo.py.

Simple Example

Start with a simple example: if you want to limit parameters to be within some range, use the `limits` and `limited` parameters.

```
# define shorthand first:
T,F = True,False
sp.specfit(fittype='gaussian', guesses=[-1,5,1,0.5,2,1],
    limits=[(0,0), (0,0), (0,0), (0,0), (0,0), (0,0)],
    limited=[(F,T), (F,F), (T,F), (T,F),  (F,F), (T,F)])
```

In this example, there are two gaussian components being fitted because a Gaussian takes 3 parameters, an amplitude, a center, and a width, and there are 6 parameters in the input guesses.

The first line is forced to be an absorption line: its limits are `(0,0)` but `limited=(F,T)` so only the 2nd parameter, the upper limit, is respected: the amplitude is forced to be $A \leq 0$.

The second line has its amplitude (the 4th parameter in `guesses`) forced *positive* since its limits are also `(0,0)` but its `limited=(T,F)`.

Both lines have their *widths* forced to be positive, which is true by default: there is no meaning to a negative width, since the width enters into the equation for a gaussian as σ^2.

Note that the need to limit parameters is the main reason for the existence of lmfit-py and mpfit.

Tying Parameters

It is also possible to explicitly state that one parameter depends on another. If, for example, you want to fit two gaussians, but they must be at a fixed wavelength separation from one another (e.g., for fitting the [S II] doublet), use `tied`:

```
sp.specfit(fittype='gaussian', guesses=[1,6716,1,0.5,6731,1],
    tied=['','','','','p[1]',''])
```

If you use `lmfit-py` by specifying `use_lmfit=True`, you can use the more advanced mathematical constraints permitted by lmfit-py.

Optical fitting: The H-[NII] complex of a type-I Seyfert galaxy shows a more complete example using `tied`.

Making your own `parinfo`

You can also build a `parinfo` class directly. Currently, the best example of this is in `tests/test_formaldehyde_mm_radex.py`.

Here's an example of how you would set up a fit using `parinfo` directly.

Warning: There is a bug in the use_lmfit section of this code that keeps it from working properly. =(

```
amplitude0 = pyspeckit.parinfo.Parinfo(n=0, parname='Amplitude0',
    shortparname='$A_0$', value=1, limits=[0, 100], limited=(True,True))
width0 = pyspeckit.parinfo.Parinfo(n=2, parname='Width0',
    shortparname='$\sigma_0$', value=1, limits=(0, 0), limited=(True,False))
center0 = pyspeckit.parinfo.Parinfo(n=1, parname='Center0',
    shortparname='$\Delta x_0$', value=6716, limits=(0, 0), limited=(False,False))
amplitude1 = pyspeckit.parinfo.Parinfo(n=3, parname='Amplitude1',
    shortparname='$A_1$', value=1, limits=[0, 100], limited=(True,True))
width1 = pyspeckit.parinfo.Parinfo(n=5, parname='Width1',
    shortparname='$\sigma_1$', value=1, limits=(0, 0), limited=(True,False))
center1 = pyspeckit.parinfo.Parinfo(n=4, parname='Center1',
    shortparname='$\Delta x_1$', value=6731, limits=(0, 0),
    limited=(False,False), tied=center0)

parinfo = pyspeckit.parinfo.ParinfoList([amplitude0,center0,width0,amplitude1,center1,
↪width1])

sp.specfit(parinfo=parinfo, use_lmfit=True)
```

2.3 Generic Models and Tools

2.3.1 Gaussian Model

The Gaussian model implements a Gaussian function and wraps it in the generic fitter tools.

The simplest and most useful model.

Until 12/23/2011, gaussian fitting used the complicated and somewhat bloated gaussfitter.py code. Now, this is a great example of how to make your own model! Just make a function like gaussian and plug it into the SpectralModel class.

Module API

pyspeckit.spectrum.models.inherited_gaussfitter.**gaussian**(*xarr*, *amplitude*, *dx*, *width*, *return_components=False*, *normalized=False*, *return_hyperfine_components=False*)

Returns a 1-dimensional gaussian of form A*np.exp(-(x-dx)**2/(2*w**2))

Area is sqrt(2*pi*sigma^2)*amplitude - i.e., this is NOT a normalized gaussian, unless normalized=True in which case A = Area

Parameters

xarr : np.ndarray

array of x values

amplitude : float

Amplitude of the Gaussian, i.e. its peak value, unless normalized=True then A is the area of the gaussian

dx : float

Center or "shift" of the gaussian

width : float

Width of the gaussian (sigma)

return_components : bool

dummy variable; return_components does nothing but is required by all fitters

return_hyperfine_components : bool

dummy variable; does nothing but is required by all fitters

normalized : bool

Return a normalized Gaussian?

pyspeckit.spectrum.models.inherited_gaussfitter.**gaussian_fitter**()

Generator for Gaussian fitter class

pyspeckit.spectrum.models.inherited_gaussfitter.**gaussian_integral**(*amplitude*, *sigma*)

Integral of a Gaussian

pyspeckit.spectrum.models.inherited_gaussfitter.**gaussian_vheight_fitter**()

Generator for Gaussian fitter class

2.3.2 Voigt Profile model

Module API

pyspeckit.spectrum.models.inherited_voigtfitter.**voigt**(*xarr*, *amp*, *xcen*, *sigma*, *gamma*, *normalized=False*)

Normalized Voigt profile

z = (x+i*gam)/(sig*sqrt(2)) V(x,sig,gam) = Re(w(z))/(sig*sqrt(2*pi))

The area of V in this definition is 1. If normalized=False, then you can divide the integral of V by sigma*sqrt(2*pi) to get the area.

Original implementation converted from http://mail.scipy.org/pipermail/scipy-user/2011-January/028327.html (had an incorrect normalization and strange treatment of the input parameters)

Modified implementation taken from wikipedia, using the definition. http://en.wikipedia.org/wiki/Voigt_profile

Parameters

xarr : np.ndarray

The X values over which to compute the Voigt profile

amp : float

Amplitude of the voigt profile if normalized = True, amp is the AREA

xcen : float

The X-offset of the profile

sigma : float

The width / sigma parameter of the Gaussian distribution

gamma : float

The width / shape parameter of the Lorentzian distribution

normalized : bool

Determines whether "amp" refers to the area or the peak of the voigt profile

pyspeckit.spectrum.models.inherited_voigtfitter.**voigt_fitter**()

Generator for voigt fitter class

pyspeckit.spectrum.models.inherited_voigtfitter.**voigt_fwhm**(*sigma*, *gamma*)

Approximation to the Voigt FWHM from wikipedia

http://en.wikipedia.org/wiki/Voigt_profile

Parameters

sigma : float

The width / sigma parameter of the Gaussian distribution

gamma : float

The width / shape parameter of the Lorentzian distribution

pyspeckit.spectrum.models.inherited_voigtfitter.**voigt_moments**(*self*, **args*, ***kwargs*)

Get the spectral moments from the moments package. Use the gaussian width for the lorentzian width (not a great guess!)

2.3.3 LTE Molecule Model

There is a tool for modeling the full LTE spectrum of a molecule. It uses the JPL or CDMS databases through their astroquery interfaces.

A very simple example looks like this:

```
import astropy.units as u
from pyspeckit.spectrum.models.lte_molecule import get_molecular_parameters, generate_fitter,
↪ generate_model

freqs, aij, deg, EU, partfunc = get_molecular_parameters('CH3OH',
                                                         fmin=90*u.GHz,
                                                         fmax=100*u.GHz)
def modfunc(xarr, vcen, width, tex, column):
    return generate_model(xarr, vcen, width, tex, column, freqs=freqs, aij=aij,
                          deg=deg, EU=EU, partfunc=partfunc)

fitter = generate_fitter(modfunc, name="CH3OH")
```

Details can be found in the API documentation:

LTE Molecule Modeling Tool

Uses astroquery to obtain molecular parameters.

Equations are based on Mangum & Shirley 2015 (2015PASP..127..266M)

Module API

pyspeckit.spectrum.models.lte_molecule.**Jnu**(*nu*, *T*)

RJ equivalent temperature (MS15 eqn 24)

pyspeckit.spectrum.models.lte_molecule.**Jnu_cgs**(*nu*, *T*)

RJ equivalent temperature (MS15 eqn 24) (use cgs constants for speed)

pyspeckit.spectrum.models.lte_molecule.**generate_fitter**(*model_func*, *name*)

Generator for hnco fitter class

pyspeckit.spectrum.models.lte_molecule.**generate_model**(*xarr*, *vcen*, *width*, *tex*, *column*, *freqs*, *aij*, *deg*, *EU*, *partfunc*, *background=None*, *tbg=2.73*, *get_tau=False*, *get_tau_sticks=False*)

Model Generator

Parameters

xarr : Quantity array [Hz]

The X-axis (frequency)

vcen : Quantity [km/s]

The central velocity

width : Quantity [km/s]

The 1-sigma Gaussian line width [not FWHM]

tex : Quantity [K]

Excitation temperature

column : Quantity [cm^-2]

The total column density of the molecule. If specified as a value < 30, it will be assumed to be the log10 of the column density.

freqs : Quantity array [Hz]

The central frequency of the lines to model. Subsequent parameters, aij, deg, EU, must have the same shape as `freqs`.

aij : array [log s^-1]

The Einstein A coefficients, assumed to be in units of log10 of the rate in 1/s

deg : array

The transition degeneracy

EU : Quantity array [erg]

The upper state energy in ergs

partfunc : function

The partition function. Can also be specified as an array of values in Kelvin.

background : None

An additional background field to include in the radiative transfer model. Must be in Kelvin. [Presently not correctly implemented]

tbg : Quantity [K]

The background brightness temperature, defaulting to 2.73 for the CMB temperature. This background will be treated as uniform with frequency and subtracted.

get_tau : bool, default False

If specified, the optical depth in each frequency bin will be returned

get_tau_sticks : bool, default False

If specified, the optical depth in each transition will be returned

Examples

Example case to produce a model:

```
from pyspeckit.spectrum.models.lte_molecule import get_molecular_parameters, generate_
↪model, generate_fitter
freqs, aij, deg, eu, partfunc = get_molecular_parameters('CH3OH')
def modfunc(xarr, vcen, width, tex, column):
    return generate_model(xarr, vcen, width, tex, column, freqs=freqs, aij=aij,
                          deg=deg, EU=EU, partfunc=partfunc)

fitter = generate_fitter(modfunc, name="CH3OH")
```

pyspeckit.spectrum.models.lte_molecule.**get_molecular_parameters**(*molecule_name*, *tex=50*, *fmin=<Quantity 1. GHz>*, *fmax=<Quantity 1. THz>*, *catalog='JPL'*, ***kwargs*)

Get the molecular parameters for a molecule from the JPL or CDMS catalog

(this version should, in principle, be entirely self-consistent)

Parameters

molecule_name : string

The string name of the molecule (normal name, like CH3OH or CH3CH2OH, but it has to match the JPL catalog spec)

tex : float

Optional excitation temperature (basically checks if the partition function calculator works)

catalog : 'JPL' or 'CDMS'

Which catalog to pull from

fmin : quantity with frequency units

fmax : quantity with frequency units

The minimum and maximum frequency to search over

Examples

```
>>> from pyspeckit.spectrum.models.lte_molecule import get_molecular_parameters
>>> freqs, aij, deg, EU, partfunc = get_molecular_parameters('CH2CHCN',
...                                                          fmin=220*u.GHz,
...                                                          fmax=222*u.GHz,
                                                            )
>>> freqs, aij, deg, EU, partfunc = get_molecular_parameters('CH3OH',
...                                                          fmin=90*u.GHz,
...                                                          fmax=100*u.GHz)
```

pyspeckit.spectrum.models.lte_molecule.**line_tau**(*tex*, *total_column*, *partition_function*, *degeneracy*, *frequency*, *energy_upper*, *einstein_A=None*)

Given the excitation temperature of the state, total column density of the molecule, the partition function, the

degeneracy of the state, the frequency of the state, and the upper-state energy level, return the optical depth of that transition.

This is a helper function for the LTE molecule calculations. It implements the equations

$$\int \tau_\nu d\nu = \frac{c^2}{8\pi\nu^2} A_{ij} N_u \left[\exp\left(\frac{h\nu}{k_B T_{ex}}\right) - 1\right]$$

or

$$\tau_\nu = \frac{c^2}{8\pi\nu^2} A_{ij} N_u \phi_\nu \left[\exp\left(\frac{h\nu}{k_B T_{ex}}\right) - 1\right]$$

where

$$N_u = N_{tot} \frac{g_u}{Q} \exp\left(\frac{-E_u}{k_B T_{ex}}\right)$$

based on Equations 11 and 29 of Mangum & Shirley 2015 (2015PASP..127..266M)

The return value is therefore

$$\tau_\nu / \phi_\nu$$

The line profile function is, for a Gaussian, given by eqn A1:

$$\phi_\nu = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(\nu-\nu_0)^2}{2\sigma^2}\right]$$

pyspeckit.spectrum.models.lte_molecule.**line_tau_cgs**(*tex*, *total_column*, *partition_function*, *degeneracy*, *frequency*, *energy_upper*, *einstein_A*)

Given the excitation temperature of the state, total column density of the molecule, the partition function, the degeneracy of the state, the frequency of the state, and the upper-state energy level, return the optical depth of that transition.

Unlike `line_tau()`, this function requires inputs in CGS units.

This is a helper function for the LTE molecule calculations. It implements the equations

$$\int \tau_\nu d\nu = \frac{c^2}{8\pi\nu^2} A_{ij} N_u \left[\exp\left(\frac{h\nu}{k_B T_{ex}}\right) - 1\right]$$

or

$$\tau_\nu = \frac{c^2}{8\pi\nu^2} A_{ij} N_u \phi_\nu \left[\exp\left(\frac{h\nu}{k_B T_{ex}}\right) - 1\right]$$

where

$$N_u = N_{tot} \frac{g_u}{Q} \exp\left(\frac{-E_u}{k_B T_{ex}}\right)$$

based on Equations 11 and 29 of Mangum & Shirley 2015 (2015PASP..127..266M)

The return value is therefore

$$\tau_\nu / \phi_\nu$$

The line profile function is, for a Gaussian, given by eqn A1:

$$\phi_\nu = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(\nu-\nu_0)^2}{2\sigma^2}\right]$$

pyspeckit.spectrum.models.lte_molecule.**ntot_of_nupper**(*nupper*, *eupper*, *tex*, *Q_rot*, *degeneracy=1*)

Given an N_upper, E_upper, tex, Q_rot, and degeneracy for a single state, give N_tot

Mangum & Shirley 2015 eqn 31

$$N_{tot}/N_u = Q_{rot}/g_u \exp(E_u/kT_{ex})$$

Example:

```
>>> import astropy.units as u
>>> tex = 50*u.K
>>> kkms = 100*u.K*u.km/u.s
>>> from pyspeckit.spectrum.models import lte_molecule
>>> freqs, aij, deg, EU, partfunc = lte_molecule.get_molecular_parameters(molecule_
↪name='HNCO v=0', molecule_name_jpl='HNCO', fmin=87*u.GHz, fmax=88*u.GHz)
>>> nupper = lte_molecule.nupper_of_kkms(kkms, freqs, 10**aij)
>>> ntot = lte_molecule.ntot_of_nupper(nupper, EU*u.erg, tex, Q_rot=partfunc(tex),␣
↪degeneracy=deg)
```

pyspeckit.spectrum.models.lte_molecule.**nupper_of_kkms**(*kkms*, *freq*, *Aul*, *replace_bad=None*)

Mangum & Shirley 2015 eqn 82 gives, for the optically thin, Rayleigh-Jeans, negligible background approximation:

$$N_{tot} = (3k)/(8\pi^3 \nu S \mu^2 R_i)(Q/g) \exp(E_u/kT_{ex}) \int (T_R/f dv)$$

Eqn 31:

$$N_{tot}/N_u = Q_{rot}/g_u \exp(E_u/kT_{ex})$$
$$-> N_{tot} = N_u Q_{rot}/g_u \exp(E_u/kT_{ex}) -> Nu = N_{tot} g/Q_{rot} \exp(-E_u/kT_{ex})$$

To get Nu of an observed line, then:

$$NuQ_{rot}/g_u \exp(E_u/kT_{ex}) = (3k)/(8\pi^3 \nu S \mu^2 R_i)(Q/g) \exp(E_u/kT_{ex}) \int (T_R/f dv)$$

This term cancels:

$$Q_{rot}/g_u \exp(E_u/kT_{ex})$$

Leaving:

$$N_u = (3k)/(8\pi^3 \nu S \mu^2 R_i) \int (T_R/f dv)$$

$int(T_R/f dv)$ is the optically thin integrated intensity in K km/s dnu/nu = dv/c [doppler eqn], so to get $int(T_R dnu)$, sub in $dv = c/nu dnu$

$$N_u = (3kc)/(8\pi^3 \nu^2 S \mu^2 R_i) \int (T_R/f d\nu)$$

We then need to deal with the S mu^2 R_i term. We assume R_i = 1, since we are not measuring any hyperfine transitions (R_i is the hyperfine degeneracy; eqn 75) Equation 11:

$$A_{ul} = 64\pi^4 \nu^3/(3hc^3)|\mu_{ul}|^2$$

Equation 62:

$$|\mu_{ul}|^2 = S\mu^2$$
$$-> S\mu^2 = (3hc^3A_{ul})/(64\pi^4\nu^3)$$

Plugging that in gives

$$Nu = (3kc)/(8\pi^3\nu^2((3hc^3A_{ul})/(64\pi^4\nu^3))) \int (T_R/fd\nu) = (3kc64\pi^4\nu^3)/(8\pi^3\nu^23hc^3A_{ul}) \int (T_R/fd\nu) = (8\pi\nu k/(A_{ul}c^2h))$$

which is the equation implemented below. We could also have left this in dv units by substituting du = nu/c dv:

$$= (8\pi\nu^2k/(A_{ul}c^3h)) \int (T_R/fdv)$$

2.3.4 Hyperfine Model

The hyperfine line modeling tool is a generic tool for modeling hyperfine lines. Given a list of hyperfine line separations and weights, it will treat the full set of lines as a single component when fitting. There are many options for the free parameters, as illustrated below:

Module API

class pyspeckit.spectrum.models.hyperfine.**hyperfinemodel**(*line_names*, *voff_lines_dict*, *freq_dict*, *line_strength_dict*, *relative_strength_total_degeneracy*)

Wrapper for the hyperfine model class. Specify the offsets and relative strengths when initializing, then you've got yourself a hyperfine modeler.

There are a wide variety of different fitter attributes, each designed to free a different subset of the parameters. Their purposes should be evident from their names.

Initialize the various parameters defining the hyperfine transitions

Parameters

line_names: list :

list of the line names to be used as indices for the dictionaries

voff_lines_dict: dict :

a linename:v_off dictionary of velocity offsets for the hyperfine components. Technically, this is redundant with freq_dict

freq_dict: dict :

frequencies of the indvidual transitions

line_strength_dict: dict :

Relative strengths of the hyperfine components, usually determined by their degeneracy and Einstein A coefficients

hyperfine(*xarr*, *Tex=5.0*, *tau=0.1*, *xoff_v=0.0*, *width=1.0*, *return_hyperfine_components=False*, *Tbackground=2.73*, *amp=None*, *return_tau=False*, *tau_total=None*, *vary_hyperfine_tau=False*, *vary_hyperfine_width=False*)

Generate a model spectrum given an excitation temperature, optical depth, offset velocity, and velocity width.

Parameters

return_tau : bool

If specified, return just the tau spectrum, ignoring Tex

tau_total : bool

If specified, use this *instead of tau*, and it tries to normalize to the *peak of the line*

vary_hyperfine_tau : bool

If set to true, allows the hyperfine transition amplitudes to vary and does not use the line_strength_dict. If set, `tau` must be a dict

hyperfine_addbackground(*xarr*, *Tbackground=2.73*, *Tex=5.0*, *tau=0.1*, *xoff_v=0.0*, *width=1.0*, *return_tau=False*, ***kwargs*)

Identical to hyperfine, but adds Tbackground as a constant continuum level

hyperfine_amp(*xarr*, *amp=None*, *xoff_v=0.0*, *width=1.0*, *return_hyperfine_components=False*, *Tbackground=2.73*, *Tex=5.0*, *tau=0.1*)

wrapper of self.hyperfine with order of arguments changed

hyperfine_background(*xarr*, *Tbackground=2.73*, *Tex=5.0*, *tau=0.1*, *xoff_v=0.0*, *width=1.0*, *return_tau=False*, ***kwargs*)

Identical to hyperfine, but with Tbackground free. Assumes already background-subtracted

hyperfine_tau(*xarr*, *tau*, *xoff_v*, *width*, ***kwargs*)

same as hyperfine, but with arguments in a different order, AND tau is returned instead of exp(-tau)

hyperfine_tau_total(*xarr*, *tau_total*, *xoff_v*, *width*, ***kwargs*)

same as hyperfine, but with arguments in a different order, AND tau is returned instead of exp(-tau), AND the *peak* tau is used

hyperfine_varyhf(*xarr*, *Tex*, *xoff_v*, *width*, **args*, ***kwargs*)

Wrapper of hyperfine for using a variable number of peaks with specified tau

hyperfine_varyhf_amp(*xarr*, *xoff_v*, *width*, **args*, ***kwargs*)

Wrapper of hyperfine for using a variable number of peaks with specified amplitude (rather than tau). Uses some opaque tricks: Tex is basically ignored, and return_tau means you're actually returning the amplitude, which is just passed in as tau

hyperfine_varyhf_amp_width(*xarr*, *xoff_v*, **args*, ***kwargs*)

Wrapper of hyperfine for using a variable number of peaks with specified amplitude (rather than tau). Uses some opaque tricks: Tex is basically ignored, and return_tau means you're actually returning the amplitude, which is just passed in as tau

2.3.5 Model Grid

Fit a line based on parameters output from a grid of models

Module API

pyspeckit.spectrum.models.modelgrid.**gaussian_line**(*xax*, *maxamp*, *tau*, *offset*, *width*)

A Gaussian line function in which the

pyspeckit.spectrum.models.modelgrid.**line_model_2par**(*xax*, *center*, *width*, *gridval1*, *gridval2*, *griddim1*, *griddim2*, *maxampgrid*, *taugrid*, *linefunction=<function gaussian_line>*)

Returns the spectral line that matches the given x-axis

xax, center, width must be in the same units!

pyspeckit.spectrum.models.modelgrid.**line_params_2D**(*gridval1*, *gridval2*, *griddim1*, *griddim2*, *valuegrid*)

Given a 2D grid of modeled line values - the amplitude, e.g. excitation temperature, and the optical depth, tau - return the model spectrum

griddims contains the names of the axes and their values... it should have the same number of entries as gridpars

2.3.6 Spectral Template Model

Spectral Template Fitter

A tool to find the optimal shift and scaling for a given template model.

Module API

pyspeckit.spectrum.models.template.**spectral_template_generator**(*template_spectrum*, *xshift_units='km/s'*, *left=0*, *right=0*)

Given a spectral_template, return a model function with scale and shift as free parameters.

Parameters

template_spectrum: `pyspeckit.spectrum.classes.Spectrum` :

The template spectrum to fit

xshift_units: str :

The units of the shift parameter

left/right: float :

The left and right edge parameters used for extrapolating outside the template if the template is smaller than the input spectrum. These cannot be NaN.

Returns

spectral_template: function :

The model function that interpolates the template onto the given X-axis

pyspeckit.spectrum.models.template.**template_fitter**(*template_spectrum*, *xshift_units='km/s'*)

Generator for Spectral Template fitter class

Parameters

template_spectrum : pyspeckit.Spectrum

A valid spectrum to be scaled and shifted to match the input

xshift_units : str in pyspeckit.units.unit_type_dict

The units of the shift to fit. If you're using a velocity unit, make sure there's a reference X-unit for both the template spectrum and the input spectrum.

Examples

```
>>> template = pyspeckit.Spectrum("template_spectrum.fits")
>>> dataspec = pyspeckit.Spectrum("DataSpectrum.fits")
>>> template_fitter = pyspeckit.models.template_fitter(template,
...                                                    xshift_units='angstroms')
>>> dataspec.Registry.add_fitter('template',template_fitter, 2)
>>> dataspec.specfit(fittype='template',guesses=[1,0])
>>> print dataspec.specfit.parinfo
```

2.4 Specific Models

2.4.1 Ammonia Models

The Ammonia modeling tools include a set of constants in the `ammonia_constants` module and the following ammonia modeling tools listed below.

There is also an ammonia fitter wrapper; see *Wrappers*.

Note that there are two modules described here: the multi-rotational-transition fitter, which has its own set of custom functions, and a generic hyperfine-line fitting module meant to fit a single metastable (or non-metastable) transition.

If you run into a problem, have a look at the ammonia-tagged issues on github

Note that the default ammonia spectrum, with only the CMB as a background, is intentionally restricted to be positive to avoid nonphysical (absorption against the CMB) parameters. If you find your spectral fits sometimes returning errors about a negative spectrum, you may need the 'restricted excitation temperature' model. In this model, the excitation temperature is no longer a free parameter, but instead is set to trot + deltat, where deltat is a positive value. The only

difference from the main ammonia model is this restriction, which can result in more numerically stable results. An example can be found in the examples directory

Ammonia inversion transition TROT fitter translated from Erik Rosolowsky's https://github.com/low-sky/nh3fit

Module API

pyspeckit.spectrum.models.ammonia.**ammonia**(*xarr*, *trot=20*, *tex=None*, *ntot=14*, *width=1*, *xoff_v=0.0*, *fortho=0.0*, *tau=None*, *fillingfraction=None*, *return_tau=False*, *return_tau_profile=False*, *background_tb=2.7326*, *verbose=False*, *return_components=False*, *debug=False*, *line_names=['oneone', 'twotwo', 'threethree', 'fourfour', 'fivefive', 'sixsix', 'sevenseven', 'eighteight', 'ninenine']*, *ignore_neg_models=False*)

Generate a model Ammonia spectrum based on input temperatures, column, and gaussian parameters. The returned model will be in Kelvin (brightness temperature) units.

Note that astropy units are not used internally for performance reasons. A wrapped version of this module including those units would be a good idea, as it is definitely possible to implement this with unit support and good performance.

Parameters

xarr: `pyspeckit.spectrum.units.SpectroscopicAxis` :

Array of wavelength/frequency values

trot: float :

The rotational temperature of the lines. This is the excitation temperature that governs the relative populations of the rotational states.

tex: float or None :

Excitation temperature. Assumed LTE if unspecified (`None`) or if tex>trot. This is the excitation temperature for *all* of the modeled lines, which means we are explicitly assuming T_ex is the same for all lines.

ntot: float :

Total log column density of NH3. Can be specified as a float in the range 5-25

width: float :

Line width (Gaussian sigma) in km/s

xoff_v: float :

Line offset in km/s

fortho: float :

Fraction of NH3 molecules in ortho state. Default assumes all para (fortho=0).

tau: None or float :

If tau (optical depth in the 1-1 line) is specified, ntot is NOT fit but is set to a fixed value. The optical depths of the other lines are fixed relative to tau_oneone

fillingfraction: None or float :

fillingfraction is an arbitrary scaling factor to apply to the model

return_tau: bool :

Return a dictionary of the optical depths in each line instead of a synthetic spectrum

return_tau_profile: bool :

Return a dictionary of the optical depth profiles in each line, i.e., the optical depths that will be used in conjunction with T_ex to produce the synthetic spectrum

return_components: bool :

Return a list of arrays, one for each hyperfine component, instead of just one array

background_tb : float

The background brightness temperature. Defaults to TCMB.

ignore_neg_models: bool :

Normally if background=TCMB and the model is negative, an exception will be raised. This parameter will simply skip that exception. Use with extreme caution: negative models (absorption spectra against the CMB) are not physical! You may want to allow this in some cases because there can be numerical issues where the model goes negative when it shouldn't.

verbose: bool :

More messages

debug: bool :

For debugging.

Returns

spectrum: `numpy.ndarray` :

Synthetic spectrum with same shape as `xarr`

component_list: list :

List of `numpy.ndarray`'s, one for each hyperfine component

tau_dict: dict :

Dictionary of optical depth values for the various lines (if `return_tau` is set)

class pyspeckit.spectrum.models.ammonia.**ammonia_model**(*npeaks=1*, *npars=6*, *parnames=['trot', 'tex', 'ntot', 'width', 'xoff_v', 'fortho']*, ***kwargs*)

The basic Ammonia (NH3) model with 6 free parameters:

Trot, Tex, ntot, width, xoff_v, and fortho

Trot is the rotational temperature. It governs the relative populations of the rotational states, i.e., the relative strength of different transitions

Tex is the excitation temperature. It is assumed constant across all states, which is not always a good assumption - a radiative transfer and excitation model is required to constrain this, though.

ntot is the total column density of p-NH3 integrated over all states.

width is the linewidth (the Gaussian sigma)

xoff_v is the velocity offset / line of sight velocity

fortho is the ortho fraction (northo / (northo+npara))

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian

fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

annotations()

Return a list of TeX-formatted labels

The values and errors are formatted so that only the significant digits are displayed. Rounding is performed using the decimal package.

Parameters

shortvarnames : list

A list of variable names (tex is allowed) to include in the annotations. Defaults to self.shortvarnames

Examples

```
>>> # Annotate a Gaussian
>>> sp.specfit.annotate(shortvarnames=['A','\Delta x','\sigma'])
```

components(*xarr*, *pars*, *hyperfine=False*, *return_hyperfine_components=False*, ***kwargs*)

Ammonia components don't follow the default, since in Galactic astronomy the hyperfine components should be well-separated. If you want to see the individual components overlaid, you'll need to pass hyperfine to the plot_fit call

moments(*Xax*, *data*, *negamp=None*, *veryverbose=False*, ***kwargs*)

Returns a very simple and likely incorrect guess

multinh3fit(*xax*, *data*, *err=None*, *parinfo=None*, *quiet=True*, *shh=True*, *debug=False*, *maxiter=200*, *use_lmfit=False*, *veryverbose=False*, ***kwargs*)

Fit multiple nh3 profiles (multiple can be 1)

Parameters

xax : array

x axis

data : array

y axis

npeaks : int

How many nh3 profiles to fit? Default 1 (this could supersede onedgaussfit)

err : array

error corresponding to data

params : list

Fit parameters: [trot, tex, ntot (or tau), width, offset, ortho fraction] * npeaks If len(params) % 6 == 0, npeaks will be set to len(params) / 6. These parameters (and the related fixed, limited, min/max, names below) need to have length = 6*npeaks. If npeaks > 1 and length = 6, they will be replicated npeaks times, otherwise they will be reset to defaults:

fixed : list

Is parameter fixed?

limitedmin : list

minpars : list

set lower limits on each parameter (default: width>0, Tex and trot > Tcmb)

limitedmax : list

maxpars : list

set upper limits on each parameter

parnames : list

default parameter names, important for setting kwargs in model ['trot','tex','ntot','width','xoff_v','fortho']

quiet : bool

should MPFIT output each iteration?

shh : bool

output final parameters?

Returns

mpp : model parameter object

Fit parameters

model : array

The model array

errors : array

the fit errors

chi2 : float

the chi^2 value of the fit

`n_ammonia`(*pars=None*, *parnames=None*, ***kwargs*)

Returns a function that sums over N ammonia line profiles, where N is the length of trot,tex,ntot,width,xoff_v,fortho *OR* N = len(pars) / 6

The background "height" is assumed to be zero (you must "baseline" your spectrum before fitting)

***pars* [list]**

a list with len(pars) = (6-nfixed)n, assuming trot,tex,ntot,width,xoff_v,fortho repeated

parnames **[list]**

len(parnames) must = len(pars). parnames determine how the ammonia function parses the arguments

parse_3par_guesses(*guesses*)

Try to convert a set of interactive guesses (peak, center, width) into guesses appropriate to the model.

For NH3 models, we add in several extra parameters:

tex = 2.73 * peak trot = tex * 2 fortho = 0.5 ntot = 15

ntot is set to a constant ~10^15 because this results in optical depths near 1, so it forces the emission to be approximately significant. trot > tex so that we're in a physical regime to begin with.

We assume tex = peak + 2.73 because most spectra are shown background-subtracted (single dish are always that way, interferometric data are intrinsically that way...) and otherwise the guessing will crash if you guess a number < 2.73.

class pyspeckit.spectrum.models.ammonia.**ammonia_model_background**(***kwargs*)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

`moments`(*Xax*, *data*, *negamp=None*, *veryverbose=False*, ***kwargs*)

Returns a very simple and likely incorrect guess

`multinh3fit`(*xax*, *data*, *npeaks=1*, *err=None*, *params=(20, 20, 14, 1.0, 0.0, 0.5, 2.7326)*, *parnames=None*, *fixed=(False, False, False, False, False, False, True)*, *limitedmin=(True, True, True, True, False, True, True)*, *limitedmax=(False, False, False, False, False, True, True)*, *minpars=(2.7326, 2.7326, 0, 0, 0, 0, 2.7326)*, *parinfo=None*, *maxpars=(0, 0, 0, 0, 0, 1, 2.7326)*, *tied=('', '', '', '', '', '')*, *quiet=True*, *shh=True*, *veryverbose=False*, ***kwargs*)

Fit multiple nh3 profiles (multiple can be 1)

Parameters

xax : array

x axis

data : array

y axis

npeaks : int

How many nh3 profiles to fit? Default 1 (this could supersede onedgaussfit)

err : array

error corresponding to data

params : list

Fit parameters: [trot, tex, ntot (or tau), width, offset, ortho fraction] * npeaks If len(params) % 6 == 0, npeaks will be set to len(params) / 6. These parameters (and the related fixed, limited, min/max, names below) need to have length = 6*npeaks. If

npeaks > 1 and length = 6, they will be replicated npeaks times, otherwise they will be reset to defaults:

fixed : list

Is parameter fixed?

limitedmin : list

minpars : list

set lower limits on each parameter (default: width>0, Tex and trot > Tcmb)

limitedmax : list

maxpars : list

set upper limits on each parameter

parnames : list

default parameter names, important for setting kwargs in model ['trot','tex','ntot','width','xoff_v','fortho']

quiet : bool

should MPFIT output each iteration?

shh : bool

output final parameters?

Returns

mpp : model parameter object

Fit parameters

model : array

The model array

errors : array

the fit errors

chi2 : float

the chi^2 value of the fit

class pyspeckit.spectrum.models.ammonia.**ammonia_model_restricted_tex**(*parnames=['trot', 'tex', 'ntot', 'width', 'xoff_v', 'fortho', 'delta'], ammonia_func=<function ammonia>, **kwargs*)

Ammonia model with an explicitly restricted excitation temperature such that tex <= trot, set by the "delta" parameter (tex = trot - delta) with delta > 0. You can choose the ammonia funciton when you initialize it (e.g., `ammonia_model_restricted_tex(ammonia_func=ammonia)` or `ammonia_model_restricted_tex(ammonia_func=cold_ammonia)`)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

make_parinfo(*params=(20, 20, 0.5, 1.0, 0.0, 0.5, 0)*, *fixed=(False, False, False, False, False, False, False)*, *limitedmin=(True, True, True, True, False, True, True)*, *limitedmax=(False, False, False, False, False, True, False)*, *minpars=(2.7326, 2.7326, 0, 0, 0, 0, 0)*, *maxpars=(0, 0, 0, 0, 0, 1, 0)*, *tied=('', 'p[0]-p[6]', '', '', '', '', '')*, ***kwargs*)

parnames=['trot', 'tex', 'ntot', 'width', 'xoff_v', 'fortho', 'delta']

'delta' is the difference between tex and trot

n_ammonia(*pars=None*, *parnames=None*, ***kwargs*)

Returns a function that sums over N ammonia line profiles, where N is the length of trot,tex,ntot,width,xoff_v,fortho *OR* N = len(pars) / 6

The background "height" is assumed to be zero (you must "baseline" your spectrum before fitting)

***pars* [list]**

a list with len(pars) = (6-nfixed)n, assuming trot,tex,ntot,width,xoff_v,fortho repeated

***parnames* [list]**

len(parnames) must = len(pars). parnames determine how the ammonia function parses the arguments

class pyspeckit.spectrum.models.ammonia.**ammonia_model_vtau**(*parnames=['trot', 'tex', 'tau', 'width', 'xoff_v', 'fortho']*, ***kwargs*)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

make_parinfo(*params=(20, 14, 0.5, 1.0, 0.0, 0.5)*, *fixed=(False, False, False, False, False, False)*, *limitedmin=(True, True, True, True, False, True)*, *limitedmax=(False, False, False, False, False, True)*, *minpars=(2.7326, 2.7326, 0, 0, 0, 0)*, *maxpars=(0, 0, 0, 0, 0, 1)*, *tied=('', '', '', '', '', '')*, ***kwargs*)

parnames=['trot', 'tex', 'tau', 'width', 'xoff_v', 'fortho']

moments(*Xax*, *data*, *negamp=None*, *veryverbose=False*, ***kwargs*)

Returns a very simple and likely incorrect guess

class pyspeckit.spectrum.models.ammonia.**ammonia_model_vtau_thin**(*parnames=['tkin', 'tau', 'width', 'xoff_v', 'fortho']*, ***kwargs*)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

`make_parinfo`(*params=(20, 14, 1.0, 0.0, 0.5)*, *fixed=(False, False, False, False, False)*, *limitedmin=(True, True, True, False, True)*, *limitedmax=(False, False, False, False, True)*, *minpars=(2.7326, 0, 0, 0, 0)*, *maxpars=(0, 0, 0, 0, 1)*, *tied=('', '', '', '', '')*, ***kwargs*)

parnames=['trot', 'tex', 'tau', 'width', 'xoff_v', 'fortho']

moments(*Xax*, *data*, *negamp=None*, *veryverbose=False*, ***kwargs*)

Returns a very simple and likely incorrect guess

pyspeckit.spectrum.models.ammonia.**ammonia_thin**(*xarr*, *tkin=20*, *tex=None*, *ntot=14*, *width=1*, *xoff_v=0.0*, *fortho=0.0*, *tau=None*, *return_tau=False*, ***kwargs*)

Use optical depth in the 1-1 line as a free parameter The optical depths of the other lines are then set by the kinetic temperature

tkin is used to compute trot assuming a 3-level system consisting of (1,1), (2,1), and (2,2) as in Swift et al, 2005 [2005ApJ...620..823S]

pyspeckit.spectrum.models.ammonia.**cold_ammonia**(*xarr*, *tkin*, ***kwargs*)

Generate a model Ammonia spectrum based on input temperatures, column, and gaussian parameters

Parameters

xarr: `pyspeckit.spectrum.units.SpectroscopicAxis` :

Array of wavelength/frequency values

tkin: float :

The kinetic temperature of the lines in K. Will be converted to rotational temperature following the scheme of Swift et al 2005 (http://esoads.eso.org/abs/2005ApJ...620..823S, eqn A6) and further discussed in Equation 7 of Rosolowsky et al 2008 (http://adsabs.harvard.edu/abs/2008ApJS..175..509R)

class pyspeckit.spectrum.models.ammonia.**cold_ammonia_model**(*parnames=['tkin', 'tex', 'ntot', 'width', 'xoff_v', 'fortho']*, ***kwargs*)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

Ammonia inversion transition: Hyperfine-only fitter

Module API

pyspeckit.spectrum.models.ammonia_hf.**nh3_vtau_multimodel_generator**(*linenames*)

If you want to use multiple hyperfines for the same spectrum, use this generator. It is useful if you want N independent tau/tex values but the same velocity and linewidth

Parameters

linenames : list

A list of line names from the set ('oneone', ..., 'eighteight')

Returns

model : `model.SpectralModel`

A SpectralModel class build from N different metastable inversion hyperfine models

2.4.2 Formaldehyde Models

The Formaldehyde model is based on the *Hyperfine Model*. There is also a mm-line fitting module that uses preconstructed model grids to fit temperature from multiple transitions.

There is a formaldehyde fitter wrapper; see *Wrappers*.

This is a formaldehyde 1_11-1_10 / 2_12-2_11 fitter. It includes hyperfine components of the formaldehyde lines and has both LTE and RADEX LVG based models

Module API

pyspeckit.spectrum.models.formaldehyde.**formaldehyde**(*xarr*, *amp=1.0*, *xoff_v=0.0*, *width=1.0*, *return_hyperfine_components=False*, *texscale=0.01*, *tau=0.01*, ***kwargs*)

Generate a model Formaldehyde spectrum based on simple gaussian parameters

the "amplitude" is an essentially arbitrary parameter; we therefore define it to be Tex given tau=0.01 when passing to the fitter The final spectrum is then rescaled to that value

class pyspeckit.spectrum.models.formaldehyde.**formaldehyde_model**(*modelfunc*, *npars*, *shortvarnames=('A', '\\Delta x', '\\sigma')*, *fitunit=None*, *centroid_par=None*, *fwhm_func=None*, *fwhm_pars=None*, *integral_func=None*, *use_lmfit=False*, *guess_types=('amplitude', 'center', 'width')*, ***kwargs*)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

formaldehyde_integral(*modelpars*, *linename='oneone'*)

Return the integral of the individual components (ignoring height)

pyspeckit.spectrum.models.formaldehyde.**formaldehyde_pyradex**(*xarr*, *density=4*, *column=13*, *temperature=20*, *xoff_v=0.0*, *opr=1.0*, *width=1.0*, *tbackground=2.73*, *grid_vwidth=1.0*, *debug=False*, *verbose=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

grid_vwidth is the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230)

pyspeckit.spectrum.models.formaldehyde.**formaldehyde_radex**(*xarr*, *density=4*, *column=13*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *grid_vwidth_scale=False*, *texgrid=None*, *taugrid=None*, *hdr=None*, *path_to_texgrid=''*, *path_to_taugrid=''*, *temperature_gridnumber=3*, *debug=False*, *verbose=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

grid_vwidth is the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230) grid_vwidth_scale is True or False: False for LVG, True for Sphere

pyspeckit.spectrum.models.formaldehyde.**formaldehyde_radex_orthopara_temp**(*xarr*, *density=4*, *column=13*, *orthopara=1.0*, *temperature=15.0*, *xoff_v=0.0*, *width=1.0*, *Tbackground1=2.73*, *Tbackground2=2.73*, *grid_vwidth=1.0*, *grid_vwidth_scale=False*, *texgrid=None*, *taugrid=None*, *hdr=None*, *path_to_texgrid=''*, *path_to_taugrid=''*, *debug=False*, *verbose=False*, *getpars=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

grid_vwidth is the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230) grid_vwidth_scale is True or False: False for LVG, True for Sphere

pyspeckit.spectrum.models.formaldehyde.**formaldehyde_radex_tau**(*xarr*, *density=4*, *column=13*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *grid_vwidth_scale=False*, *taugrid=None*, *hdr=None*, *path_to_taugrid=''*, *temperature_gridnumber=3*, *debug=False*, *verbose=False*, *return_hyperfine_components=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

- uses hyperfine components
- assumes *tau* varies but *tex* does not!

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

grid_vwidth is the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230) grid_vwidth_scale is True or False: False for LVG, True for Sphere

This is a formaldehyde 3_03-2_02 / 3_22-221 and 3_03-2_02/3_21-2_20 fitter. It is based entirely on RADEX models.

Module API

pyspeckit.spectrum.models.formaldehyde_mm.**build_despotic_grids**(*gridfile='ph2co_grid_despotic.fits'*, *ph2coAbund=1e-08*, *nDens=21*, *logDensLower=2.0*, *logDensUpper=6.0*, *nCol=21*, *logColLower=11.0*, *logColUpper=15.0*, *nTemp=51*, *Tlower=10.0*, *Tupper=300.0*, *nDv=5*, *DvLower=1.0*, *DvUpper=5.0*)

Generates grids of p-H2CO line intensities using Despotic. Outputs a astropy Table.

Parameters

gridfile : string

Name of grid file to output.

ph2coAbund : float

Fractional abundance of p-H2CO

nDens : int

Number of grid points in the volume density

logDensLower : float

log of volume density at lower bound of grid (log(n/cm**-3))

logDensUpper : float

log of volume density at upper bound of grid (log(n/cm**-3))

nCol : int

Number of grid points in the column density

logColLower : float

log of column density of p-H2CO at lower bound of grid (log(N/cm**-2))

logColUpper : float

log of column density of p-H2CO at upper bound of grid (log(N/cm**-2))

nTemp : int

Number of grid points in the temperature grid

Tower : float

temperature at lower bound of grid (K)

Tupper : float

temperature at upper bound of grid (K)

nDv : int

Number of grid points in the line width

DvLower : float

line width (non-thermal) at lower bound of grid (km/s)

DvUpper : float

line width (non-thermal) at upper bound of grid (km/s)

pyspeckit.spectrum.models.formaldehyde_mm.**formaldehyde_mm**(*xarr*, *amp=1.0*, *xoff_v=0.0*, *width=1.0*, *return_components=False*)

Generate a model Formaldehyde spectrum based on simple gaussian parameters

the "amplitude" is an essentially arbitrary parameter; we therefore define it to be Tex given tau=0.01 when passing to the fitter The final spectrum is then rescaled to that value

The components are independent, but with offsets set by frequency... in principle.

pyspeckit.spectrum.models.formaldehyde_mm.**formaldehyde_mm_despotic**(*xarr*, *temperature=25*, *column=13*, *density=4*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *h2co_303_202=None*, *h2co_322_221=None*, *h2co_321_220=None*, *debug=False*, *verbose=False*, ***kwargs*)

Fitter to p-H2CO using despotic grids. Requires building grids and passing in functions for interpolating the h2co transition optical depth and excitation temperatures.

pyspeckit.spectrum.models.formaldehyde_mm.**formaldehyde_mm_despotic_functions**(*gridtable*)

This builds interpolation functions for use in fitting.

Parameters

gridtable : str

Name of grid in astropy table

Returns

h2co_303_202, h2co_322_221, h2co_321_220 : function

Functions that return the excitation temperature and optical depth given input density, temperature, column density and line width.

class pyspeckit.spectrum.models.formaldehyde_mm.**formaldehyde_mm_model**(*modelfunc*, *npars*, *shortvarnames=('A', '\\Delta x', '\\sigma')*, *fitunit=None*, *centroid_par=None*, *fwhm_func=None*, *fwhm_pars=None*, *integral_func=None*, *use_lmfit=False*, *guess_types=('amplitude', 'center', 'width')*, ***kwargs*)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

pyspeckit.spectrum.models.formaldehyde_mm.**formaldehyde_mm_radex**(*xarr*, *temperature=25*, *column=13*, *density=4*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *texgrid=None*, *taugrid=None*, *hdr=None*, *path_to_texgrid=''*, *path_to_taugrid=''*, *debug=False*, *verbose=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

Parameters

grid_vwidth : float

the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230)

density : float

Density!

Erik R's "fork" of the mm fitter:

This is a formaldehyde 3_03-2_02 / 3_22-221 and 3_03-2_02/3_21-2_20 fitter. It is based entirely on RADEX models.

This is the EWR fork of the fitter in pyspeckit.

Module API

pyspeckit.spectrum.models.h2co_mm.**formaldehyde_mm**(*xarr*, *amp=1.0*, *xoff_v=0.0*, *width=1.0*, *return_components=False*)

Generate a model Formaldehyde spectrum based on simple gaussian parameters

the "amplitude" is an essentially arbitrary parameter; we therefore define it to be Tex given tau=0.01 when passing to the fitter The final spectrum is then rescaled to that value

The components are independent, but with offsets set by frequency... in principle.

class pyspeckit.spectrum.models.h2co_mm.**formaldehyde_mm_model**(*modelfunc*, *npars*, *shortvarnames=('A', '\\Delta x', '\\sigma')*, *fitunit=None*, *centroid_par=None*, *fwhm_func=None*, *fwhm_pars=None*, *integral_func=None*, *use_lmfit=False*, *guess_types=('amplitude', 'center', 'width')*, ***kwargs*)

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

pyspeckit.spectrum.models.h2co_mm.**h2co_mm_radex**(*xarr*, *Temperature=25*, *logColumn=13*, *logDensity=4*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *gridbundle=None*, *debug=False*, *verbose=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

Parameters

grid_vwidth : float

the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230)

density : float

Density!

2.4.3 HCN Model

HCN is a molecule with hyperfine lines. It uses the hyperfine wrapper.

This is an HCN fitter... ref for line params: http://www.strw.leidenuniv.nl/~moldata/datafiles/hcn@hfs.dat

Module API

pyspeckit.spectrum.models.hcn.**aval_dict = {'10-01': 2.4075e-05, '11-01': 2.4075e-05, '12-01': 2.4075e-05}**

Line strengths of the 15 hyperfine components in J = 1 - 0 transition. The thickness of the lines indicates their relative weight compared to the others. Line strengths are normalized in such a way that summing over all initial J = 1 levels gives the degeneracy of the J = 0 levels, i.e., for JF1F = 012, three for JF1F = 011, and one for JF1F = 010. Thus, the sum over all 15 transitions gives the total spin degeneracy

pyspeckit.spectrum.models.hcn.**hcn_radex**(*xarr*, *density=4*, *column=13*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *grid_vwidth_scale=False*, *texgrid=None*, *taugrid=None*, *hdr=None*, *path_to_texgrid=''*, *path_to_taugrid=''*, *temperature_gridnumber=3*, *debug=False*, *verbose=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

grid_vwidth is the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230) grid_vwidth_scale is True or False: False for LVG, True for Sphere

2.4.4 Hill5 Infall Model

The Hill5 infall model is a specific realization of a collapsing cloud profile.

Code translated from: https://bitbucket.org/devries/analytic_infall/overview

Original source: http://adsabs.harvard.edu/abs/2005ApJ...620..800D

Module API

pyspeckit.spectrum.models.hill5infall.**hill5_model**(*xarr*, *tau*, *v_lsr*, *v_infall*, *sigma*, *tpeak*, *TBG=2.73*)

The hill5 from de Vries and Myers 2005. This model implicitly has zero optical depth in the envelope, no envelope velocity, and has a fixed background radiation temperature (see Table 2 in the paper).

Parameters

xarr : np.ndarray

array of x values

tau : float

tau_c, the core-center optical depth

v_lsr : float

The centroid velocity in km/s

v_infall : float

The infall velocity

sigma : float

The line width

tpeak : float

The peak brightness temperature

TBG : float

The background temperature

pyspeckit.spectrum.models.hill5infall.**jfunc**(*t*, *nu*)

t- kelvin nu - Hz?

2.4.5 N2H+ Model

N2H+ fitter

Reference for line params: Dore (Private Communication), improving on the determinations from L. Pagani, F. Daniel, and M. L. Dubernet A&A 494, 719-727 (2009) DOI: 10.1051/0004-6361:200810570

http://www.strw.leidenuniv.nl/~moldata/N2H+.html

http://adsabs.harvard.edu/abs/2005MNRAS.363.1083D

pyspeckit.spectrum.models.n2hp.**n2hp_radex**(*xarr*, *density=4*, *column=13*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *grid_vwidth_scale=False*, *texgrid=None*, *taugrid=None*, *hdr=None*, *path_to_texgrid=''*, *path_to_taugrid=''*, *temperature_gridnumber=3*, *debug=False*, *verbose=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

grid_vwidth is the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230) grid_vwidth_scale is True or False: False for LVG, True for Sphere

2.4.6 Ionized Hydrogen models

Hydrogen Models

Hydrogen in HII regions is typically assumed to follow Case B recombination theory.

The values for the Case B recombination coefficients are given by Hummer & Storey (1987). They are also computed in Hummer (1994) and tabulated at a wiki. I had to OCR and pull out by hand some of the coefficients.

However, Storey & Hummer 1995 might be a better resource now? https://cdsarc.unistra.fr/viz-bin/cat/VI/64

Module API

pyspeckit.spectrum.models.hydrogen.**add_to_registry**(*sp*)

Add the Hydrogen model to the Spectrum's fitter registry

pyspeckit.spectrum.models.hydrogen.**find_lines**(*xarr*)

Given a `pyspeckit.units.SpectrosopicAxis` instance, finds all the lines that are in bounds. Returns a list of line names.

pyspeckit.spectrum.models.hydrogen.**hydrogen_fitter**(*sp*, *temperature=10000*, *tiedwidth=False*)

Generate a set of parameters identifying the hydrogen lines in your spectrum. These come in groups of 3 assuming you're fitting a gaussian to each. You can tie the widths or choose not to.

***temperature* [5000, 10000, 20000]**

The case B coefficients are computed for 3 temperatures

***tiedwidth* [bool]**

Should the widths be tied?

Returns a list of `tied` and `guesses` in the xarr's units

pyspeckit.spectrum.models.hydrogen.**hydrogen_model**(*xarr*, *amplitude=1.0*, *width=0.0*, *velocity=0.0*, *a_k=0.0*, *temperature=10000*)

Generate a set of parameters identifying the hydrogen lines in your spectrum. These come in groups of 3 assuming you're fitting a gaussian to each. You can tie the widths or choose not to.

Parameters

sp : pyspeckit.Spectrum

The spectrum to fit

temperature : [5000, 10000, 20000]

The case B coefficients are computed for 3 temperatures

a_k : float

The K-band extinction normalized to 2.2 microns. Simple exponential.

width : float

Line width in km/s

velocity : float

Line center in km/s

amplitude : float

arbitrary amplitude of the first line (all other lines will be scaled to this value)

Returns

np.ndarray with same shape as sp.xarr :

pyspeckit.spectrum.models.hydrogen.**parse_storey1995**(*data*, *e_or_r='E'*)

Returns a dictionary with keys like:" (100.0, 1, 1000.0, 'B', 2, 124)

where the keys are

dens, Z, temp, case, nmin, nc

dens is the density in cm^-3, Z is the number of electrons, temp is the temperature in K, case is case A or B recombination (Case B, no transitions into n=1, no Lyman transitions, are included) nmin and nc - I don't know what these are

Each entry contains a mapping from [NU,NL] -> emissivity, where the emissivity is in erg/s/cm^3 and NU, NL are the upper and lower electronic levels

pyspeckit.spectrum.models.hydrogen.**retrieve_storey1995**(*temperature=10000*, *case='b'*)

Retrieve the Hummer and Storey tables from Vizier: https://cdsarc.cds.unistra.fr/viz-bin/cat/VI/64

based on the temperature and recombination case.

The data are parsed into a dictionary using the function parse_storey1995 below

pyspeckit.spectrum.models.hydrogen.**rrl**(*n*, *dn=1*, *amu=1.007825*, *Z=1*)

compute Radio Recomb Line freqs in GHz from Brown, Lockman & Knapp ARAA 1978 16 445

UPDATED:

Gordon & Sorochenko 2009, eqn A6

Parameters

n : int

The number of the lower level of the recombination line (H1a is Lyman alpha, for example)

dn : int

The delta-N of the transition. alpha=1, beta=2, etc.

amu : float

The mass of the central atom

Z : int

The ionization parameter for the atom. Z=1 is neutral, Z=2 is singly ionized, etc. For hydrogen, only z=1 makes sense, since ionized hydrogen has no electrons and therefore cannot have recombination lines.

Returns

frequency in GHz :

2.5 API Documentation for Models

We include the API documentation for the generic model and fitter wrappers here.

2.5.1 Module API

class pyspeckit.spectrum.models.model.**AstropyModel**(*model*, *shortvarnames=None*, ***kwargs*)

Override the SpectralModel initialization

fitter(*xax*, *data*, *err=None*, *quiet=True*, *veryverbose=False*, *debug=False*, *parinfo=None*, *params=None*, *npeaks=None*, ***kwargs*)

Run the fitter using mpfit.

kwargs will be passed to _make_parinfo and mpfit.

Parameters

xax : SpectroscopicAxis

The X-axis of the spectrum

data : ndarray

The data to fit

err : ndarray (optional)

The error on the data. If unspecified, will be uniform unity

parinfo : ParinfoList

The guesses, parameter limits, etc. See `pyspeckit.spectrum.parinfo` for details

quiet : bool

pass to mpfit. If False, will print out the parameter values for each iteration of the fitter

veryverbose : bool

print out a variety of mpfit output parameters

debug : bool

raise an exception (rather than a warning) if chi^2 is nan

n_modelfunc(*pars=None*, *debug=False*, ***kwargs*)

Only deals with single-peak functions

class pyspeckit.spectrum.models.model.**SpectralModel**(*modelfunc*, *npars*, *shortvarnames=('A', '\\Delta x', '\\sigma')*, *fitunit=None*, *centroid_par=None*, *fwhm_func=None*, *fwhm_pars=None*, *integral_func=None*, *use_lmfit=False*, *guess_types=('amplitude', 'center', 'width')*, ***kwargs*)

A wrapper class for a spectra model. Includes internal functions to generate multi-component models, annotations, integrals, and individual components. The declaration can be complex, since you should name individual variables, set limits on them, set the units the fit will be performed in, and set the annotations to be used. Check out some of the hyperfine codes (hcn, n2hp) for examples.

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

`analytic_centroids`(*centroidpar=None*)

Return the *analytic* centroids of the model components

Parameters

centroidpar : None or string

The name of the parameter in the fit that represents the centroid *some models have default centroid parameters - these will be used if centroidpar is unspecified*

Returns

List of the centroid values (even if there's only 1) :

`analytic_fwhm`(*parinfo=None*)

Return the FWHMa of the model components *if* a fwhm_func has been defined

Done with incomprehensible list comprehensions instead of nested for loops... readability sacrificed for speed and simplicity. This is unpythonic.

`analytic_integral`(*modelpars=None*, *npeaks=None*, *npars=None*)

Placeholder for analyic integrals; these must be defined for individual models

`annotations`(*shortvarnames=None*, *debug=False*)

Return a list of TeX-formatted labels

The values and errors are formatted so that only the significant digits are displayed. Rounding is performed using the decimal package.

Parameters

shortvarnames : list

A list of variable names (tex is allowed) to include in the annotations. Defaults to self.shortvarnames

Examples

```
>>> # Annotate a Gaussian
>>> sp.specfit.annotate(shortvarnames=['A','\Delta x','\sigma'])
```

component_integrals(*xarr*, *dx=None*)

Compute the integrals of each component

components(*xarr*, *pars*, ***kwargs*)

Return a numpy ndarray of shape [npeaks x modelshape] of the independent components of the fits

computed_centroid(*xarr=None*)

Return the *computed* centroid of the model

Parameters

xarr : None or np.ndarray

The X coordinates of the model over which the centroid should be computed. If unspecified, the centroid will be in pixel units

fitter(*xax*, *data*, *err=None*, *quiet=True*, *veryverbose=False*, *debug=False*, *parinfo=None*, ***kwargs*)

Run the fitter using mpfit.

kwargs will be passed to _make_parinfo and mpfit.

Parameters

xax : SpectroscopicAxis

The X-axis of the spectrum

data : ndarray

The data to fit

err : ndarray (optional)

The error on the data. If unspecified, will be uniform unity

parinfo : ParinfoList

The guesses, parameter limits, etc. See `pyspeckit.spectrum.parinfo` for details

quiet : bool

pass to mpfit. If False, will print out the parameter values for each iteration of the fitter

veryverbose : bool

print out a variety of mpfit output parameters

debug : bool

raise an exception (rather than a warning) if chi^2 is nan

get_emcee_ensemblesampler(*xarr*, *data*, *error*, *nwalkers*, ***kwargs*)

Get an emcee walker ensemble for the data & model

Parameters

data : np.ndarray

error : np.ndarray

nwalkers : int

Number of walkers to use

Examples

```
>>> import pyspeckit
>>> x = pyspeckit.units.SpectroscopicAxis(np.linspace(-10,10,50), unit='km/s')
>>> e = np.random.randn(50)
>>> d = np.exp(-np.asarray(x)**2/2.)*5 + e
>>> sp = pyspeckit.Spectrum(data=d, xarr=x, error=np.ones(50)*e.std())
>>> sp.specfit(fittype='gaussian')
>>> nwalkers = sp.specfit.fitter.npars * 2
>>> emcee_ensemble = sp.specfit.fitter.get_emcee_ensemblesampler(sp.xarr, sp.data,␣
↪sp.error, nwalkers)
>>> p0 = np.array([sp.specfit.parinfo.values] * nwalkers)
>>> p0 *= np.random.randn(*p0.shape) / 10. + 1.0
>>> pos,logprob,state = emcee_ensemble.run_mcmc(p0,100)
```

get_emcee_sampler(*xarr*, *data*, *error*, ***kwargs*)

Get an emcee walker for the data & model

Parameters

xarr : pyspeckit.units.SpectroscopicAxis

data : np.ndarray

error : np.ndarray

Examples

```
>>> import pyspeckit
>>> x = pyspeckit.units.SpectroscopicAxis(np.linspace(-10,10,50), unit='km/s')
>>> e = np.random.randn(50)
>>> d = np.exp(-np.asarray(x)**2/2.)*5 + e
>>> sp = pyspeckit.Spectrum(data=d, xarr=x, error=np.ones(50)*e.std())
>>> sp.specfit(fittype='gaussian')
>>> emcee_sampler = sp.specfit.fitter.get_emcee_sampler(sp.xarr, sp.data, sp.error)
>>> p0 = sp.specfit.parinfo
>>> emcee_sampler.run_mcmc(p0,100)
```

get_pymc(*xarr*, *data*, *error*, *use_fitted_values=False*, *inf=inf*, *use_adaptive=False*, *return_dict=False*, ***kwargs*)

Create a pymc MCMC sampler. Defaults to 'uninformative' priors

Parameters

data : np.ndarray

error : np.ndarray

use_fitted_values : bool

Each parameter with a measured error will have a prior defined by the Normal distribution with sigma = par.error and mean = par.value

use_adaptive : bool

Use the Adaptive Metropolis-Hastings sampler?

Examples

```
>>> x = pyspeckit.units.SpectroscopicAxis(np.linspace(-10,10,50), unit='km/s')
>>> e = np.random.randn(50)
>>> d = np.exp(-np.asarray(x)**2/2.)*5 + e
>>> sp = pyspeckit.Spectrum(data=d, xarr=x, error=np.ones(50)*e.std())
>>> sp.specfit(fittype='gaussian')
>>> MCuninformed = sp.specfit.fitter.get_pymc(sp.xarr, sp.data, sp.error)
>>> MCwithpriors = sp.specfit.fitter.get_pymc(sp.xarr, sp.data, sp.error, use_
↪fitted_values=True)
>>> MCuninformed.sample(1000)
>>> MCuninformed.stats()['AMPLITUDE0']
>>> # WARNING: This will fail because width cannot be set <0, but it may randomly␣
↪reach that...
>>> # How do you define a likelihood distribution with a lower limit?!
>>> MCwithpriors.sample(1000)
>>> MCwithpriors.stats()['AMPLITUDE0']
```

integral(*modelpars*, *dx=None*, ***kwargs*)

Extremely simple integrator: IGNORES modelpars; just sums self.model

lmfitfun(*x*, *y*, *err=None*, *debug=False*)

Wrapper function to compute the fit residuals in an lmfit-friendly format

lmfitter(*xax*, *data*, *err=None*, *parinfo=None*, *quiet=True*, *debug=False*, ***kwargs*)

Use lmfit instead of mpfit to do the fitting

Parameters

xax : SpectroscopicAxis

The X-axis of the spectrum

data : ndarray

The data to fit

err : ndarray (optional)

The error on the data. If unspecified, will be uniform unity

parinfo : ParinfoList

The guesses, parameter limits, etc. See `pyspeckit.spectrum.parinfo` for details

quiet : bool

If false, print out some messages about the fitting

logp(*xarr*, *data*, *error*, *pars=None*)

Return the log probability of the model. If the parameter is out of range, return -inf

mpfitfun(*x*, *y*, *err=None*)

Wrapper function to compute the fit residuals in an mpfit-friendly format

n_modelfunc(*pars=None*, *debug=False*, ***kwargs*)

Simple wrapper to deal with N independent peaks for a given spectral model

parse_3par_guesses(*guesses*)

Try to convert a set of interactive guesses (peak, center, width) into guesses appropriate to the model.

slope(*xinp*)

Find the local slope of the model at location x (x must be in xax's units)

pyspeckit.spectrum.models.model.**parse_offset_guess**(*gname*, *gval*)

Utility function for handling guesses. Allows guess types to be specified as 'amplitude*2' or 'width+3'.

Adds a variable height (background) component to any model

2.5.2 Module API

CHAPTER

THREE

BASIC PLOTTING GUIDE

The plotting tool in pyspeckit is intended to make publication-quality plots straightforward to produce.

For details on the various plotting tools, please see the `examples` and the `plotter documentation`.

A few basic examples are shown in the snippet below, with comments describing the various steps. This example (https://github.com/jmangum/spectrumplot) shows how to use pyspeckit with spectral_cube to extract and label spectra. Other examples:

```
import numpy as np
from astropy import units as u
import pyspeckit

xaxis = np.linspace(-50,150,100.) * u.km/u.s
sigma = 10. * u.km/u.s
center = 50. * u.km/u.s
synth_data = np.exp(-(xaxis-center)**2/(sigma**2 * 2.))

# Add noise
stddev = 0.1
noise = np.random.randn(xaxis.size)*stddev
error = stddev*np.ones_like(synth_data)
data = noise+synth_data

# this will give a "blank header" warning, which is fine
sp = pyspeckit.Spectrum(data=data, error=error, xarr=xaxis,
                        unit=u.erg/u.s/u.cm**2/u.AA)

sp.plotter()
sp.plotter.savefig('basic_plot_example.png')

# Fit with automatic guesses
sp.specfit(fittype='gaussian')
# (this will produce a plot overlay showing the fit curve and values)
sp.plotter.savefig('basic_plot_example_withfit.png')

# Redo the overlay with no annotation

# remove both the legend and the model overlay
sp.specfit.clear()
# then re-plot the model without an annotation (legend)
sp.specfit.plot_fit(annotate=False)
```

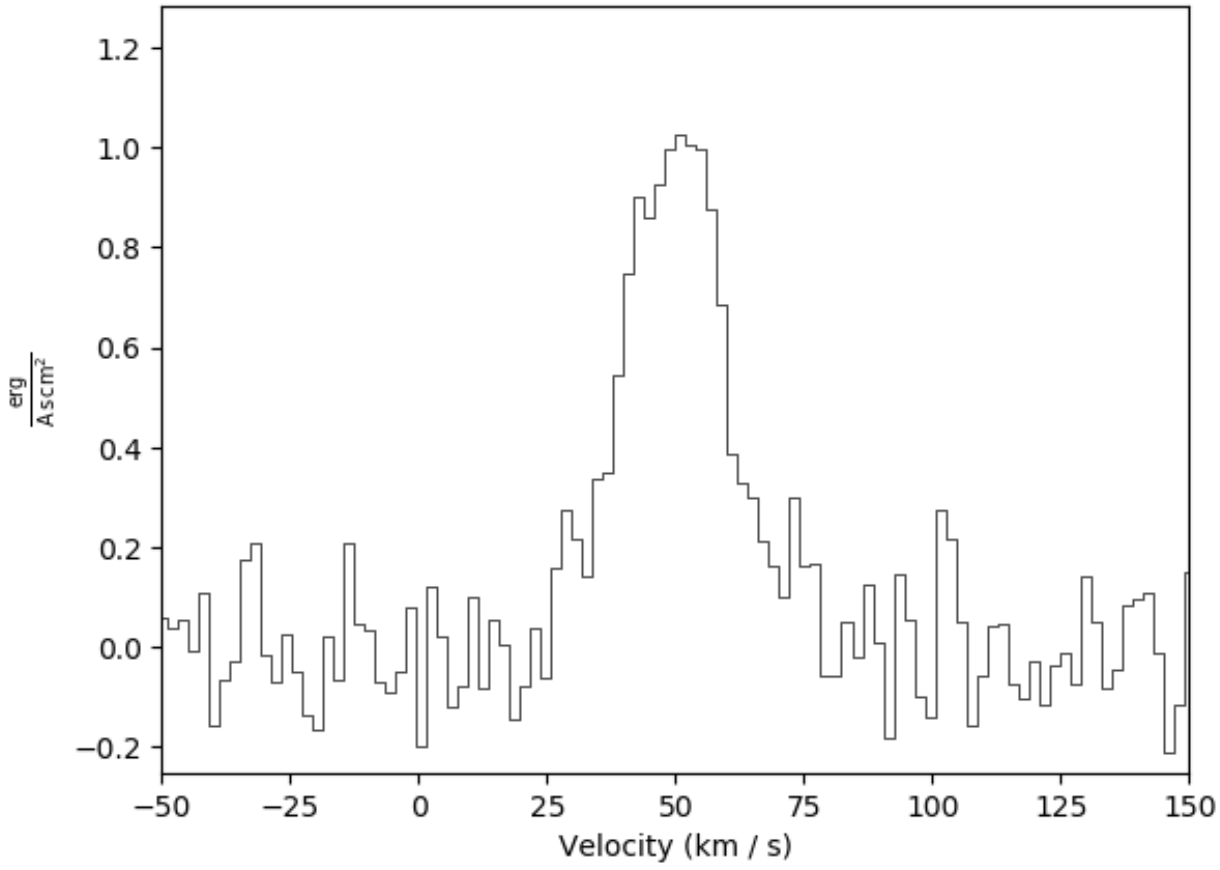
(continues on next page)

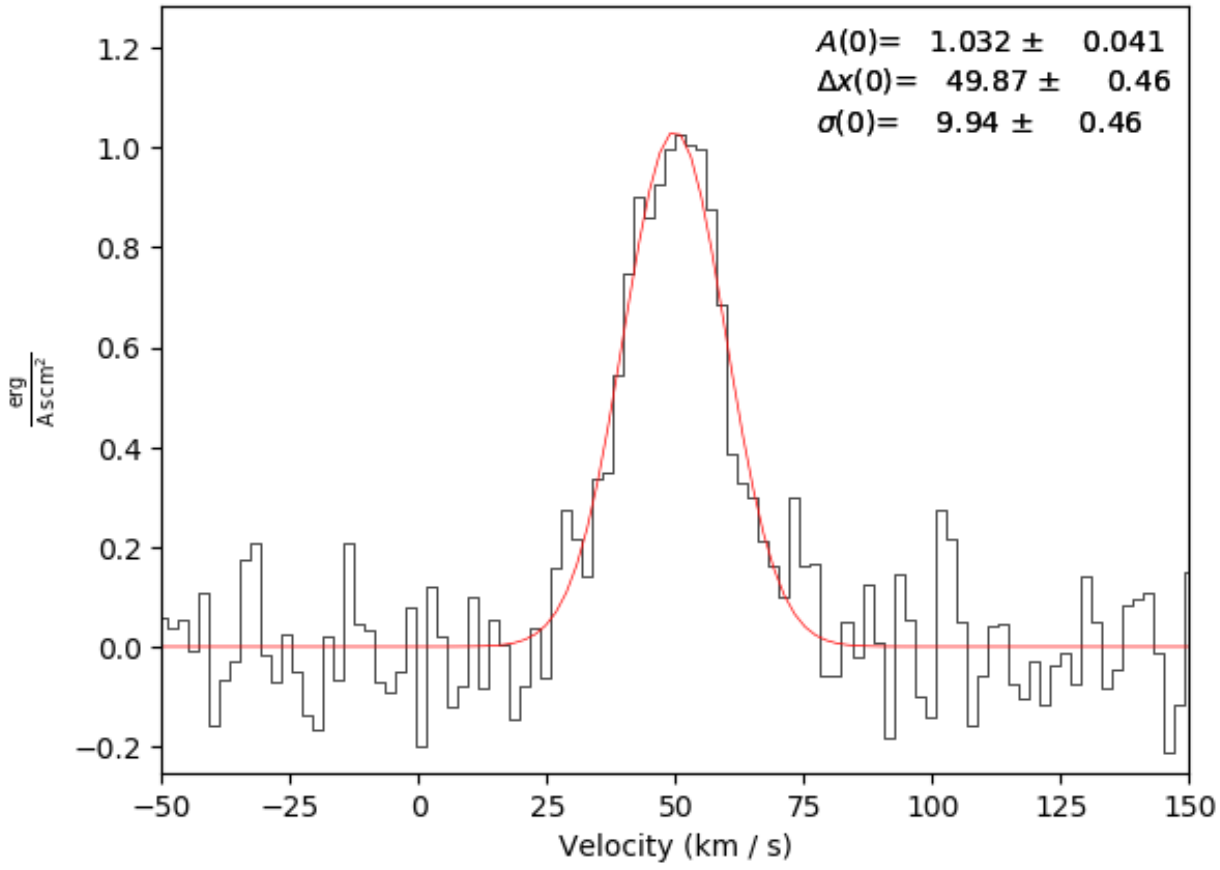
(continued from previous page)

```
sp.plotter.savefig('basic_plot_example_withfit_no_annotation.png')


# overlay another spectrum
# We use the 'synthetic' spectrum with no noise, then shift it by 10 km/s
sp2 = pyspeckit.Spectrum(data=synth_data, error=None, xarr=xaxis+10*u.km/u.s,
                         unit=u.erg/u.s/u.cm**2/u.AA)

# again, remove the overlaid model fit
sp.specfit.clear()

# to overplot, you need to tell the plotter which matplotlib axis to use and
# tell it not to clear the plot first
sp2.plotter(axis=sp.plotter.axis,
            clear=False,
            color='g')

# sp2.plotter and sp.plotter can both be used here (they refer to the same axis
# and figure now)
sp.plotter.savefig('basic_plot_example_with_second_spectrum_overlaid_in_green.png')

# the plot window will follow the last plotted spectrum's limits by default;
# that can be overridden with the xmin/xmax keywords
sp2.plotter(axis=sp.plotter.axis,
            xmin=-100, xmax=200,
            ymin=-0.5, ymax=1.5,
            clear=False,
            color='g')
sp.plotter.savefig('basic_plot_example_with_second_spectrum_overlaid_in_green_wider_limits.
↪png')


# you can also offset the spectra and set different
# this time, we need to clear the axis first, then do a fresh overlay

# fresh plot
sp.plotter(clear=True)

# overlay, shifted down by 0.2 in y and with a wider linewidth
sp2.plotter(axis=sp.plotter.axis,
            offset=-0.2,
            clear=False,
            color='r',
            linewidth=2,
            alpha=0.5,
           )
# you can also modify the axis properties directly
sp.plotter.axis.set_ylim(-0.25, 1.1)
sp2.plotter.savefig('basic_plot_example_with_second_spectrum_offset_overlaid_in_red.png')
```

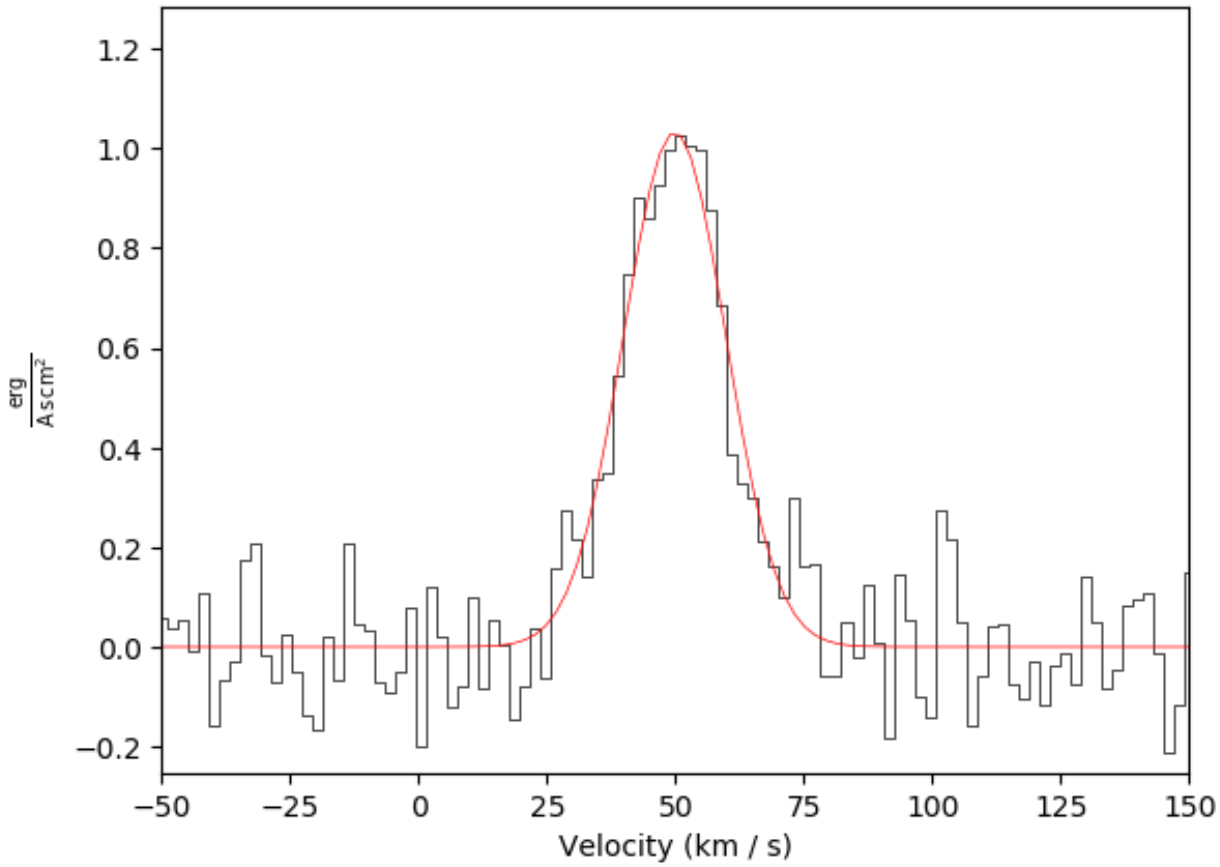
Basic plot example:

Basic plot example with a fit and an annotation (annotation is on by default):


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
erg / A s cm²
−50
−25
0
25
50
75
100
125
150
Velocity (km / s)


A(0)= 1.032 ± 0.041
Δx(0)= 49.87 ± 0.46
σ(0)= 9.94 ± 0.46
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
erg / A s cm²
−50
−25
0
25
50
75
100
125
150
Velocity (km / s)

Basic plot example with a fit, but with no annotation:



Basic plot example with a second spectrum overlaid in green:

Basic plot example with a second spectrum overlaid in green plus adjusted limits:

Basic plot example with a second spectrum offset and overlaid in red, again with adjusted limits:

3.1 API Documentation for Plotting

We include the API documentation for the generic model and fitter wrappers here.

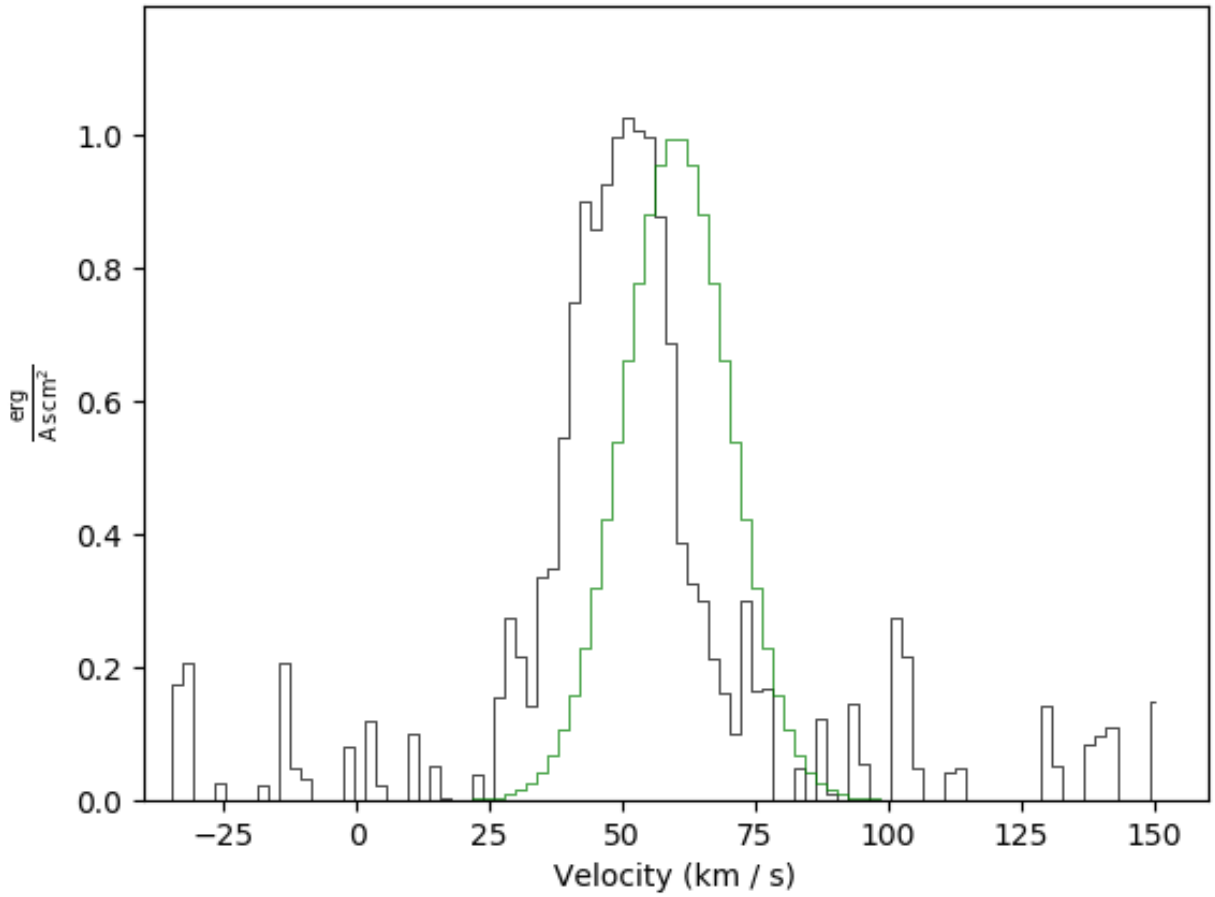
class pyspeckit.spectrum.plotters.**BoundMethodProxy**(*meth*, *callback=None*)

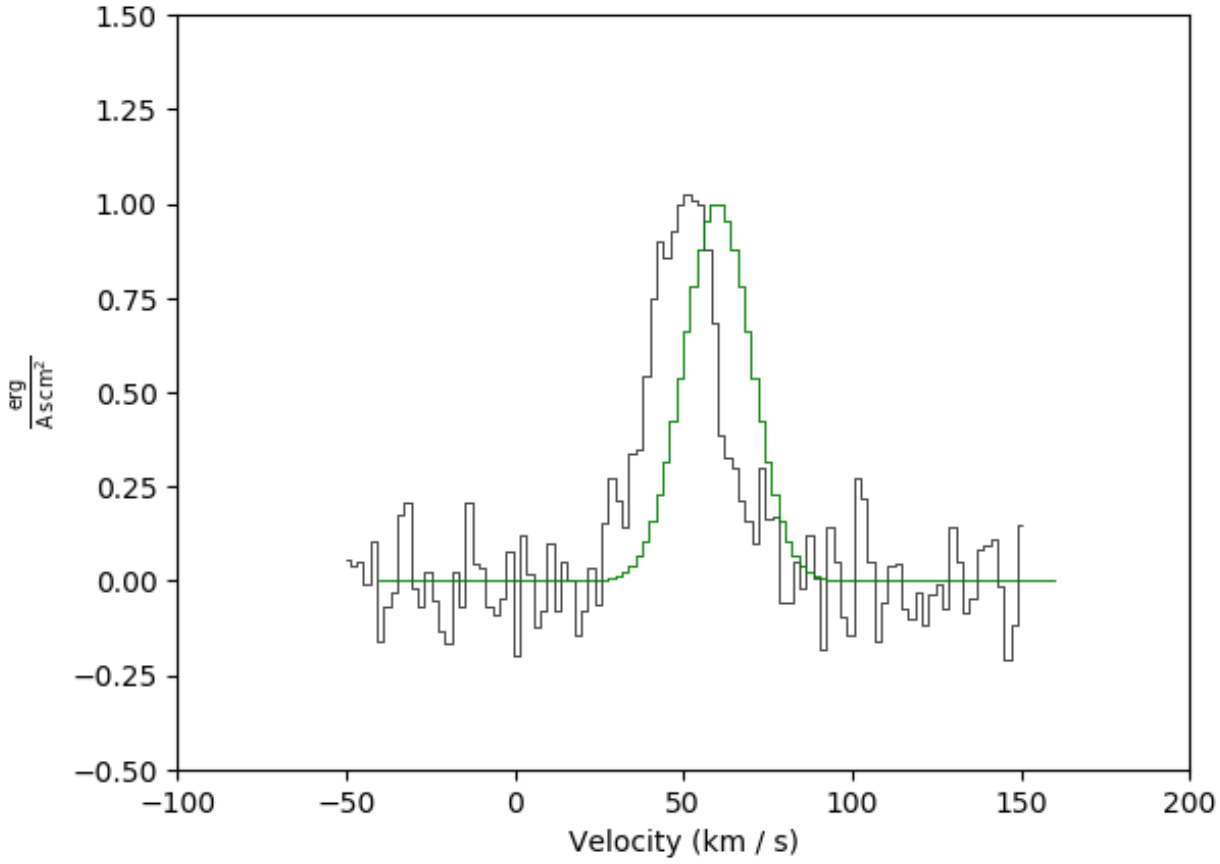
class pyspeckit.spectrum.plotters.**Plotter**(*Spectrum*, *autorefresh=True*, *title=''*, *xlabel=None*, *silent=True*, *plotscale=1.0*, ***kwargs*)

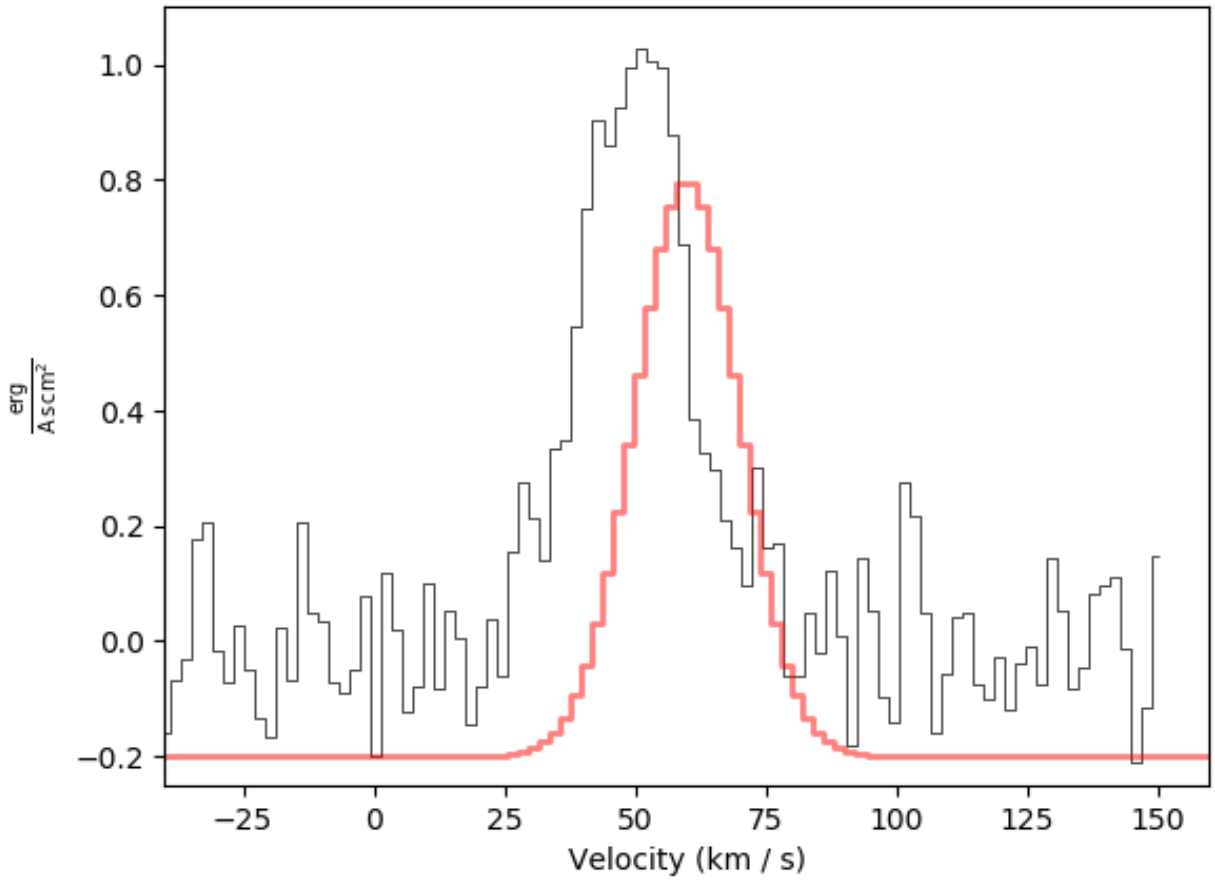
Class to plot a spectrum

activate_interactive_baseline_fitter(***kwargs*)

Attempt to activate the interactive baseline fitter


1.0
0.8
0.6
0.4
0.2
0.0
erg / Ascm²
−25
0
25
50
75
100
125
150
Velocity (km / s)


1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
erg / Ascm²
−100
−50
0
50
100
150
200
Velocity (km / s)


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
erg / Ascm²
−25
0
25
50
75
100
125
150
Velocity (km / s)

activate_interactive_fitter()

Attempt to activate the interactive fitter

connect()

Connect to the matplotlib key-parsing interactivity

copy(*parent=None*)

Create a copy of the plotter with blank (uninitialized) axis & figure

[parent]

A spectroscopic axis instance that is the parent of the specfit instance. This needs to be specified at some point, but defaults to None to prevent overwriting a previous plot.

disconnect()

Disconnect the matplotlib interactivity of this pyspeckit plotter.

label(*title=None*, *xlabel=None*, *ylabel=None*, *verbose_label=False*, ***kwargs*)

Label the plot, with an attempt to parse standard units into nice latex labels

Parameters

title : str

xlabel : str

ylabel : str

verbose_label: bool :

line_ids(*line_names*, *line_xvals*, *xval_units=None*, *auto_yloc=True*, *velocity_offset=None*, *velocity_convention='radio'*, *auto_yloc_fraction=0.9*, ***kwargs*)

Add line ID labels to a plot using lineid_plot http://oneau.wordpress.com/2011/10/01/line-id-plot/ https://github.com/phn/lineid_plot http://packages.python.org/lineid_plot/

Parameters

line_names : list

A list of strings to label the specified x-axis values

line_xvals : list

List of x-axis values (e.g., wavelengths) at which to label the lines. Can be a list of quantities.

xval_units : string

The unit of the line_xvals if they are not given as quantities

velocity_offset : quantity

A velocity offset to apply to the inputs if they are in frequency or wavelength units

velocity_convention : 'radio' or 'optical' or 'doppler'

Used if the velocity offset is given

auto_yloc : bool

If set, overrides box_loc and arrow_tip (the vertical position of the lineid labels) in kwargs to be `auto_yloc_fraction` of the plot range

auto_yloc_fraction: float in range [0,1] :

The fraction of the plot (vertically) at which to place labels

Examples

```
>>> import numpy as np
>>> import pyspeckit
>>> sp = pyspeckit.Spectrum(
        xarr=pyspeckit.units.SpectroscopicAxis(np.linspace(-50,50,101),
            unit='km/s', refX=6562.8, refX_unit='angstrom'),
        data=np.random.randn(101), error=np.ones(101))
>>> sp.plotter()
>>> sp.plotter.line_ids(['H$\alpha$'],[6562.8],xval_units='angstrom')
```

line_ids_from_measurements(*auto_yloc=True*, *auto_yloc_fraction=0.9*, ***kwargs*)

Add line ID labels to a plot using lineid_plot http://oneau.wordpress.com/2011/10/01/line-id-plot/ https://github.com/phn/lineid_plot http://packages.python.org/lineid_plot/

Parameters

auto_yloc : bool

If set, overrides box_loc and arrow_tip (the vertical position of the lineid labels) in kwargs to be `auto_yloc_fraction` of the plot range

auto_yloc_fraction: float in range [0,1] :

The fraction of the plot (vertically) at which to place labels

Examples

```
>>> import numpy as np
>>> import pyspeckit
>>> sp = pyspeckit.Spectrum(
        xarr=pyspeckit.units.SpectroscopicAxis(np.linspace(-50,50,101),
            units='km/s', refX=6562.8, refX_unit='angstroms'),
        data=np.random.randn(101), error=np.ones(101))
>>> sp.plotter()
>>> sp.specfit(multifit=None, fittype='gaussian', guesses=[1,0,1]) # fitting noise..
↪..
>>> sp.measure()
>>> sp.plotter.line_ids_from_measurements()
```

parse_keys(*event*)

Parse key commands entered from the keyboard

plot(*offset=0.0*, *xoffset=0.0*, *color='k'*, *drawstyle='steps-mid'*, *linewidth=0.5*, *errstyle=None*, *erralpha=0.2*, *errcolor=None*, *silent=None*, *reset=True*, *refresh=True*, *use_window_limits=None*, *useOffset=False*, ***kwargs*)

Plot the spectrum!

Tries to automatically find a reasonable plotting range if one is not set.

Parameters

offset : float

vertical offset to add to the spectrum before plotting. Useful if you want to overlay multiple spectra on a single plot

xoffset: float :

An x-axis shift. I don't know why you'd want this...

color : str

default to plotting spectrum in black

drawstyle : 'steps-mid' or str

'steps-mid' for histogram-style plotting. See matplotlib's plot for more information

linewidth : float

Line width in pixels. Narrow lines are helpful when histo-plotting

errstyle : 'fill', 'bars', or None

can be "fill", which draws partially transparent boxes around the data to show the error region, or "bars" which draws standard errorbars. None will display no errorbars

useOffset : bool

Use offset-style X/Y coordinates (e.g., 1 + 1.483e10)? Defaults to False because these are usually quite annoying.

xmin/xmax/ymin/ymax : float

override defaults for plot range. Once set, these parameters are sticky (i.e., replotting will use the same ranges). Passed to `reset_limits`

reset_[xy]limits : bool

Reset the limits to "sensible defaults". Passed to `reset_limits`

ypeakscale : float

Scale up the Y maximum value. Useful to keep the annotations away from the data. Passed to `reset_limits`

reset : bool

Reset the x/y axis limits? If set, `reset_limits` will be called.

reset_limits(*xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *reset_xlimits=True*, *reset_ylimits=True*, *ypeakscale=1.2*, *silent=None*, *use_window_limits=False*, ***kwargs*)

Automatically or manually reset the plot limits

savefig(*fname*, *bbox_inches='tight'*, ***kwargs*)

simple wrapper of maplotlib's savefig.

set_limits_from_visible_window(*debug=False*)

Hopefully self-descriptive: set the x and y limits from the currently visible window (use this if you use the pan/zoom tools or manually change the limits)

pyspeckit.spectrum.plotters.**parse_norm**(*norm*)

Expected format: norm = 10E15

pyspeckit.spectrum.plotters.**steppify**(*arr*, *isX=False*)

support function Converts an array to double-length for step plotting

CHAPTER

FOUR

FEATURES

4.1 Baseline Fitting

There are a number of cool features in baselining that aren't well-described below, partly due to Sphinx errors as of 12/22/2011.

`exclude` and `include` allow you to specify which parts of the spectrum to use for baseline fitting. Enter values as pairs of coordinates.

Excludefit makes use of an existing fit and excludes all points with signal above a (very low) threshold when fitting the baseline. Going back and forth between `baseline(excludefit=True)` and `specfit()` is a nice way to iteratively measure the baseline & emission/absorption line components.

4.1.1 API

class pyspeckit.spectrum.baseline.**Baseline**(*Spectrum*)

Class to measure and subtract baselines from spectra.

While the term 'baseline' is generally used in the radio to refer to broad-band features in a spectrum not necessarily associated with a source, in this package it refers to general continuum fitting. In principle, there's no reason to separate 'continuum' and 'spectral feature' fitting into different categories (both require some model, data, and optional weights when fitting). In practice, however, 'continuum' is frequently something to be removed and ignored, while spectral features are the desired measurable quantity. In order to accurately measure spectral features, it is necessary to allow baselines of varying complexity.

The Baseline class has both interactive and command-based data selection features. It can be used to fit both polynomial and power-law continua. Blackbody fitting is not yet implemented [12/21/2011]. Baseline fitting is a necessary prerequisite for Equivalent Width measurement.

As you may observe in the comments on this code, this has been one of the buggiest and least adequately tested components of pyspeckit. Bug reports are welcome. (as of 1/15/2012, a major change has probably fixed most of the bugs, and the code base is much simpler)

Declare interactive variables.

Must have a parent Spectrum class

Must declare button2action and button3action

__call__(**args*, ***kwargs*)

Fit and remove a polynomial from the spectrum. It will be saved in the variable "self.basespec" and the fit parameters will be saved in "self.order"

Parameters

order: int :

Order of the polynomial to fit

excludefit: bool :

If there is a spectroscopic line fit, you can automatically exclude the region with signal above some tolerance set by `exclusionlevel` (it works for absorption lines by using the absolute value of the signal)

exclusionlevel: float :

The minimum value of the spectroscopic fit to exclude when fitting the baseline

save: bool :

Write the baseline fit coefficients into the spectrum's header in the keywords BLCOEFnn

interactive: bool :

Specify the include/exclude regions through the interactive plot window

fit_original: bool :

Fit the original spectrum instead of the baseline-subtracted spectrum. If disabled, will overwrite the original data with the baseline-subtracted version.

Warning: If this is set False, behavior of `unsubtract` may be unexpected

fit_plotted_area: bool :

Will respect user-specified zoom (using the pan/zoom buttons) unless xmin/xmax have been set manually

reset_selection: bool :

Reset the selected region to those specified by this command only (will override previous xmin/xmax settings)

select_region: bool :

Run the region selection procedure? If false, will leave 'includemask' untouched

baseline_fit_color: color name (string) :

[plot parameter] Color to plot the baseline

clear_all_connections: bool :

[plot parameter] Disable any previous interactive sessions

highlight_fitregion: bool :

[plot parameter] Highlight the selected region for baseline fitting (default green)

`__init__`(*Spectrum*)

Declare interactive variables.

Must have a parent Spectrum class

Must declare button2action and button3action

__module__ = 'pyspeckit.spectrum.baseline'

annotate(*loc='upper left'*, *fontsize=10*)

button2action(*event=None*, *debug=False*, *subtract=True*, *powerlaw=None*, *fit_original=False*, *spline=False*, *spline_sampling=None*, *spline_downsampler=<function median>*, *baseline_fit_color='orange'*, ***kwargs*)

Do the baseline fitting and save and plot the results.

Note that powerlaw fitting will only consider positive data.

button3action(**args*, ***kwargs*)

Wrapper - same as button2action, but with subtract=False

clearlegend()

copy(*parent=None*)

Create a copy of the baseline fit

[parent]

A spectroscopic axis instance that is the parent of the specfit instance. This needs to be specified at some point, but defaults to None to prevent overwriting a previous plot.

crop(*x1pix*, *x2pix*)

When spectrum.crop is called, this must be too

downsample(*factor*)

fit(*powerlaw=None*, *order=None*, *includemask=None*, *spline=False*, *spline_sampling=10*, *spline_downsampler=<function median>*, *xarr_fit_unit='pixels'*, ***kwargs*)

Run the fit and set `self.basespec`

get_model(*xarr=None*, *baselinepars=None*)

plot_baseline(*annotate=True*, *baseline_fit_color='orange'*, *use_window_limits=None*, *linewidth=1*, *alpha=0.75*, *plotkwargs={}*, ***kwargs*)

Overplot the baseline fit

Parameters

annotate : bool

Display the fit parameters for the best-fit baseline on the top-left of the plot

baseline_fit_color : matplotlib color

What color to use for overplotting the line (default is slightly transparent orange)

use_window_limits : None or bool

Keep the current window or expand the plot limits? If left as None, will use `self.use_window_limits`

`savefit()`

`set_basespec_frompars`(*baselinepars=None*, *xarr_fit_unit=None*)

Set the baseline spectrum based on the fitted parameters

Parameters

baselinepars : list

Optional list of fit parameters, e.g. a list of polynomial coefficients

xarr_fit_unit : None or 'pixels' or 'native' or unit

The units that were used in the baseline fit

`set_spectofit`(*fit_original=True*, *fit_residuals=False*)

Reset the spectrum-to-fit from the data

`unsubtract`(*replot=True*, *preserve_limits=True*)

Restore the spectrum to "pristine" state (un-subtract the baseline)

replot **[True]**

Re-plot the spectrum? (only happens if unsubtraction proceeds, i.e. if there was a baseline to unsubtract)

preserve_limits **[True]**

Preserve the current x,y limits

4.2 Model Fitting

General documentation for model fitting is below. There are several multi-component model fitting tools that have been developed based on pyspeckit:

- Mike Chen's multi-vlsr
- Vlas Sokolov's pyspecnest and multicube
- Jonny Henshaw's SCOUSE
- Jared Keown's astroclover

class pyspeckit.spectrum.fitters.**Specfit**(*Spectrum*, *Registry=None*)

Bases: Interactive

Declare interactive variables.

Must have a parent Spectrum class

Must declare button2action and button3action

EQW(*plot=False*, *plotcolor='g'*, *fitted=True*, *continuum=None*, *components=False*, *annotate=False*, *alpha=0.5*, *loc='lower left'*, *xmin=None*, *xmax=None*, *xunits=None*, *continuum_as_baseline=False*, *xunit='pixel'*, *midpt_location='plot-center'*)

Returns the equivalent width (integral of "baseline" or "continuum" minus the spectrum) over the selected range (the selected range defaults to self.xmin:self.xmax, so it may include multiple lines!)

Parameters

plot : bool

Plots a box indicating the EQW if plot==True (i.e., it will have a width equal to the equivalent width, and a height equal to the measured continuum)

fitted : bool

Use the fitted model? If false, uses the data

continuum : None or float

Can specify a fixed continuum with this keyword, otherwise will use the fitted baseline. WARNING: continuum=0 will still "work", but will give numerically invalid results. Similarly, a negative continuum will work, but will yield results with questionable physical meaning.

continuum_as_baseline : bool

Replace the baseline with the specified continuum when computing the absorption depth of the line

components : bool

If your fit is multi-component, will attempt to acquire centroids for each component and print out individual EQWs

xmin : float

xmax : float

The range over which to compute the EQW

xunit : str

The units of xmin/xmax

midpt_location : 'fitted', 'plot-center'

If 'plot' is set, this determines where the EQW will be drawn. It can be the fitted centroid or the plot-center, i.e. (xmin+xmax)/2

Returns

Equivalent Width, or widths if components=True :

add_sliders(*parlimitdict=None*, ***kwargs*)

Add a Sliders window in a new figure appropriately titled

Parameters

parlimitdict: dict :

Each parameter needs to have displayed limits; these are set in min-max pairs. If this is left empty, the widget will try to guess at reasonable limits, but the guessing is not very sophisticated yet.

.. todo:: Add a button in the navbar that makes this window pop up :

http://stackoverflow.com/questions/4740988/add-new-navigate-modes-in-matplotlib :

annotate(*loc='upper right'*, *labelspacing=0.25*, *markerscale=0.01*, *borderpad=0.1*, *handlelength=0.1*, *handletextpad=0.1*, *fontsize=10*, *frameon=False*, *chi2=None*, *optimal_chi2_kwargs={}*, ***kwargs*)

Add a legend to the plot showing the fitted parameters

_clearlegend() will remove the legend

chi2 : {True or 'reduced' or 'optimal' or 'allthree'}

kwargs passed to legend

button3action(*event*, *debug=False*, *nwidths=1*)

Disconnect the interactiveness Perform the fit (or die trying) Hide the guesses

clear(*legend=True*, *components=True*)

Remove the fitted model from the plot

Also removes the legend by default

clear_all_connections(*debug=False*)

Prevent overlapping interactive sessions

clear_highlights()

Hide and remove "highlight" colors from the plot indicating the selected region

copy(*parent=None*, *registry=None*)

Create a copy of the spectral fit - includes copies of the _full_model, the registry, the fitter, parinfo, modelpars, modelerrs, model, npeaks

Parameters

parent : `pyspeckit.classes.Spectrum`

A `Spectrum` instance that is the parent of the specfit instance. This needs to be specified at some point, but defaults to None to prevent overwriting a previous plot.

crop(*x1pix*, *x2pix*)

When spectrum.crop is called, this must be too

property dof

degrees of freedom in fit

downsample(*factor*)

Downsample the model spectrum (and the spectofit spectra) This should only be done when Spectrum.smooth is called

event_manager(*event*, *force_over_toolbar=False*, *debug=False*)

Decide what to do given input (click, keypress, etc.)

firstclick_guess()

Initialize self.guesses

property fitter

fullsizemodel()

If the model was fit to a sub-region of the spectrum, expand it (with zeros wherever the model was not defined) to fill the spectrum.

Examples

```
>>> noise = np.random.randn(100)
>>> xarr = np.linspace(-50,50,100)
>>> signal = np.exp(-(xarr-5)**2/(2*3.**2))
>>> sp = pyspeckit.Spectrum(data=noise + signal, xarr=xarr, xarrkwargs={'units':'km/
↪s'})
>>> sp.specfit(xmin=-25,xmax=25)
>>> sp.specfit.model.shape
(48,)
>>> sp.specfit.fullsizemodel()
>>> sp.specfit.model.shape
(100,)
```

get_components(***kwargs*)

If a model has been fitted, return the components of the model

Parameters

kwargs are passed to self.fitter.components :

get_emcee(*nwalkers=None*, ***kwargs*)

Get an emcee walker ensemble for the data & model using the current model type

Parameters

data : np.ndarray

error : np.ndarray

nwalkers : int

Number of walkers to use. Defaults to 2 * self.fitters.npars

Examples

```
>>> import pyspeckit
>>> x = pyspeckit.units.SpectroscopicAxis(np.linspace(-10,10,50), unit='km/s')
>>> e = np.random.randn(50)
>>> d = np.exp(-np.asarray(x)**2/2.)*5 + e
>>> sp = pyspeckit.Spectrum(data=d, xarr=x, error=np.ones(50)*e.std())
>>> sp.specfit(fittype='gaussian')
>>> emcee_ensemble = sp.specfit.get_emcee()
>>> p0 = emcee_ensemble.p0 * (np.random.randn(*emcee_ensemble.p0.shape) / 10. + 1.0)
>>> pos,logprob,state = emcee_ensemble.run_mcmc(p0,100)
```

get_full_model(*debug=False*, ***kwargs*)

compute the model over the full axis

get_model(*xarr*, *pars=None*, *debug=False*, *add_baseline=None*)

Compute the model over a given axis

get_model_frompars(*xarr*, *pars*, *debug=False*, *add_baseline=None*)

Compute the model over a given axis

get_model_xlimits(*threshold='auto'*, *peak_fraction=0.01*, *add_baseline=False*, *unit='pixels'*, *units=None*)

Return the x positions of the first and last points at which the model is above some threshold

Parameters

threshold : 'auto' or 'error' or float

If 'auto', the threshold will be set to peak_fraction * the peak model value. If 'error', uses the error spectrum as the threshold

peak_fraction : float

ignored unless threshold == 'auto'

add_baseline : bool

Include the baseline when computing whether the model is above the threshold? default FALSE. Passed to get_full_model.

units : str

A valid unit type, e.g. 'pixels' or 'angstroms'

get_pymc(***kwargs*)

Create a pymc MCMC sampler from the current fitter. Defaults to 'uninformative' priors

`kwargs` are passed to the fitter's get_pymc method, with parameters defined below.

Parameters

data : np.ndarray

error : np.ndarray

use_fitted_values : bool

Each parameter with a measured error will have a prior defined by the Normal distribution with sigma = par.error and mean = par.value

Examples

```
>>> x = pyspeckit.units.SpectroscopicAxis(np.linspace(-10,10,50), unit='km/s')
>>> e = np.random.randn(50)
>>> d = np.exp(-np.asarray(x)**2/2.)*5 + e
>>> sp = pyspeckit.Spectrum(data=d, xarr=x, error=np.ones(50)*e.std())
>>> sp.specfit(fittype='gaussian')
>>> MCuninformed = sp.specfit.get_pymc()
>>> MCwithpriors = sp.specfit.get_pymc(use_fitted_values=True)
>>> MCuninformed.sample(1000)
>>> MCuninformed.stats()['AMPLITUDE0']
>>> # WARNING: This will fail because width cannot be set <0, but it may randomly␣
↪reach that...
>>> # How do you define a likelihood distribution with a lower limit?!
>>> MCwithpriors.sample(1000)
>>> MCwithpriors.stats()['AMPLITUDE0']
```

guesspeakwidth(*event*, *debug=False*, *nwidths=1*, ***kwargs*)

Interactively guess the peak height and width from user input

Width is assumed to be half-width-half-max

highlight_fitregion(*drawstyle='steps-mid'*, *color=(0, 0.8, 0, 0.5)*, *linewidth=2*, *alpha=0.5*, *clear_highlights=True*, ***kwargs*)

Re-highlight the fitted region

kwargs are passed to `matplotlib.plot`

history_fitpars()

integral(*analytic=False*, *direct=False*, *threshold='auto'*, *integration_limits=None*, *integration_limit_units='pixels'*, *return_error=False*, ***kwargs*)

Return the integral of the fitted spectrum

Parameters

analytic : bool

Return the analytic integral of the fitted function? .. WARNING:: This approach is only implemented for some models .. todo:: Implement error propagation for this approach

direct : bool

Return the integral of the *spectrum* (as opposed to the *fit*) over a range defined by the `integration_limits` if specified or `threshold` otherwise

threshold : 'auto' or 'error' or float

Determines what data to be included in the integral based off of where the model is greater than this number If 'auto', the threshold will be set to peak_fraction * the peak model value. If 'error', uses the error spectrum as the threshold See `self.get_model_xlimits` for details

integration_limits : None or 2-tuple

Manually specify the limits in `integration_limit_units` units

return_error : bool

Return the error on the integral if set. The error computed by sigma = sqrt(sum(sigma_i^2)) * dx

kwargs : :

passed to `self.fitter.integral` if `not(direct)`

Returns

np.scalar or np.ndarray with the integral or integral & error :

property mask

Mask: True means "exclude"

property mask_sliced

Sliced (subset) Mask: True means "exclude"

measure_approximate_fwhm(*threshold='error'*, *emission=True*, *interpolate_factor=1*, *plot=False*, *grow_threshold=2*, ***kwargs*)

Measure the FWHM of a fitted line

This procedure is designed for multi-component *blended* lines; if the true FWHM is known (i.e., the line is well-represented by a single gauss/voigt/lorentz profile), use that instead. Do not use this for multiple independently peaked profiles.

This MUST be run AFTER a fit has been performed!

Parameters

threshold : 'error' | float

The threshold above which the spectrum will be interpreted as part of the line. This threshold is applied to the *model*. If it is 'noise', self.error will be used.

emission : bool

Is the line absorption or emission?

interpolate_factor : integer

Magnification factor for determining sub-pixel FWHM. If used, "zooms-in" by using linear interpolation within the line region

plot : bool

Overplot a line at the FWHM indicating the FWHM. kwargs are passed to matplotlib.plot

grow_threshold : int

Minimum number of valid points. If the total # of points above the threshold is <= to this number, it will be grown by 1 pixel on each side

Returns

The approximated FWHM, if it can be computed :

If there are <= 2 valid pixels, a fwhm cannot be computed :

model_mask(***kwargs*)

Get a mask (boolean array) of the region where the fitted model is significant

Parameters

threshold : 'auto' or 'error' or float

The threshold to compare the model values to for selecting the mask region.

- auto: uses `peak_fraction` times the model peak
- error: use the spectrum error
- float: any floating point number as an absolute threshold

peak_fraction : float

Parameter used if `threshold=='auto'` to determine fraction of model peak to set threshold at

add_baseline : bool

Add the fitted baseline to the model before comparing to threshold?

Returns

mask : `ndarray`

A boolean mask array with the same size as the spectrum, set to `True` where the fitted model has values above a specified threshold

moments(*fittype=None*, ***kwargs*)

Return the moments

see the `moments` module

Parameters

fittype : None or str

The registered fit type to use for moment computation

multifit(*fittype=None*, *renormalize='auto'*, *annotate=None*, *show_components=None*, *verbose=True*, *color=None*, *guesses=None*, *parinfo=None*, *reset_fitspec=True*, *use_window_limits=None*, *use_lmfit=False*, *plot=True*, ***kwargs*)

Fit multiple gaussians (or other profiles)

Parameters

fittype : str

What function will be fit? fittype must have been Registryed in the peakbgfitters dict. Uses default ('gaussian') if not specified

renormalize : 'auto' or bool

if 'auto' or True, will attempt to rescale small data (<1e-9) to be closer to 1 (scales by the median) so that the fit converges better

parinfo : `parinfo` structure

Guess structure; supercedes `guesses`

guesses : list or 'moments'

Either a list of guesses matching the number of parameters * the number of peaks for the model, or 'moments' to fit a single spectrum with the moments as guesses

optimal_chi2(*reduced=True*, *threshold='error'*, ***kwargs*)

Compute an "optimal" χ^2 statistic, i.e. one in which only pixels in which the model is statistically significant are included

Parameters

reduced : bool

Return the reduced χ^2

threshold : 'auto' or 'error' or float

If 'auto', the threshold will be set to peak_fraction * the peak model value, where peak_fraction is a kwarg passed to get_model_xlimits reflecting the fraction of the model peak to consider significant If 'error', uses the error spectrum as the threshold

kwargs : dict

passed to `get_model_xlimits()`

Returns

chi2 : float

χ^2 statistic or reduced χ^2 statistic (χ^2/n)

$$\chi^2 = \sum((d_i - m_i)^2/e_i^2)$$

peakbgfit(*usemoments=True*, *annotate=None*, *vheight=True*, *height=0*, *negamp=None*, *fittype=None*, *renormalize='auto'*, *color=None*, *use_lmfit=False*, *show_components=None*, *debug=False*, *use_window_limits=True*, *guesses=None*, *nsigcut_moments=None*, *plot=True*, *parinfo=None*, ***kwargs*)

Fit a single peak (plus a background)

Parameters

usemoments : bool

The initial guess will be set by the fitter's 'moments' function (this overrides 'guesses')

annotate : bool

Make a legend?

vheight : bool

Fit a (constant) background as well as a peak?

height : float

initial guess for background

negamp : bool

If True, assumes amplitude is negative. If False, assumes positive. If None, can be either.

fittype : bool

What function will be fit? fittype must have been Registryed in the peakbgfitters dict

renormalize : 'auto' or bool

if 'auto' or True, will attempt to rescale small data (<1e-9) to be closer to 1 (scales by the median) so that the fit converges better

nsigcut_moments : bool

pass to moment guesser; can do a sigma cut for moment guessing

`plot_components`(*xarr=None*, *show_hyperfine_components=None*, *component_yoffset=0.0*, *component_lw=0.75*, *pars=None*, *component_fit_color='blue'*, *component_kwargs={}*, *add_baseline=False*, *plotkwargs={}*, ***kwargs*)

Overplot the individual components of a fit

Parameters

xarr : None

If none, will use the spectrum's xarr. Otherwise, plot the specified xarr. This is useful if you want to plot a well-sampled model when the input spectrum is undersampled

show_hyperfine_components : None | bool

Keyword argument to pass to component codes; determines whether to return individual (e.g., hyperfine) components of a composite model

component_yoffset : float

Vertical (y-direction) offset to add to the components when plotting

component_lw : float

Line width of component lines

component_fitcolor : color

Color of component lines

component_kwargs : dict

Keyword arguments to pass to the fitter.components method

add_baseline : bool

Add the fit to the components before plotting. Makes sense to use if self.Spectrum.baseline.subtracted == False

pars : parinfo

A parinfo structure or list of model parameters. If none, uses best-fit

plot_fit(*xarr=None*, *annotate=None*, *show_components=None*, *composite_fit_color='red'*, *lw=0.5*, *composite_lw=0.75*, *pars=None*, *offset=None*, *use_window_limits=None*, *show_hyperfine_components=None*, *plotkwargs={}*, ***kwargs*)

Plot the fit. Must have fitted something before calling this!

It will be automatically called whenever a spectrum is fit (assuming an axis for plotting exists)

kwargs are passed to the fitter's components attribute

Parameters

xarr : None

If none, will use the spectrum's xarr. Otherwise, plot the specified xarr. This is useful if you want to plot a well-sampled model when the input spectrum is undersampled

annotate : None or bool

Annotate the plot? If not specified, defaults to self.autoannotate

show_components : None or bool

show_hyperfine_components : None or bool

Show the individual gaussian components overlaid on the composite fit

use_window_limits : None or bool

If False, will reset the window to include the whole spectrum. If True, leaves the window as is. Defaults to self.use_window_limits if None.

pars : parinfo

A parinfo structure or list of model parameters. If none, uses best-fit

offset : None or float

Y-offset. If none, uses the default self.Spectrum.plotter offset, otherwise, uses the specified float.

plot_model(*pars*, *offset=0.0*, *annotate=False*, *clear=False*, ***kwargs*)

Plot a model from specified input parameters (see plot_fit for kwarg specification)

annotate is set to "false" because arbitrary annotations are not yet implemented

plotresiduals(*fig=2*, *axis=None*, *clear=True*, *color='k'*, *linewidth=0.5*, *drawstyle='steps-mid'*, *yoffset=0.0*, *label=True*, *pars=None*, *zeroline=None*, *set_limits=True*, ***kwargs*)

Plot residuals of the fit. Specify a figure or axis; defaults to figure(2).

Parameters

fig : int

Figure number. Overridden by axis

axis : axis

The axis to plot on

pars : None or parlist

If set, the residuals will be computed for the input parameters

zeroline : bool or None

Plot the "zero" line through the center of the residuals. If None, defaults to "True if yoffset!=0, False otherwise"

kwargs are passed to matplotlib plot :

print_fit(*print_baseline=True*, ***kwargs*)

Print the best-fit parameters to the command line

refit(*use_lmfit=False*)

Redo a fit using the current parinfo as input

register_fitter(**args*, ***kwargs*)

Register a model fitter

Register a fitter function.

Parameters

name: string :

The fit function name.

function: function :

The fitter function. Single-fitters should take npars + 1 input parameters, where the +1 is for a 0th order baseline fit. They should accept an X-axis and data and standard fitting-function inputs (see, e.g., gaussfitter). Multi-fitters should take N * npars, but should also operate on X-axis and data arguments.

npars: int :

How many parameters does the function being fit accept?

savefit()

Save the fit parameters from a Gaussian fit to the FITS header

selectregion(*xmin=None*, *xmax=None*, *xtype='wcs'*, *highlight=False*, *fit_plotted_area=True*, *reset=False*, *verbose=False*, *debug=False*, *use_window_limits=None*, *exclude=None*, ***kwargs*)

Pick a fitting region in either WCS units or pixel units

Parameters

xmin / xmax : [float]

The min/max X values to use in X-axis units (or pixel units if xtype is set). TAKES PRECEDENCE ALL OTHER BOOLEAN OPTIONS

xtype : [string]

A string specifying the xtype that xmin/xmax are specified in. It can be either 'wcs' or any valid xtype from `pyspeckit.spectrum.units`

reset : [bool]

Reset the selected region to the full spectrum? Only takes effect if xmin and xmax are not (both) specified. TAKES PRECEDENCE ALL SUBSEQUENT BOOLEAN OPTIONS

fit_plotted_area : [bool]

Use the plot limits *as specified in :class:`pyspeckit.spectrum.plotters`*? Note that this is not necessarily the same as the window plot limits!

use_window_limits : [bool]

Use the plot limits *as displayed*. Defaults to self.use_window_limits (`pyspeckit.spectrum.interactive.use_window_limits`). Overwrites xmin,xmax set by plotter

exclude: {list of length 2n,'interactive', None} :

- interactive: start an interactive session to select the include/exclude regions
- list: parsed as a series of (startpoint, endpoint) in the spectrum's X-axis units. Will exclude the regions between startpoint and endpoint
- None: No exclusion

seterrspec(*usestd=None*, *useresiduals=True*)

Simple wrapper function to set the error spectrum; will either use the input spectrum or determine the error using the RMS of the residuals, depending on whether the residuals exist.

setfitspec()

Set the spectrum that will be fit. This is primarily to remove NANs from consideration: if you simply remove the data from both the X-axis and the Y-axis, it will not be considered for the fit, and a linear X-axis is not needed for fitting.

However, it may be possible to do this using masked arrays instead of setting errors to be 1e10....

shift_pars(*frame=None*)

Shift the velocity / wavelength / frequency of the fitted parameters into a different frame

Right now this only takes care of redshift and only if redshift is defined. It should be extended to do other things later

start_interactive(*debug=False*, *LoudDebug=False*, *reset_selection=False*, *print_message=True*, *clear_all_connections=True*, ***kwargs*)

Initialize the interative session

Parameters

print_message : bool

Print the interactive help message?

clear_all_connections : bool

Clear all matplotlib event connections? (calls `self.clear_all_connections()`)

reset_selection : bool

Reset the include mask to be empty, so that you're setting up a fresh region.

property xmax

property xmin

4.2.1 Fitting a user-defined model to a spectrum

damped_lya_profile is the function that generates the model and damped_lya_fitter creates the fitter class.

```
def damped_lya_profile(wave_to_fit,vs_n,am_n,fw_n,vs_b,am_b,fw_b,h1_col,h1_b,
                       h1_vel,d2h,resolution,single_component_flux=False,
                       return_components=False,
                       return_hyperfine_components=False):

    """
    Computes a damped Lyman-alpha profile (by calling the functions
    lya_intrinsic_profile_func and total_tau_profile_func) and convolves it
    to the proper resolution.

    """

    lya_intrinsic_profile = lya_intrinsic_profile_func(wave_to_fit,vs_n,am_n,fw_n,
                              vs_b,am_b,fw_b,single_component_flux=single_component_flux)
    total_tau_profile = total_tau_profile_func(wave_to_fit,h1_col,h1_b,h1_vel,d2h)

    lya_obs_high = lya_intrinsic_profile * total_tau_profile

    ## Convolving the data ##
    fw = lya_rest/resolution
    aa = make_kernel(grid=wave_to_fit,fwhm=fw)
    lyman_fit = np.convolve(lya_obs_high,aa,mode='same')

    return lyman_fit*1e14

def damped_lya_fitter(multisingle='multi'):

  """
  Generator for Damped LyA fitter class

  """
  myclass =  pyspeckit.models.model.SpectralModel(damped_lya_profile,
          11,
          parnames=['vs_n','am_n','fw_n','vs_b','am_b','fw_b','h1_col',
                    'h1_b','h1_vel','d2h','resolution'],
          parlimited=[(False,False),(True,False),(True,False),(False,False),
                      (True,False),(True,False),(True,True),(True,True),
                      (False,False),(True,True),(True,False)],
          parlimits=[(14.4,14.6), (1e-16,0), (50.,0),(0,0), (1e-16,0),(50.,0),
                     (17.0,19.5), (5.,20.), (0,0),(1.0e-5,2.5e-5),(10000.,0)])
  myclass.__name__ = "damped_lya"

  return myclass

## Load the spectrum, register the fitter, and fit
spec = pyspeckit.Spectrum(xarr=wave_to_fit, data=flux_to_fit*1e14,
                          error=error_to_fit*1e14, doplot=False,
                          header=spec_header)
```

(continues on next page)

(continued from previous page)

```
spec.Registry.add_fitter('damped_lya',damped_lya_fitter(),11)

spec.specfit(fittype='damped_lya',
             guesses=[vs_n, am_n, fw_n, vs_b, am_b, fw_b, h1_col, h1_b,
                      h1_vel, d2h, resolution],
             quiet=False, fixed=[False, False, False, False, False,
                                 False, False, False, False, True, True])

## Check out the results
spec.specfit.parinfo.values
spec.specfit.parinfo.errors
reduced_chi2 = spec.specfit.chi2/spec.specfit.dof
```

4.3 Measurements

class pyspeckit.spectrum.measurements.**Measurements**(*Spectrum*, *z=None*, *d=None*, *fluxnorm=None*, *miscline=None*, *misctol=10.0*, *ignore=None*, *derive=True*, *debug=False*, *restframe=False*, *ptol=2*, *sort=False*)

Bases: object

This can be called after a fit is run. It will inherit the specfit object and derive as much as it can from modelpars. Just do: spec.measure(z, xunits, fluxnorm)

Notes: If z (redshift) or d (distance) are present, we can compute integrated line luminosities rather than just fluxes. Provide distance in cm.

Only works with Gaussians. To generalize:

1. make sure we manipulate modelpars correctly, i.e. read in entries corresponding to wavelength/frequency/whatever correctly.

Parameters

z: float or None :

redshift

d: float or None :

distance in cm (used for luminosities)

fluxnorm: bool :

Normalize the fluxes?

miscline: dictionary :

miscline = [{'name': H_alpha', 'wavelength': 6565}]

misctol: tolerance (in Angstroms) for identifying an unmatched line :

to the line(s) we specify in miscline dictionary.

sort: bool :

Sort the entries in order of observed wavelength (or velocity or frequency)

bisection(*f*, *x_guess*)

Find root of function using bisection method. Absolute tolerance of 1e-4 is being used.

bracket_root(*f*, *x_guess*, *atol=0.0001*)

Bracket root by finding points where function goes from positive to negative.

compute_amplitude(*pars*)

Calculate amplitude of emission line. Should be easy - add multiple components if they exist. Currently assumes multiple components have the same centroid.

compute_flux(*pars*)

Calculate integrated flux of emission line. Works for multi-component fits too. Unnormalized.

compute_fwhm(*pars*)

Determine full-width at half maximum for multi-component fit numerically, or analytically if line has only a single component. Uses bisection technique for the former with absolute tolerance of 1e-4.

compute_luminosity(*pars*)

Determine luminosity of line (need distance and flux units).

derive()

Calculate luminosity and FWHM for all spectral lines.

identify_by_position(*ptol*)

Match observed lines to nearest reference line. Don't use spacing at all.

ptol = tolerance (in angstroms) to accept positional match

identify_by_spacing()

Determine identity of lines in self.modelpars. Fill entries of self.lines dictionary.

Note: This method will be infinitely slow for more than 10 or so lines.

separate()

For multicomponent lines, separate into broad and narrow components (assume only one of components is narrow).

to_tex()

Write out fit results to tex format.

4.4 Units

Unit parsing and conversion tool. The SpectroscopicAxis class is meant to deal with unit conversion internally

Open Questions: Are there other FITS-valid projection types, unit types, etc. that should be included? What about for other fields (e.g., wavenumber?)

class pyspeckit.spectrum.units.**SpectroscopicAxis**(*xarr, unit=None, refX=None, redshift=None, refX_unit=None, velocity_convention=None, use128bits=False, bad_unit_response='raise', equivalencies=[(Unit("m"), Unit("Hz"), <function spectral.<locals>.<lambda>>), (Unit("m"), Unit("J"), <function spectral.<locals>.<lambda>>), (Unit("Hz"), Unit("J"), <function spectral.<locals>.<lambda>>, <function spectral.<locals>.<lambda>>), (Unit("m"), Unit("1 / m"), <function spectral.<locals>.<lambda>>), (Unit("Hz"), Unit("1 / m"), <function spectral.<locals>.<lambda>>, <function spectral.<locals>.<lambda>>), (Unit("J"), Unit("1 / m"), <function spectral.<locals>.<lambda>>, <function spectral.<locals>.<lambda>>), (Unit("1 / m"), Unit("rad / m"), <function spectral.<locals>.<lambda>>, <function spectral.<locals>.<lambda>>), (Unit("m"), Unit("rad / m"), <function spectral.<locals>.<lambda>>), (Unit("Hz"), Unit("rad / m"), <function spectral.<locals>.<lambda>>, <function spectral.<locals>.<lambda>>), (Unit("J"), Unit("rad / m"), <function spectral.<locals>.<lambda>>, <function spectral.<locals>.<lambda>>)], center_frequency=None, center_frequency_unit=None*)

Bases: Quantity

A Spectroscopic Axis object to store the current units of the spectrum and allow conversion to other units and frames. Typically, units are velocity, wavelength, frequency, or redshift. Wavenumber is also hypothetically possible.

WARNING: If you index a SpectroscopicAxis, the resulting array will be a SpectroscopicAxis without a dxarr attribute! This can result in major problems; a workaround is being sought but subclassing numpy arrays is harder than I thought

Make a new spectroscopic axis instance Default units Hz

as_unit(*unit*, *equivalencies=[]*, *velocity_convention=None*, *refX=None*, *refX_unit=None*, *center_frequency=None*, *center_frequency_unit=None*, ***kwargs*)

Convert the spectrum to the specified units. This is a wrapper function to convert between frequency/velocity/wavelength and simply change the units of the X axis. Frame conversion is... not necessarily implemented.

cdelt(*tolerance=1e-08*, *approx=False*)

Return the channel spacing if channels are linear

convert_to_unit(*unit*, *make_dxarr=True*, ***kwargs*)

Return the X-array in the specified units without changing it Uses as_unit for the conversion, but changes internal values rather than returning them.

coord_to_x(*xval*, *xunit*)

Given an X-value assumed to be in the coordinate axes, return that value converted to xunit e.g.: xarr.unit = 'km/s' xarr.refX = 5.0 xarr.refX_unit = GHz xarr.coord_to_x(6000,'GHz') == 5.1 # GHz

property dxarr

classmethod find_equivalencies(*velocity_convention=None*, *center_frequency=None*, *equivalencies=[]*)

Utility function to add equivalencies from the velocity_convention and the center_frequency

in_range(*xval*)

Given an X coordinate in SpectroscopicAxis' units, return whether the pixel is in range

make_dxarr(*coordinate_location='center'*)

Create a "delta-x" array corresponding to the X array. It will have the same length as the input array, which is achieved by concatenating an extra pixel somewhere.

property refX

property refX_unit

property refX_units

set_unit(*unit*, *bad_unit_response='raise'*)

umax(*unit=None*)

Return the maximum value of the SpectroscopicAxis. If units specified, convert to those units first

umin(*unit=None*)

Return the minimum value of the SpectroscopicAxis. If units specified, convert to those units first

property units

classmethod validate_unit(*unit*, *bad_unit_response='raise'*)

x_to_coord(*xval*, *xunit*, *verbose=False*)

Given a wavelength/frequency/velocity, return the value in the SpectroscopicAxis's units e.g.: xarr.unit = 'km/s' xarr.refX = 5.0 xarr.refX_unit = GHz xarr.x_to_coord(5.1,'GHz') == 6000 # km/s

x_to_pix(*xval*, *xval_unit=None*, *xval_units=None*, *equivalencies=None*)

Given an X coordinate in SpectroscopicAxis' units, return the corresponding pixel number

class pyspeckit.spectrum.units.**SpectroscopicAxes**(*axislist*, *frame='rest'*, *xtype=None*, *refX=None*, *redshift=None*)

Bases: SpectroscopicAxis

Counterpart to Spectra: takes a list of SpectroscopicAxis's and concatenates them while checking for consistency and maintaining header parameter

Make a new spectroscopic axis instance Default units Hz

4.5 Registration

PySpecKit is made extensible by allowing user-registered modules for reading, writing, and fitting data.

For examples of registration in use, look at the source code of `pyspeckit.spectrum.__init__` and `pyspeckit.spectrum.fitters`.

The registration functions can be accessed directly:

```
pyspeckit.register_reader
pyspeckit.register_writer
```

However, models are bound to individual instances of the Spectrum class, so they must be accessed via a specfit instance

```
sp = pyspeckit.Spectrum('myfile.fits')
sp.specfit.register_fitter
```

Alternatively, you can access and edit the default Registry

```
pyspeckit.fitters.default_Registry.add_fitter
```

If you've already loaded a Spectrum instance, but then you want to reload fitters from the default_Registry, or if you want to make your own `Registry`, you can use the semi-private method

```
MyRegistry = pyspeckit.fitters.Registry()
sp._register_fitters(registry=MyRegistry)
```

4.6 Examples

If you want to register a new variable-optical-depth deuterated ammonia model, you could do the following:

sp.specfit.register_fitter(name='nh2d', function=nh2d.nh2d_vtau_fitter, npars=4)

4.6.1 API

pyspeckit.spectrum.__init__.**register_reader**(*filetype*, *function*, *suffix*, *default=False*)

Register a reader function.

Parameters

filetype: str :

The file type name

function: function :

The reader function. Should take a filename as input and return an X-axis object (see units.py), a spectrum, an error spectrum (initialize it to 0's if empty), and a pyfits header instance

suffix: int :

What suffix should the file have?

pyspeckit.spectrum.__init__.**register_writer**(*filetype*, *function*, *suffix*, *default=False*)

Register a writer function.

Parameters

filetype: string :

The file type name

function: function :

The writer function. Will be an attribute of Spectrum object, and called as spectrum.Spectrum.write_hdf5(), for example.

suffix: int :

What suffix should the file have?

class pyspeckit.spectrum.fitters.**Registry**

This class is a simple wrapper to prevent fitter properties from being globals

add_fitter(*name*, *function*, *npars*, *override=False*, *key=None*, *multisingle=None*)

Register a fitter function.

Parameters

name: string :

The fit function name.

function: function :

The fitter function. Single-fitters should take npars + 1 input parameters, where the +1 is for a 0th order baseline fit. They should accept an X-axis and data and standard fitting-function inputs (see, e.g., gaussfitter). Multi-fitters should take N * npars, but should also operate on X-axis and data arguments.

npars: int :

How many parameters does the function being fit accept?

4.7 Classes

4.7.1 spectrum

The spectrum module consists of the Spectrum class, with child classes ObsBlock and Spectra for multi-spectrum analysis of different types.

The Spectrum class is the main functional code. ObsBlocks are containers of multiple spectra of different objects The Spectra class is a container of multiple spectra of the *same* object at different wavelengths/frequencies

class pyspeckit.spectrum.classes.**Spectrum**(*filename_or_magic=None*, *filetype=None*, *xarr=None*, *data=None*, *error=None*, *header=None*, *doplot=False*, *maskdata=True*, *unit=None*, *plotkwargs={}*, *xarrkwargs={}*, *model_registry=None*, *filename=None*, ***kwargs*)

Bases: `SingleSpectrum`

The core class for the spectroscopic toolkit. Contains the data and error arrays along with wavelength / frequency / velocity information in various formats.

Create a Spectrum object.

Must either pass in a filename or ALL of xarr, data, and header, plus optionally error.

kwargs are passed to the file reader

Parameters

filename_or_magic : string or something else

The filename or something with an hdu attribute. If data, xarr, and error are specified, leave filename blank.

filetype : string

Specify the file type (only needed if it cannot be automatically determined from the filename)

xarr : `units.SpectroscopicAxis` or `np.ndarray`

The X-axis of the data. If it is an np.ndarray, you must pass `xarrkwargs` or a valid header if you want to use any of the unit functionality.

data : `np.ndarray`

The data array (must have same length as xarr)

error : `np.ndarray`

The error array (must have same length as the data and xarr arrays)

header : `pyfits.Header` or dict

The header from which to read unit information. Needs to be a `pyfits.Header` instance or another dictionary-like object with the appropriate information

maskdata : boolean

turn the array into a masked array with all nan and inf values masked

doplot : boolean

Plot the spectrum after loading it?

plotkwargs : dict

keyword arguments to pass to the plotter

xarrkwargs : dict

keyword arguments to pass to the SpectroscopicAxis initialization (can be used in place of a header)

unit : str

The data unit

filename : string

The file to read the spectrum from. If data, xarr, and error are specified, leave filename blank.

Examples

```
>>> sp = pyspeckit.Spectrum(data=np.random.randn(100),
           xarr=np.linspace(-50, 50, 100), error=np.ones(100)*0.1,
           xarrkwargs={'unit':'km/s', 'refX':4.829, 'refX_unit':'GHz',
               'xtype':'VLSR-RAD'}, header={})
```

```
>>> xarr = pyspeckit.units.SpectroscopicAxis(np.linspace(-50,50,100),
           units='km/s', refX=6562.83, refX_unit='angstroms')
>>> data = np.random.randn(100)*5 + np.random.rand(100)*100
>>> err = np.sqrt(data/5.)*5. # Poisson noise
>>> sp = pyspeckit.Spectrum(data=data, error=err, xarr=xarr, header={})
```

```
>>> # if you already have a simple fits file
>>> sp = pyspeckit.Spectrum('test.fits')
```

copy(*deep=True*)

Create a copy of the spectrum with its own plotter, fitter, etc. Useful for, e.g., comparing smoothed to unsmoothed data

crop(*x1*, *x2*, *unit=None*, ***kwargs*)

Replace the current spectrum with a subset from x1 to x2 in current units

Fixes CRPIX1 and baseline and model spectra to match cropped data spectrum

property data_quantity

downsample(*dsfactor*)

Downsample the spectrum (and all of its subsidiaries) without smoothing

Parameters

dsfactor : int

Downsampling Factor

property error_quantity

property flux

The data in the spectrum (flux = data, for compatibility with astropy's Spectrum1D object).

classmethod from_hdu(*hdu*)

Create a pyspeckit Spectrum object from an HDU

classmethod from_spectrum1d(*spec1d*)

Tool to load a pyspeckit Spectrum from a specutils object

Examples

```
>>> # grab many spectra from a multiextension FITS file
>>> spectra = specutils.io.fits.read_fits_spectrum1d('AAO.fits')
>>> sp = pyspeckit.Spectrum.from_spectrum1d(spectra[0])
```

```
>>> # open a single spectrum that could have been opened directly with pyspeckit
>>> spectrum = specutils.io.fits.read_fits_spectrum1d('gbt_1d.fits')
>>> sp = pyspeckit.Spectrum.from_spectrum1d(spectrum)
```

getlines(*linetype='radio'*, ***kwargs*)

Access a registered database of spectral lines. Will add an attribute with the name linetype, which then has properties defined by the speclines module (most likely, a table and a "show" function to display the lines)

interpnans()

Interpolate over NAN values, replacing them with values interpolated from their neighbors using linear interpolation.

measure(*z=None*, *d=None*, *fluxnorm=None*, *miscline=None*, *misctol=10.0*, *ignore=None*, *derive=True*, ***kwargs*)

Initialize the measurements class - only do this after you have run a fitter otherwise pyspeckit will be angry!

`moments`(*unit='km/s'*, ***kwargs*)

Return the moments of the spectrum. In order to assure that the 1st and 2nd moments are meaningful, a 'default' unit is set. If unit is not set, will use current unit.

Documentation imported from the moments module:

Returns the gaussian parameters of a 1D distribution by calculating its moments. Depending on the input parameters, will only output a subset of the above.

Theory, from first principles (in the absence of noise): integral(gaussian) = sqrt(2*pi*sigma^2) * amp sigma = integral / amp / sqrt(2*pi)

In the presence of noise, this gets much more complicated. The noisy approach is inspired by mpfit

Parameters

Xax : np.ndarray

The x-axis for computing the 1st and 2nd moments

data : np.ndarray

The data from which to compute the various moments

estimator : function

A function to estimate the "height" or "background level" of the data, e.g. mean or median. If masked arrays are being used, use the np.ma versions of the numpy functions

negamp: bool or None :

Force the peak negative (True), positive (False), or the sign of the peak will be "autodetected" (negamp=None)

nsigcut: float or None :

If specified, the code will attempt to estimate the noise and only use data above/below `n`-sigma above the noise. The noise will be estimated from the data unless the noise is specified with `noise_estimate`

noise_estimate: float or None :

Guess for the noise value. Only matters if `nsigcut` is specified.

vheight : bool

Include an estimate of the background level?

Returns

(height, amplitude, x, width_x) :

height : float

is the background level

amplitude : float

is the maximum (or minimum) of the data after background subtraction

x : float

is the first moment

width_x : float

is the second moment

parse_hdf5_header(*hdr*)

HDF5 reader will create a hdr dictionary from HDF5 dataset attributes if they exist. This routine will convert that dict to a pyfits header instance.

parse_header(*hdr*, *specname=None*)

Parse parameters from a .fits header into required spectrum structure parameters

parse_text_header(*Table*)

Grab relevant parameters from a table header (xaxis type, etc)

This function should only exist for Spectrum objects created from .txt or other atpy table type objects

property shape

Return the data shape (a property of the Spectrum)

slice(*start=None*, *stop=None*, *unit='pixel'*, *copy=True*, *xcopy=True*, *preserve_fits=False*)

Slicing the spectrum

Warning: this is the same as cropping right now, but it returns a copy instead of cropping inplace

Parameters

start : numpy.float or int or astropy quantity

start of slice

stop : numpy.float or int or astropy quantity

stop of slice

unit : str

allowed values are any supported physical unit, 'pixel'

copy : bool

Return a 'view' of the data or a copy?

preserve_fits : bool

Save the fitted parameters from self.fitter?

smooth(*smooth*, *downsample=True*, ***kwargs*)

Smooth the spectrum by factor `smooth`.

Documentation from the `smooth` module:

sm.smooth doc:

Smooth and downsample the data array. NaN data points will be replaced with interpolated values

Parameters

smooth : float

Number of pixels to smooth by

smoothtype : ['gaussian','hanning', or 'boxcar']

type of smoothing kernel to use

downsample : bool

Downsample the data?

downsample_factor : int

Downsample by the smoothing factor, or something else?

convmode : ['full','valid','same']

see `numpy.convolve`. 'same' returns an array of the same length as 'data' (assuming data is larger than the kernel)

stats(*statrange=()*, *interactive=False*)

Return some statistical measures in a dictionary (somewhat self-explanatory)

Parameters

statrange : 2-element tuple

X-range over which to perform measures

interactive : bool

specify range interactively in plotter

property unit

property units

write(*filename*, *type=None*, ***kwargs*)

Write the spectrum to a file. The available file types are listed in spectrum.writers.writers

type - what type of file to write to? If not specified, will attempt to determine type from suffix

class pyspeckit.spectrum.classes.**Spectra**(*speclist*, *xunit=None*, *model_registry=None*, ***kwargs*)

Bases: `BaseSpectrum`

A list of individual Spectrum objects. Intended to be used for concatenating different wavelength observations of the SAME OBJECT. Can be operated on just like any Spectrum object, incuding fitting. Useful for fitting multiple lines on non-continguous axes simultaneously. Be wary of plotting these though…

Can be indexed like python lists.

X array is forcibly sorted in increasing order

fiteach(***kwargs*)

Fit each spectrum within the Spectra object

classmethod from_spectrum1d_list(*lst*)

Tool to load a collection of pyspeckit Spectra from a specutils list of Spectrum1D objects

Examples

```
>>> # grab many spectra from a multiextension FITS file
>>> spectra = specutils.io.fits.read_fits_spectrum1d('AAO.fits')
>>> sp = pyspeckit.Spectrum.from_spectrum1d_list(spectra)
```

ploteach(*xunit=None*, *inherit_fit=False*, *plot_fit=True*, *plotfitkwargs={}*, ***plotkwargs*)

Plot each spectrum in its own window inherit_fit - if specified, will grab the fitter & fitter properties from Spectra

smooth(*smooth*, ***kwargs*)

Smooth the spectrum by factor "smooth". Options are defined in sm.smooth

because 'Spectra' does not have a header attribute, don't do anything to it…

class pyspeckit.spectrum.classes.**ObsBlock**(*speclist*, *xtype='frequency'*, *xarr=None*, *force=False*, *model_registry=None*, ***kwargs*)

Bases: Spectra

An Observation Block

Consists of multiple spectra with a shared X-axis. Intended to hold groups of observations of the same object in the same setup for later averaging.

ObsBlocks can be indexed like python lists.

average(*weight=None*, *inverse_weight=False*, *error='erravgrtn'*, *debug=False*)

Average all scans in an ObsBlock. Returns a single Spectrum object

Parameters

weight : string

a header keyword to weight by. If not specified, the spectra will be averaged without weighting

inverse_weight : bool

Is the header keyword an inverse-weight (e.g., a variance?)

error : ['scanrms','erravg','erravgrtn']

estimate the error spectrum by one of three methods. 'scanrms' : the standard deviation of each pixel across all scans 'erravg' : the average of all input error spectra 'erravgrtn' : the average of all input error spectra divided by sqrt(n_obs)

smooth(*smooth*, ***kwargs*)

Smooth the spectrum by factor "smooth". Options are defined in sm.smooth

pyspeckit.spectrum.**register_reader**(*filetype*, *function*, *suffix*, *default=False*)

Register a reader function.

Parameters

filetype: str :

The file type name

function: function :

The reader function. Should take a filename as input and return an X-axis object (see units.py), a spectrum, an error spectrum (initialize it to 0's if empty), and a pyfits header instance

suffix: int :

What suffix should the file have?

pyspeckit.spectrum.**register_writer**(*filetype*, *function*, *suffix*, *default=False*)

Register a writer function.

Parameters

filetype: string :

The file type name

function: function :

The writer function. Will be an attribute of Spectrum object, and called as spectrum.Spectrum.write_hdf5(), for example.

suffix: int :

What suffix should the file have?

4.7.2 Cubes

Pyspeckit can do a few things with spectral cubes. The most interesting is the spectral line fitting.

`Cube` objects have a `fiteach()` method that will fit each spectral line within a cube. It can be made to do this in parallel with the `multicore` option.

As of version 0.16, pyspeckit cubes can be read from SpectralCube objects:

```
>>> pcube = pyspeckit.Cube(cube=mySpectralCube)
```

Otherwise, they can be created from FITS cubes on disk:

```
>>> pcube = pyspeckit.Cube(filename="mycube.fits")
```

or from arrays:

```
>>> mycube = np.random.randn(250,50,50)
>>> myxaxis = np.linspace(-100,100,250)
>>> pcube = pyspeckit.Cube(cube=mycube, xarr=myxaxis, xunit='km/s')
```

The most interesting features of the Cube object are the fiteach() method, which fits a model spectrum to each element of the cube, and mapplot, which plots up various projections of the cube.

Cube.mapplot will create an interactive plot window. You can click on any pixel shown in that window and pull up a second window showing the spectrum at that pixel. If you've fitted the cube, the associated best-fit model will also be shown. This interactive setup can be a bit fragile, though, so please report bugs aggressively so we can weed them out!

The interactive viewer has a few button interactions described here.

Cubes

Tools to deal with spectroscopic data cubes.

Some features in Cubes require additional packages:

- smoothing - requires agpy's smooth and parallel_map routines
- pyregion

The 'grunt work' is performed by the cubes module

class pyspeckit.cubes.SpectralCube.**Cube**(*filename=None*, *cube=None*, *xarr=None*, *xunit=None*, *errorcube=None*, *header=None*, *x0=0*, *y0=0*, *maskmap=None*, ***kwargs*)

Bases: Spectrum

A pyspeckit Cube object. Can be created from a FITS file on disk or from an array or a spectral_cube.SpectralCube object. If an array is used to insantiate the cube, the xarr keyword must be given, specifying the X-axis units

Parameters

filename : str, optional

The name of a FITS file to open and read from. Must be 3D

cube : np.ndarray, spectral_cube.SpectralCube, or astropy.units.Quantity

The data from which to instantiate a Cube object. If it is an array or an astropy Quantity (which is an array with attached units), the X-axis must be specified. If this is given as a SpectralCube object, the X-axis and units should be handled automatically.

xarr : np.ndarray or astropy.units.Quantity, optional

The X-axis of the spectra from each cube. This actually corresponds to axis 0, or what we normally refer to as the Z-axis of the cube, but it indicates the X-axis in a plot of intensity vs wavelength. The units for this array are specified in the xunit keyword unless a Quantity is given.

xunit : str, optional

The unit of the xarr array if xarr is given as a numpy array

errorcube : np.ndarray, spectral_cube.SpectralCube, or Quantity, optional

A cube with the same shape as the input cube providing the 1-sigma error for each voxel. This can be specified more efficiently as an error map for most use cases, but that approach has not yet been implemented. However, you can pass a 2D error map to `fiteach`.

header : `fits.Header` or dict, optional

The header associated with the data. Only needed if the cube is given as an array or a quantity.

x0, y0 : int

The initial spectrum to use. The `Cube` object can be treated as a `pyspeckit.Spectrum` object, with all the associated tools (plotter, fitter) using the `set_spectrum` method to select a pixel from the cube to plot and fit. However, it is generally more sensible to extract individual spectra and treat them separately using the `get_spectrum` method, so these keywords MAY BE DEPRECATED in the future.

maskmap : `np.ndarray`, optional

A boolean mask map, where `True` implies that the data are good. This will be used for both plotting using `mapplot` and fitting using `fiteach`.

kwargs : dict

passed to `load_fits`, which are passed to `SpectralCube.read`

copy(*deep=True*)

Create a copy of the spectral cube with its own plotter, fitter, etc. Useful for, e.g., comparing smoothed to unsmoothed data

crop(*x1*, *x2*, *unit=None*, ***kwargs*)

Replace the current spectrum with a subset from x1 to x2 in current units

Fixes CRPIX1 and baseline and model spectra to match cropped data spectrum

property data_quantity

downsample(*dsfactor*)

Downsample the spectrum (and all of its subsidiaries) without smoothing

Parameters

dsfactor : int

Downsampling Factor

property error_quantity

fiteach(*errspec=None*, *errmap=None*, *guesses=()*, *verbose=True*, *verbose_level=1*, *quiet=True*, *signal_cut=3*, *usemomentcube=None*, *blank_value=0*, *integral=False*, *direct_integral=False*, *absorption=False*, *use_nearest_as_guess=False*, *use_neighbor_as_guess=False*, *start_from_point=(0, 0)*, *multicore=1*, *position_order=None*, *continuum_map=None*, *prevalidate_guesses=False*, *maskmap=None*, *skip_failed_fits=False*, ***fitkwargs*)

Fit a spectrum to each valid pixel in the cube

For guesses, priority is *use_nearest_as_guess*, *usemomentcube*, *guesses*, None

Once you have successfully run this function, the results will be stored in the `.parcube` and `.errcube` attributes, which are each cubes of shape `[npars, ny, nx]`, where npars is the number of fitted parameters and `nx`, `ny` are the shape of the map. `errcube` contains the errors on the fitted parameters (1-sigma, as returned from the Levenberg-Marquardt fit's covariance matrix). You can use the attribute `has_fit`, which is a map of shape `[ny,nx]` to find which pixels have been successfully fit.

Parameters

use_nearest_as_guess: bool :

Unless the fitted point is the first, it will find the nearest other point with a successful fit and use its best-fit parameters as the guess

use_neighbor_as_guess: bool :

Set this keyword to use the average best-fit parameters from neighboring positions with successful fits as the guess

start_from_point: tuple(int,int) :

Either start from the center or from a point defined by a tuple. Work outward from that starting point.

position_order: ndarray[naxis=2] :

2D map of region with pixel values indicating the order in which to carry out the fitting. Any type with increasing pixel values.

guesses: tuple or ndarray[naxis=3] :

Either a tuple/list of guesses with len(guesses) = npars or a cube of guesses with shape [npars, ny, nx]. NOT TRUE, but a good idea in principle: You can also use a dictionary of the form {(y,x): [list of length npars]}, where (y,x) specifies a pixel location. If the dictionary method is used, npars must be specified and it sets the length of the first parameter axis

signal_cut: float :

Minimum signal-to-noise ratio to "cut" on (i.e., if peak in a given spectrum has s/n less than this value, ignore it)

blank_value: float :

Value to replace non-fitted locations with.

errmap: ndarray[naxis=2] or ndarray[naxis=3] :

A map of errors used for the individual pixels of the spectral cube. 2D errmap results in an equal weighting of each given spectrum, while a 3D array sets individual weights of each channel

verbose: bool :

verbose_level: int :

Controls how much is output. 0,1 - only changes frequency of updates in loop 2 - print out messages when skipping pixels 3 - print out messages when fitting pixels 4 - specfit will be verbose

multicore: int :

if >1, try to use multiprocessing via parallel_map to run on multiple cores

continuum_map: np.ndarray :

Same shape as error map. Subtract this from data before estimating noise.

prevalidate_guesses: bool :

An extra check before fitting is run to make sure the guesses are all within the specified limits. May be slow, so it is off by default. It also should not be necessary, since careful checking is performed before each fit.

maskmap : `np.ndarray`, optional

A boolean mask map, where `True` implies that the data are good. This will be used for both plotting using `mapplot` and fitting using `fiteach`. If `None`, will use `self.maskmap`.

integral : bool

If set, the integral of each spectral fit will be computed and stored in the attribute .`integralmap`

direct_integral : bool

Return the integral of the *spectrum* (as opposed to the fitted model) over a range defined by the `integration_limits` if specified or `threshold` otherwise

skip_failed_fits : bool

Flag to forcibly skip failed fits that fail with "unknown error". Generally, you do not want this on, but this is the 'finger-over-the-engine-light' approach that will allow these incomprehensible failures to go by and just ignore them. Keep an eye on how many of these you get: if it's just one or two out of hundreds, then maybe those are just pathological cases that can be ignored. If it's a significant fraction, you probably want to take a different approach.

property flux

The data in the spectrum (flux = data, for compatibility with astropy's Spectrum1D object).

classmethod from_hdu(*hdu*)

Create a pyspeckit Spectrum object from an HDU

classmethod from_spectrum1d(*spec1d*)

Tool to load a pyspeckit Spectrum from a specutils object

Examples

```
>>> # grab many spectra from a multiextension FITS file
>>> spectra = specutils.io.fits.read_fits_spectrum1d('AAO.fits')
>>> sp = pyspeckit.Spectrum.from_spectrum1d(spectra[0])
```

```
>>> # open a single spectrum that could have been opened directly with pyspeckit
>>> spectrum = specutils.io.fits.read_fits_spectrum1d('gbt_1d.fits')
>>> sp = pyspeckit.Spectrum.from_spectrum1d(spectrum)
```

get_apspec(*aperture*, *coordsys=None*, *method='mean'*, ***kwargs*)

Extract an aperture using cubes.extract_aperture (defaults to Cube pixel coordinates)

***aperture* [tuple or list] (x, y, radius)**

The aperture to use when extracting the data

***coordsys* ['celestial' | 'galactic' | None]**

the coordinate system the aperture is specified in None indicates pixel coordinates (default)

***wunit* [str]**

arcsec, arcmin, or degree

get_modelcube(*update=False*, *multicore=1*)

Return or generate a "model cube", which will have the same shape as the `.cube` but will have spectra generated from the fitted model.

If the model cube does not yet exist, one will be generated

Parameters

update : bool

If the cube has already been computed, set this to `True` to recompute the model.

multicore: int :

if >1, try to use multiprocessing via parallel_map to run on multiple cores

get_spectrum(*x*, *y*)

Very simple: get the spectrum at coordinates x,y

(inherits fitter from self)

Returns a SpectroscopicAxis instance

getlines(*linetype='radio'*, ***kwargs*)

Access a registered database of spectral lines. Will add an attribute with the name linetype, which then has properties defined by the speclines module (most likely, a table and a "show" function to display the lines)

interpnans()

Interpolate over NAN values, replacing them with values interpolated from their neighbors using linear interpolation.

load_fits(*fitsfile*, ***kwargs*)

load_model_fit(*fitsfilename*, *npars*, *npeaks=1*, *fittype=None*, *_temp_fit_loc=(0, 0)*)

Load a parameter + error cube into the `.parcube` and `.errcube` attributes. The models can then be examined and plotted using `.mapplot` as if you had run `.fiteach`.

Parameters

fitsfilename : str

The filename containing the parameter cube written with `write_fit`

npars : int

The number of parameters in the model fit for a single spectrum

npeaks : int

The number of independent peaks fit toward each spectrum

fittype : str, optional

The name of the fittype, e.g. 'gaussian' or 'voigt', from the pyspeckit fitter registry. This is optional; it should have been written to the FITS header and will be read from there if it is not specified

_temp_fit_loc : tuple (int,int)

The initial spectrum to use to generate components of the class. This should not need to be changed.

load_spectral_cube(*cube*)

Load the cube from a spectral_cube.SpectralCube object

measure(*z=None*, *d=None*, *fluxnorm=None*, *miscline=None*, *misctol=10.0*, *ignore=None*, *derive=True*, ***kwargs*)

Initialize the measurements class - only do this after you have run a fitter otherwise pyspeckit will be angry!

momenteach(*verbose=True*, *verbose_level=1*, *multicore=1*, ***kwargs*)

Return a cube of the moments of each pixel

Parameters

multicore: int :

if >1, try to use multiprocessing via parallel_map to run on multiple cores

moments(*unit='km/s'*, ***kwargs*)

Return the moments of the spectrum. In order to assure that the 1st and 2nd moments are meaningful, a 'default' unit is set. If unit is not set, will use current unit.

Documentation imported from the moments module:

Returns the gaussian parameters of a 1D distribution by calculating its moments. Depending on the input parameters, will only output a subset of the above.

Theory, from first principles (in the absence of noise): integral(gaussian) = sqrt(2*pi*sigma^2) * amp sigma = integral / amp / sqrt(2*pi)

In the presence of noise, this gets much more complicated. The noisy approach is inspired by mpfit

Parameters

Xax : np.ndarray

The x-axis for computing the 1st and 2nd moments

data : np.ndarray

The data from which to compute the various moments

estimator : function

A function to estimate the "height" or "background level" of the data, e.g. mean or median. If masked arrays are being used, use the np.ma versions of the numpy functions

negamp: bool or None :

Force the peak negative (True), positive (False), or the sign of the peak will be "autodetected" (negamp=None)

nsigcut: float or None :

If specified, the code will attempt to estimate the noise and only use data above/below `n`-sigma above the noise. The noise will be estimated from the data unless the noise is specified with `noise_estimate`

noise_estimate: float or None :

Guess for the noise value. Only matters if `nsigcut` is specified.

vheight : bool

Include an estimate of the background level?

Returns

(height, amplitude, x, width_x) :

height : float

is the background level

amplitude : float

is the maximum (or minimum) of the data after background subtraction

x : float

is the first moment

width_x : float

is the second moment

`parse_hdf5_header`(*hdr*)

HDF5 reader will create a hdr dictionary from HDF5 dataset attributes if they exist. This routine will convert that dict to a pyfits header instance.

`parse_header`(*hdr*, *specname=None*)

Parse parameters from a .fits header into required spectrum structure parameters

parse_text_header(*Table*)

Grab relevant parameters from a table header (xaxis type, etc)

This function should only exist for Spectrum objects created from .txt or other atpy table type objects

plot_apspec(*aperture*, *coordsys=None*, *reset_ylimits=True*, *wunit='arcsec'*, *method='mean'*, ***kwargs*)

Extract an aperture using cubes.extract_aperture (defaults to Cube coordinates)

Parameters

aperture : list

A list of aperture parameters, e.g.

- For a circular aperture, len(ap)=3: `+ ap = [xcen,ycen,radius]`
- For an elliptical aperture, len(ap)=5: `+ ap = [xcen,ycen,height,width,PA]`

coordsys : None or str

The coordinate system of the aperture (e.g., galactic, fk5, None for pixel)

method : 'mean' or 'sum'

Either average over parellel spectra or sum them.

plot_fit(*x*, *y*, *silent=False*, ***kwargs*)

If fiteach has been run, plot the best fit at the specified location

Parameters

x : int

y : int

The x, y coordinates of the pixel (indices 2 and 1 respectively in numpy notation)

plot_spectrum(*x*, *y*, *plot_fit=False*, ***kwargs*)

Fill the .data array with a real spectrum and plot it

set_apspec(*aperture*, *coordsys=None*, *method='mean'*)

Extract an aperture using cubes.extract_aperture (defaults to Cube coordinates)

set_spectrum(*x*, *y*)

property shape

Return the data shape (a property of the Spectrum)

show_fit_param(*parnumber*, ***kwargs*)

If pars have been computed, display them in the mapplot window

Parameters

parnumber : int

The index of the parameter in the parameter cube

show_moment(*momentnumber*, ***kwargs*)

If moments have been computed, display them in the mapplot window

slice(*start=None*, *stop=None*, *unit='pixel'*, *preserve_fits=False*, *copy=True*, *update_header=False*)

Slice a cube along the spectral axis (equivalent to "spectral_slab" from the spectral_cube package)

Parameters

start : numpy.float or int

start of slice

stop : numpy.float or int

stop of slice

unit : str

allowed values are any supported physical unit, 'pixel'

update_header : bool

modifies the header of the spectral cube according to the slice

smooth(*factor*, ***kwargs*)

Smooth the spectrum by factor `factor`.

Documentation from the `cubes.spectral_smooth` module:

stats(*statrange=()*, *interactive=False*)

Return some statistical measures in a dictionary (somewhat self-explanatory)

Parameters

statrange : 2-element tuple

X-range over which to perform measures

interactive : bool

specify range interactively in plotter

property unit

property units

write(*filename*, *type=None*, ***kwargs*)

Write the spectrum to a file. The available file types are listed in spectrum.writers.writers

type - what type of file to write to? If not specified, will attempt to determine type from suffix

write_cube()

write_fit(*fitcubefilename*, *overwrite=False*)

Write out a fit cube containing the `.parcube` and `.errcube` using the information in the fit's parinfo to set the header keywords. The `PLANE#` keywords will be used to indicate the content of each plane in the data cube written to the FITS file. All of the fitted parameters will be written first, followed by all of the errors on those parameters. So, for example, if you have fitted a single gaussian to each pixel, the dimensions of the saved cube will be `[6, ny, nx]`, and they will be the amplitude, centroid, width, error on amplitude, error on centroid, and error on width, respectively.

To load such a file back in for plotting purposes, see `SpectralCube.load_model_fit`.

Parameters

fitcubefilename: string :

Filename to write to

overwrite: bool :

Overwrite file if it exists?

class pyspeckit.cubes.SpectralCube.**CubeStack**(*cubelist*, *xunit='GHz'*, *x0=0*, *y0=0*, *maskmap=None*, ***kwargs*)

Bases: Cube

The Cube equivalent of Spectra: for stitching multiple cubes with the same spatial grid but different frequencies together

Initialize the Cube. Accepts FITS files.

x0,y0 - initial spectrum to use (defaults to lower-left corner)

MapPlot

Make plots of the cube and interactively connect them to spectrum plotting. This is really an interactive component of the package; nothing in here is meant for publication-quality plots, but more for user interactive analysis.

That said, the plotter makes use of APLpy, so it is possible to make publication-quality plots.

author

Adam Ginsburg

date

03/17/2011

class pyspeckit.cubes.mapplot.**MapPlotter**(*Cube=None*, *figure=None*, *doplot=False*, ***kwargs*)

Bases: object

Class to plot a spectrum

See mapplot for use documentation; this docstring is only for initialization.

Create a map figure for future plotting

circle(*x1*, *y1*, *x2*, *y2*, ***kwargs*)

Plot the spectrum of a circular aperture

click(*event*)

Record location of downclick

copy(*parent=None*)

Create a copy of the map plotter with blank (uninitialized) axis & figure

[parent]

A spectroscopic axis instance that is the parent of the specfit instance. This needs to be specified at some point, but defaults to None to prevent overwriting a previous plot.

makeplane(*estimator=<function nanmean>*)

Create a "plane" view of the cube, either by slicing or projecting it or by showing a slice from the best-fit model parameter cube.

Parameters

estimator : [function | 'max' | 'int' | FITS filename | integer | slice]

A non-pythonic, non-duck-typed variable. If it's a function, apply that function along the cube's spectral axis to obtain an estimate (e.g., mean, min, max, etc.). 'max' will do the same thing as passing np.max 'int' will attempt to integrate the image (which is why I didn't duck-type) (integrate means sum and multiply by dx) a .fits filename will be read using pyfits (so you can make your own cover figure) an integer will get the n'th slice in the parcube if it exists If it's a slice, slice the input data cube along the Z-axis with this slice

mapplot(*convention='calabretta'*, *colorbar=True*, *useaplpy=True*, *vmin=None*, *vmax=None*, *cmap=None*, *plotkwargs={}*, ***kwargs*)

Plot up a map based on an input data cube.

The map to be plotted is selected using makeplane. The `estimator` keyword argument is passed to that function.

The plotted map, once shown, is interactive. You can click on it with any of the three mouse buttons.

Button 1 or keyboard '1':

Plot the selected pixel's spectrum in another window. Mark the clicked pixel with an 'x'

Button 2 or keyboard 'o':

Overplot a second (or third, fourth, fifth...) spectrum in the external plot window

Button 3:

Disconnect the interactive viewer

You can also click-and-drag with button 1 to average over a circular region. This same effect can be achieved by using the 'c' key to set the /c/enter of a circle and the 'r' key to set its /r/adius (i.e., hover over the center and press 'c', then hover some distance away and press 'r').

Parameters

convention : 'calabretta' or 'griesen'

The default projection to assume for Galactic data when plotting with aplpy.

colorbar : bool

Whether to show a colorbar

plotkwargs : dict, optional

A dictionary of keyword arguments to pass to aplpy.show_colorscale or matplotlib.pyplot.imshow

useaplpy : bool

Use aplpy if a FITS header is available

vmin, vmax: float or None :

Override values for the vmin/vmax values. Will be automatically determined if left as None

.. todo: :

Allow mapplot in subfigure

plot_spectrum(*event*, *plot_fit=True*)

Connects map cube to Spectrum...

refresh()

cubes.py

From agpy, contains functions to perform various transformations on data cubes and their headers.

pyspeckit.cubes.cubes.**aper_world2pix**(*ap*, *wcs*, *coordsys='galactic'*, *wunit='arcsec'*)

Converts an elliptical aperture (x,y,width,height,PA) from WCS to pixel coordinates given an input wcs (an instance of the pywcs.WCS class). Must be a 2D WCS header.

pyspeckit.cubes.cubes.**baseline_cube**(*cube*, *polyorder=None*, *cubemask=None*, *splineorder=None*, *numcores=None*, *sampling=1*)

Given a cube, fit a polynomial to each spectrum

Parameters

cube: np.ndarray :

An ndarray with ndim = 3, and the first dimension is the spectral axis

polyorder: int :

Order of the polynomial to fit and subtract

cubemask: boolean ndarray :

Mask to apply to cube. Values that are True will be ignored when fitting.

numcores : None or int

Number of cores to use for parallelization. If None, will be set to the number of available cores.

pyspeckit.cubes.cubes.**blfunc_generator**(*x=None*, *polyorder=None*, *splineorder=None*, *sampling=1*)

Generate a function that will fit a baseline (polynomial or spline) to a data set. Either `splineorder` or `polyorder` must be set

Parameters

x : np.ndarray or None

The X-axis of the fitted array. Will be set to `np.arange(len(data))` if not specified

polyorder : None or int

The polynomial order.

splineorder : None or int

sampling : int

The sampling rate to use for the data. Can set to higher numbers to effectively downsample the data before fitting

pyspeckit.cubes.cubes.**coords_in_image**(*fitsfile*, *lon*, *lat*, *system='galactic'*)

Determine whether the coordinates are inside the image

pyspeckit.cubes.cubes.**extract_aperture**(*cube*, *ap*, *r_mask=False*, *wcs=None*, *coordsys='galactic'*, *wunit='arcsec'*, *debug=False*, *method='mean'*)

Extract an aperture from a data cube. E.g. to acquire a spectrum of an outflow that is extended.

Cube should have shape [z,y,x], e.g. cube = fits.getdata('datacube.fits')

Apertures are specified in PIXEL units with an origin of 0,0 (NOT the 1,1 fits standard!) unless wcs and coordsys are specified

Parameters

ap : list

For a circular aperture, len(ap)=3:

ap = [xcen,ycen,radius]

For an elliptical aperture, len(ap)=5:

ap = [xcen,ycen,height,width,PA]

wcs : wcs

a pywcs.WCS instance associated with the data cube

coordsys : str

the coordinate system the aperture is specified in. Options are 'celestial' and 'galactic'. Default is 'galactic'

wunit : str

units of width/height. default 'arcsec', options 'arcmin' and 'degree'

method : str

'mean' or 'sum' (average over spectra, or sum them) or 'error' for sqrt(sum-of-squares / n)

pyspeckit.cubes.cubes.**flatten_header**(*header*, *delete=False*)

Attempt to turn an N-dimensional fits header into a 2-dimensional header Turns all CRPIX[>2] etc. into new keywords with suffix 'A'

header must be a fits.Header instance

pyspeckit.cubes.cubes.**getspec**(*lon*, *lat*, *rad*, *cube*, *header*, *r_fits=True*, *inherit=True*, *wunit='arcsec'*)

Given a longitude, latitude, aperture radius (arcsec), and a cube file, return a .fits file or a spectrum.

Parameters

lon: float :

lat: float :

longitude and latitude center of a circular aperture in WCS coordinates must be in coordinate system of the file

rad: float :

radius (default degrees) of aperture

pyspeckit.cubes.cubes.**getspec_reg**(*cubefilename*, *region*, ***kwargs*)

Aperture extraction from a cube using a pyregion circle region

The region must be in the same coordinate system as the cube header

Warning: The second argument of getspec_reg requires a pyregion region list, and therefore this code depends on pyregion.

pyspeckit.cubes.cubes.**integ**(*file*, *vrange*, *xcen=None*, *xwidth=None*, *ycen=None*, *ywidth=None*, ***kwargs*)

wrapper of subimage_integ that defaults to using the full image

pyspeckit.cubes.cubes.**plane_smooth**(*cube*, *cubedim=0*, *parallel=True*, *numcores=None*, ***kwargs*)

parallel-map the smooth function

Parameters

parallel: bool :

defaults True. Set to false if you want serial (for debug purposes?)

numcores: int :

pass to parallel_map (None = use all available)

pyspeckit.cubes.cubes.**speccen_header**(*header*, *lon=None*, *lat=None*, *proj='TAN'*, *system='celestial'*, *spectral_axis=3*, *celestial_axes=[1, 2]*)

Turn a cube header into a spectrum header, retaining RA/Dec vals where possible (speccen is like flatten; spec-ify would be better but, specify? nah)

Assumes 3rd axis is velocity

pyspeckit.cubes.cubes.**spectral_smooth**(*cube*, *smooth_factor*, *downsample=True*, *parallel=True*, *numcores=None*, ***kwargs*)

Smooth the cube along the spectral direction

pyspeckit.cubes.cubes.**subcube**(*cube*, *xcen*, *xwidth*, *ycen*, *ywidth*, *header=None*, *dvmult=False*, *return_HDU=False*, *units='pixels'*, *widthunits='pixels'*)

Crops a data cube

All units assumed to be pixel units

cube has dimensions (velocity, y, x)

xwidth and ywidth are "radius" values, i.e. half the length that will be extracted

if dvmult is set, multiple the average by DV (this is useful if you set average=sum and dvmul=True to get an integrated value)

pyspeckit.cubes.cubes.**subimage_integ**(*cube*, *xcen*, *xwidth*, *ycen*, *ywidth*, *vrange*, *header=None*, *average=<function mean>*, *dvmult=False*, *return_HDU=False*, *units='pixels'*, *zunits=None*)

Returns a sub-image from a data cube integrated over the specified velocity range

NOTE: With spectral_cube, subcube features can be easily applied with the subcube method, and integration is handled separately.

Parameters

cube : np.ndarray

A 3-dimensional numpy array with dimensions (velocity, y, x)

xcen,ycen : float

The center in the X,Y-dimension. See `units` below for unit information

xwidth,ywidth : float

The width in the X,Y-dimension. See `units` below for unit information xwidth and ywidth are "radius" values, i.e. half the length that will be extracted

vrange : (float,float)

The velocity range to integrate over. See `zunits` below for unit information

header : `astropy.io.fits.Header` or None

If specified, will allow the use of WCS units

average : function

The function to apply when 'integrating' over the subcube

dvmult : bool

If dvmult is set, multiply the average by DV (this is useful if you set average=sum and dvmul=True to get an integrated value, e.g. K km/s or Jy km/s)

return_hdu : bool

If specified, will return an HDU object, otherwise will return the array and header

units : 'pixels' or 'wcs'

If 'pixels', all units (xcen, ycen, xwidth, ywidth) will be in pixels. If 'wcs', the values will be converted from WCS units to pixel units using the WCS specified by the `header`

zunits : 'pixels' or 'wcs' or None

If None, will be set to be the same as `units`

Returns

subim, hdu : tuple

A tuple (integrated array, header) if `return_hdu` is `False`, or an HDU if it is True

CHAPTER

FIVE

READERS

5.1 Plain Text

Text files should be of the form:

```
wavelength flux err
3637.390 0.314 0.000
3638.227 0.717 0.000
3639.065 1.482 0.000
```

where there 'err' column is optional but the others are not. The most basic spectrum file allowed would have no header and two columns, e.g.:

```
1  0.5
2  1.5
3  0.1
```

If the X-axis is not monotonic, the data will be sorted so that the X-axis is in ascending order.

5.1.1 API

Routines for reading in ASCII format spectra. If astropy.io.ascii is not installed, will use a very simple routine for reading in the data.

pyspeckit.spectrum.readers.txt_reader.**open_1d_txt**(*filename*, *xaxcol=0*, *datacol=1*, *errorcol=2*, *text_reader='simple'*, *format=None*, ***kwargs*)

Attempt to read a 1D spectrum from a text file assuming wavelength as the first column, data as the second, and (optionally) error as the third.

Reading can be done either with astropy.io.ascii or a 'simple' reader. If you have an IPAC, CDS, or formally formatted table, you'll want to use astropy.io.ascii and specify a format.

If you have a simply formatted file of the form, e.g. # name name # unit unit data data data data

kwargs are passed to astropy.io.ascii.read

pyspeckit.spectrum.readers.txt_reader.**simple_txt**(*filename*, *xaxcol=0*, *datacol=1*, *errorcol=2*, *skiplines=0*, ***kwargs*)

Very simple method for reading columns from ASCII file.

5.2 FITS

A minimal header should look like this:

```
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -32 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  659
NAXIS2  =                    2
CRPIX1  =                  1.0
CRVAL1  =   -4953.029632560421
CDELT1  =    212.5358581542998
CTYPE1  = 'VRAD-LSR'
CUNIT1  = 'm/s     '
BUNIT   = 'K       '
RESTFRQ =          110.20137E9
SPECSYS = 'LSRK    '
END
```

A fits file with a header as above should be easily readable without any user effort:

```
sp = pyspeckit.Spectrum('test.fits')
```

If you have multiple spectroscopic axes, e.g.

```
CRPIX1A =                  1.0
CRVAL1A =    110.2031747948101
CTYPE1A = 'FREQ-LSR'
CUNIT1A = 'GHz     '
RESTFRQA=            110.20137
```

you can load that axis with the 'wcstype' keyword:

```
sp = pyspeckit.Spectrum('test.fits',wcstype='A')
```

If you have a .fits file with a non-linear X-axis that is stored in the .fits file as data (as opposed to being implicitly included in a heaer), you can load it using a custom .fits reader. An example implementation is given in the tspec_reader. It can be registered using *Registration*:

```
tspec_reader = check_reader(tspec_reader.tspec_reader)
pyspeckit.register_reader('tspec',tspec_reader,'fits')
```

5.2.1 API

pyspeckit.spectrum.readers.fits_reader.**open_1d_fits**(*filename*, *hdu=0*, ***kwargs*)

Grabs all the relevant pieces of a simple FITS-compliant 1d spectrum

Inputs:

wcstype - the suffix on the WCS type to get to

velocity/frequency/whatever

specnum - Which # spectrum, along the y-axis, is

the data?

errspecnum - Which # spectrum, along the y-axis,

is the error spectrum?

pyspeckit.spectrum.readers.fits_reader.**open_1d_pyfits**(*pyfits_hdu*, *specnum=0*, *wcstype=''*, *specaxis='1'*, *errspecnum=None*, *autofix=True*, *scale_keyword=None*, *scale_action=<built-in function truediv>*, *verbose=False*, *apnum=0*, ***kwargs*)

This is open_1d_fits but for a pyfits_hdu so you don't necessarily have to open a fits file

pyspeckit.spectrum.readers.fits_reader.**read_echelle**(*pyfits_hdu*)

Read an IRAF Echelle spectrum

http://iraf.noao.edu/iraf/ftp/iraf/docs/specwcs.ps.Z

5.3 hdf5

(work in progress)

5.3.1 API

Routines for reading in spectra from HDF5 files.

Note: Current no routines for parsing HDF5 headers in classes.py.

pyspeckit.spectrum.readers.hdf5_reader.**open_hdf5**(*filename*, *xaxkey='xarr'*, *datakey='data'*, *errkey='error'*)

This reader expects three datasets to exist in the hdf5 file 'filename': 'xarr', 'data', and 'error', by default. Can specify other dataset names.

5.4 Gildas CLASS files

Pyspeckit is capable of reading files from some versions of CLASS. The CLASS developers have stated that the GILDAS file format is private and will remain so, and therefore there are no guarantees that the CLASS reader will work for your file.

Nonetheless, if you want to develop in python instead of SIC, the `read_class` module is probably the best way to access CLASS data.

The CLASS file specification is incomplete, so much of the data reading is hacked together. The code style is based off of Tom Robitaille's idlsave package.

An example usage. Note that `telescope` and `line` are NOT optional keyword arguments, they are just specified as such for clarity

```
n2hp = class_to_obsblocks(fn1, telescope=['SMT-F1M-HU','SMT-F1M-VU'],
    line=['N2HP(3-2)','N2H+(3-2)'])
```

This will generate a `ObsBlock` from all data tagged with the 'telescope' flags listed and lines matching either of those above. The data selection is equivalent to a combination of

```
find /telescope SMT-F1M-HU
find /telescope SMT-F1M-VU
find /line N2HP(3-2)
find /line N2H+(3-2)
```

ALL of the data matching those criteria will be included in an ObsBlock. They will then be accessible through the ObsBlock's `speclist` attribute, or just by indexing the ObsBlock directly.

5.4.1 An essentially undocumented API

GILDAS CLASS file reader

Read a CLASS file into an `pyspeckit.spectrum.ObsBlock`

class pyspeckit.spectrum.readers.read_class.**LazyItem**(*parent*)

Simple lazy spectrum-retriever wrapper

pyspeckit.spectrum.readers.read_class.**class_to_obsblocks**(**arg*, ***kwargs*)

Load an entire CLASS observing session into a list of ObsBlocks based on matches to the 'telescope', 'line' and 'source' names

Parameters

filename : string

The Gildas CLASS data file to read the spectra from.

telescope : list

List of telescope names to be matched.

line : list

List of line names to be matched.

source : list (optional)

List of source names to be matched. Defaults to None.

imagfreq : bool

Create a SpectroscopicAxis with the image frequency.

pyspeckit.spectrum.readers.read_class.**class_to_spectra**(**arg*, ***kwargs*)

Load each individual spectrum within a CLASS file into a list of Spectrum objects

pyspeckit.spectrum.readers.read_class.**downsample_1d**(*myarr*, *factor*, *estimator=<function mean>*, *weight=None*)

Downsample a 1D array by averaging over *factor* pixels. Crops right side if the shape is not a multiple of factor.

This code is pure numpy and should be fast.

keywords:

estimator - default to mean. You can downsample by summing or

something else if you want a different estimator (e.g., downsampling error: you want to sum & divide by sqrt(n))

weight: np.ndarray

An array of weights to use for the downsampling. If None, assumes uniform 1

pyspeckit.spectrum.readers.read_class.**ensure_bytes**(*string*)

Ensure a given string is in byte form

pyspeckit.spectrum.readers.read_class.**filedescv2_nw1 = 14**

GENERAL

integer(kind=obsnum_length) :: num ! [ ] Observation number integer(kind=4) :: ver ! [ ] Version number integer(kind=4) :: teles(3) ! [ ] Telescope name integer(kind=4) :: dobs ! [MJD-60549] Date of observation integer(kind=4) :: dred ! [MJD-60549] Date of reduction integer(kind=4) :: typec ! [code] Type of coordinates integer(kind=4) :: kind ! [code] Type of data integer(kind=4) :: qual ! [code] Quality of data integer(kind=4) :: subscan ! [ ] Subscan number integer(kind=obsnum_length) :: scan ! [ ] Scan number ! Written in the entry real(kind=8) :: ut ! 1-2 [rad] UT of observation real(kind=8) :: st ! 3-4 [rad] LST of observation real(kind=4) :: az ! 5 [rad] Azimuth real(kind=4) :: el ! 6 [rad] Elevation real(kind=4) :: tau ! 7 [neper] Opacity real(kind=4) :: tsys ! 8 [K] System temperature real(kind=4) :: time ! 9 [s] Integration time ! Not in this section in file integer(kind=4) :: xunit ! [code] X unit (if X coordinates section is present) ! NOT in data — character(len=12) :: cdobs ! [string] Duplicate of dobs character(len=12) :: cdred ! [string] Duplicate of dred

pyspeckit.spectrum.readers.read_class.**gi8_dicho**(*ninp*, *lexn*, *xval*, *ceil=True*)

! @ public ! Find ival such as ! X(ival-1) < xval <= X(ival) (ceiling mode) ! or ! X(ival) <= xval < X(ival+1) (floor mode) ! for input data ordered. Use a dichotomic search for that. call gi8_dicho(nex,file%desc%lexn,entry_num,.true.,kex,error)

pyspeckit.spectrum.readers.read_class.**is_ascii**(*s*)

Check if there are non-ascii characters in Unicode string

Parameters

s : str

The string to be checked

Returns

is_ascii : bool

Returns True if all characters in the string are ascii. False otherwise.

pyspeckit.spectrum.readers.read_class.**make_axis**(*header*, *imagfreq=False*)

Create a pyspeckit.spectrum.units.SpectroscopicAxis from the CLASS "header"

pyspeckit.spectrum.readers.read_class.**print_timing**(*func*)

Prints execution time of decorated function. Included here because CLASS files can take a little while to read; this should probably be replaced with a progressbar

pyspeckit.spectrum.readers.read_class.**read_class**(**arg*, ***kwargs*)

Read a binary class file. Based on the GILDAS CLASS file type Specification

Parameters

filename: str :

downsample_factor: None or int :

Factor by which to downsample data by averaging. Useful for overresolved data.

sourcename: str or list of str :

Source names to match to the data (uses regex)

telescope: str or list of str :

'XTEL' or 'TELE' parameters: the telescope & instrument

line: str or list of str :

The line name

posang: tuple of 2 floats :

The first float is the minimum value for the position angle. The second float is the maximum value for the position angle.

verbose: bool :

Log messages with severity INFO

flag_array: np.ndarray :

An array with the same shape as the data used to flag out (remove) data when downsampling. True = flag out

pyspeckit.spectrum.readers.read_class.**tests**()

Tests are specific to the machine on which this code was developed.

5.5 GBTIDL FITS files

GBTIDL SDFITS sessions can be loaded as `pyspeckit.ObsBlock` objects using the GBTSession reader:

```
gbtsession = pyspeckit.readers.GBTSession('AGBTsession.fits')
```

5.5.1 API

GBTIDL SDFITS file

GBTIDL SDFITS files representing GBT observing sessions can be read into pyspeckit. Additional documentation is needed. Nodding reduction is supported, frequency switching is not.

class pyspeckit.spectrum.readers.gbt.**GBTSession**(*sdfitsfile*)

A class wrapping all of the above features

Load an SDFITS file or a pre-loaded FITS file

load_target(*target*, ***kwargs*)

Load a Target...

reduce_all()

reduce_target(*target*, ***kwargs*)

Reduce the data for a given object name

class pyspeckit.spectrum.readers.gbt.**GBTTarget**(*Session*, *target*, ***kwargs*)

A collection of ObsBlocks or Spectra

Container for the individual scans of a target from a GBT session

reduce(*obstype='nod'*, ***kwargs*)

Reduce nodded observations (they should have been read in __init__)

pyspeckit.spectrum.readers.gbt.**average_IF**(*block*, *debug=False*)

Average the polarizations for each feed in each IF

pyspeckit.spectrum.readers.gbt.**average_pols**(*block*)

Average the polarizations for each feed in each IF

pyspeckit.spectrum.readers.gbt.**count_integrations**(*sdfitsfile*, *target*)

Return the number of integrations for a given target (uses one sampler; assumes same number for all samplers)

pyspeckit.spectrum.readers.gbt.**dcmeantsys**(*calon*, *caloff*, *tcal*, *debug=False*)

from GBTIDL's dcmeantsys.py ; mean_tsys = tcal * mean(nocal) / (mean(withcal-nocal)) + tcal/2.0

pyspeckit.spectrum.readers.gbt.**find_feeds**(*block*)

Get a dictionary of the feed numbers for each sampler

pyspeckit.spectrum.readers.gbt.**find_matched_freqs**(*reduced_blocks*, *debug=False*)

Use frequency-matching to find which samplers observed the same parts of the spectrum

WARNING These IF numbers don't match GBTIDL's! I don't know how to get those to match up!

pyspeckit.spectrum.readers.gbt.**find_pols**(*block*)

Get a dictionary of the polarization for each sampler

pyspeckit.spectrum.readers.gbt.**identify_samplers**(*block*)

Identify each sampler with an IF number, a feed number, and a polarization

pyspeckit.spectrum.readers.gbt.**list_targets**(*sdfitsfile*, *doprint=True*)

List the targets, their location on the sky...

pyspeckit.spectrum.readers.gbt.**read_gbt_scan**(*sdfitsfile*, *obsnumber=0*)

Read a single scan from a GBTIDL SDFITS file

pyspeckit.spectrum.readers.gbt.**read_gbt_target**(*sdfitsfile*, *objectname*, *verbose=False*)

Give an object name, get all observations of that object as an 'obsblock'

pyspeckit.spectrum.readers.gbt.**reduce_gbt_target**(*sdfitsfile*, *objectname*, *nbeams*, *verbose=False*)

Wrapper - read an SDFITS file, get an object, reduce it (assuming nodded) and return it

pyspeckit.spectrum.readers.gbt.**reduce_nod**(*blocks*, *verbose=False*, *average=True*, *fdid=(1, 2)*)

Do a nodded on/off observation given a dict of observation blocks as produced by read_gbt_target

Parameters

fdid : 2-tuple

pyspeckit.spectrum.readers.gbt.**reduce_totalpower**(*blocks*, *verbose=False*, *average=True*, *fdid=1*)

Reduce a total power observation

pyspeckit.spectrum.readers.gbt.**round_to_resolution**(*frequency*, *resolution*)

kind of a hack, but round the frequency to the nearest integer multiple of the resolution, then multiply it back into frequency space

pyspeckit.spectrum.readers.gbt.**sigref**(*nod1*, *nod2*, *tsys_nod2*)

Signal-Reference ('nod') calibration ; ((dcsig-dcref)/dcref) * dcref.tsys see GBTIDL's dosigref

pyspeckit.spectrum.readers.gbt.**totalpower**(*calon*, *caloff*, *average=True*)

Do a total-power calibration of an on/off data set (see dototalpower.pro in GBTIDL)

pyspeckit.spectrum.readers.gbt.**uniq**(*seq*)

from http://stackoverflow.com/questions/480214/how-do-you-remove-duplicates-from-a-list-in-python-whilst-preserving-order

CHAPTER

SIX

WRAPPERS

These are wrappers to simplify some of the more complicated (and even some of the simpler) functions in PySpecKit

6.1 Cube Fitting

Complicated code for fitting of a whole data cube, pixel-by-pixel

pyspeckit.wrappers.cube_fit.**cube_fit**(*cubefilename*, *outfilename*, *errfilename=None*, *scale_keyword=None*, *vheight=False*, *verbose=False*, *signal_cut=3*, *verbose_level=2*, *overwrite=True*, ***kwargs*)

Light-weight wrapper for cube fitting

Takes a cube and error map (error will be computed naively if not given) and computes moments then fits for each spectrum in the cube. It then saves the fitted parameters to a reasonably descriptive output file whose header will look like

```
PLANE1  = 'amplitude'
PLANE2  = 'velocity'
PLANE3  = 'sigma'
PLANE4  = 'err_amplitude'
PLANE5  = 'err_velocity'
PLANE6  = 'err_sigma'
PLANE7  = 'integral'
PLANE8  = 'integral_error'
CDELT3  = 1
CTYPE3  = 'FITPAR'
CRVAL3  = 0
CRPIX3  = 1
```

Parameters

errfilename: [None | string name of .fits file] :

A two-dimensional error map to use for computing signal-to-noise cuts

scale_keyword: [None | Char] :

Keyword to pass to the data cube loader - multiplies cube by the number indexed by this header kwarg if it exists. e.g., if your cube is in T_A units and you want T_A*

vheight: [bool] :

Is there a background to be fit? Used in moment computation

verbose: [bool] :

verbose_level: [int] :

How loud will the fitting procedure be? Passed to momenteach and fiteach

signal_cut: [float] :

Signal-to-Noise ratio minimum. Spectra with a peak below this S/N ratio will not be fit and will be left blank in the output fit parameter cube

overwrite: [bool] :

Overwrite parameter .fits cube if it exists?

`kwargs` are passed to :class:`pyspeckit.Spectrum.specfit` :

pyspeckit.wrappers.fit_gaussians_to_simple_spectra.**fit_gaussians_to_simple_spectra**(*filename*, *unit='km/s'*, *doplot=True*, *baseline=True*, *plotresiduals=False*, *figuresavename=None*, *croprange=None*, *savename=None*, ***kwargs*)

As stated in the name title, will fit Gaussians to simple spectra!

kwargs will be passed to specfit

***figuresavename* [None | string]**

After fitting, save the figure to this filename if specified

***croprange* [list of 2 floats]**

Crop the spectrum to (min,max) in the specified units

***savename* [None | string]**

After fitting, save the spectrum to this filename

Note that this wrapper can be used from the command line:

python fit_gaussians_to_simple_spectra.py spectrum.fits

6.2 NH3 fitter wrapper

Wrapper to fit ammonia spectra. Generates a reasonable guess at the position and velocity using a gaussian fit

Example use:

```
import pyspeckit
sp11 = pyspeckit.Spectrum('spec.nh3_11.dat', errorcol=999)
sp22 = pyspeckit.Spectrum('spec.nh3_22.dat', errorcol=999)
sp33 = pyspeckit.Spectrum('spec.nh3_33.dat', errorcol=999)
sp11.xarr.refX = pyspeckit.spectrum.models.ammonia.freq_dict['oneone']
sp22.xarr.refX = pyspeckit.spectrum.models.ammonia.freq_dict['twotwo']
sp33.xarr.refX = pyspeckit.spectrum.models.ammonia.freq_dict['threethree']
input_dict={'oneone':sp11, 'twotwo':sp22, 'threethree':sp33}
spf = pyspeckit.wrappers.fitnh3.fitnh3tkin(input_dict)
```

Note that if you want to use the plotter wrapper with cubes, you need to do something like the following, where the `plot_special` method of the stacked `cubes` object is set to the `plotter_override` function defined in the fitnh3_wrapper code:

```
cubes.plot_special = pyspeckit.wrappers.fitnh3.plotter_override
cubes.plot_special_kwargs = {'fignum':3, 'vrange':[55,135]}
cubes.plot_spectrum(160,99)
```

pyspeckit.wrappers.fitnh3.**BigSpectrum_to_NH3dict**(*sp*, *vrange=None*)

A rather complicated way to make the spdicts above given a spectrum...

pyspeckit.wrappers.fitnh3.**fitnh3**(*spectrum*, *vrange=[-100, 100]*, *vrangeunit='km/s'*, *quiet=False*, *Tex=20*, *trot=15*, *column=1000000000000000.0*, *fortho=1.0*, *tau=None*, *Tkin=None*, *fittype='ammonia'*, *spec_convert_kwargs={}*)

pyspeckit.wrappers.fitnh3.**fitnh3tkin**(*input_dict*, *dobaseline=True*, *baselinekwargs={}*, *crop=False*, *cropunit=None*, *guessline='twotwo'*, *tex=15*, *trot=20*, *column=15.0*, *fortho=0.66*, *tau=None*, *thin=False*, *quiet=False*, *doplot=True*, *fignum=1*, *guessfignum=2*, *smooth=False*, *scale_keyword=None*, *rebase=False*, *tkin=None*, *npeaks=1*, *guesses=None*, *fittype='ammonia'*, *guess_error=True*, *plotter_wrapper_kwargs={}*, ***kwargs*)

Given a dictionary of filenames and lines, fit them together e.g. {'oneone':'G000.000+00.000_nh3_11.fits'}

Parameters

input_dict : dict

A dictionary in which the keys are the ammonia line names (e.g., 'oneone', 'twotwo', etc) and the values are either Spectrum objects or filenames of spectra

dobaseline : bool

Fit and subtract a baseline prior to fitting the model? Keyword arguments to `pyspeckit.spectrum.Spectrum.baseline` are specified in `baselinekwargs`.

baselinekwargs : dict

The keyword arguments for the baseline

crop : bool or tuple

A range of values to crop the spectrum to. The units are specified by `cropunit`; the default `None` will use pixels. If False, no cropping will be performed.

cropunit : None or astropy unit

The unit for the crop parameter

guess_error : bool

Use the guess line to estimate the error in all spectra?

plotter_wrapper_kwargs : dict

Keyword arguments to pass to the plotter

fittype: 'ammonia' or 'cold_ammonia' :

The fitter model to use. This is overridden if `tau` is specified, in which case one of the `ammonia_tau` models is used (see source code)

pyspeckit.wrappers.fitnh3.**make_axdict**(*splist*, *spdict*)

pyspeckit.wrappers.fitnh3.**plot_nh3**(*spdict*, *spectra*, *fignum=1*, *show_components=False*, *residfignum=None*, *show_hyperfine_components=True*, *annotate=True*, *axdict=None*, *figure=None*, ***plotkwargs*)

Plot the results from a multi-nh3 fit

spdict needs to be dictionary with form:

'oneone': spectrum, 'twotwo': spectrum, etc.

pyspeckit.wrappers.fitnh3.**plotter_override**(*sp*, *vrange=None*, ***kwargs*)

Do plot_nh3 with syntax similar to plotter()

6.3 N2H+ fitter wrapper

Wrapper to fit N2H+ using RADEX models. This is meant to be used from the command line, e.g.

```
python n2hp_wrapper.py file.fits
```

and therefore has no independently defined functions.

pyspeckit.wrappers.n2hp_wrapper.**make_n2hp_fitter**(*path_to_radex='/Users/adam/work/n2hp/'*, *fileprefix='1-2_T=5to55_lvg'*)

Create a n2hp fitter using RADEX data cubes. The following files must exist:

```
path_to_radex+fileprefix+'_tex1.fits'
path_to_radex+fileprefix+'_tau1.fits'
path_to_radex+fileprefix+'_tex2.fits'
path_to_radex+fileprefix+'_tau2.fits'
```

e.g. `/Users/adam/work/n2hp/1-2_T=5to55_lvg_tau1.fits`

6.4 N2H+ extras

In place of the actual contents of N2H+ fitter, here are the modules used to make the wrapper

class pyspeckit.spectrum.models.model.**SpectralModel**(*modelfunc*, *npars*, *shortvarnames=('A', '\\Delta x', '\\sigma')*, *fitunit=None*, *centroid_par=None*, *fwhm_func=None*, *fwhm_pars=None*, *integral_func=None*, *use_lmfit=False*, *guess_types=('amplitude', 'center', 'width')*, ***kwargs*)

A wrapper class for a spectra model. Includes internal functions to generate multi-component models, annotations, integrals, and individual components. The declaration can be complex, since you should name individual variables, set limits on them, set the units the fit will be performed in, and set the annotations to be used. Check out some of the hyperfine codes (hcn, n2hp) for examples.

Spectral Model Initialization

Create a Spectral Model class for data fitting

Parameters

modelfunc : function

the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input followed by input parameters. Returns an array with the same shape as the input X-axis

npars : int

number of parameters required by the model

use_lmfit: bool :

Use lmfit instead of mpfit to do the fitting

parnames : list (optional)

a list or tuple of the parameter names

parvalues : list (optional)

the initial guesses for the input parameters (defaults to ZEROS)

parlimits : list (optional)

the upper/lower limits for each variable (defaults to ZEROS)

parfixed : list (optional)

Can declare any variables to be fixed (defaults to ZEROS)

parerror : list (optional)

technically an output parameter. Specifying it here will have no effect. (defaults to ZEROS)

partied : list (optional)

not the past tense of party. Can declare, via text, that some parameters are tied to each other. Defaults to zeros like the others, but it's not clear if that's a sensible default

fitunit : str (optional)

convert X-axis to these units before passing to model

parsteps : list (optional)

minimum step size for each paremeter (defaults to ZEROS)

npeaks : list (optional)

default number of peaks to assume when fitting (can be overridden)

shortvarnames : list (optional)

TeX names of the variables to use when annotating

amplitude_types : tuple

A tuple listing the types of the different parameters when guessing. The valid values are 'amplitude', 'width', and 'center'. These are handled by parse_3par_guesses, which translate these into input guess lists for the fitter. For a "standard" 3-parameter Gaussian fitter, nothing changes, but for other models that have more than 3 parameters, some translation is needed.

Returns

A tuple containing (model best-fit parameters, the model, parameter :

errors, chi^2 value) :

n2hp.**n2hp_radex**(*density=4*, *column=13*, *xoff_v=0.0*, *width=1.0*, *grid_vwidth=1.0*, *grid_vwidth_scale=False*, *texgrid=None*, *taugrid=None*, *hdr=None*, *path_to_texgrid=''*, *path_to_taugrid=''*, *temperature_gridnumber=3*, *debug=False*, *verbose=False*, ***kwargs*)

Use a grid of RADEX-computed models to make a model line spectrum

The RADEX models have to be available somewhere. OR they can be passed as arrays. If as arrays, the form should be: texgrid = ((minfreq1,maxfreq1,texgrid1),(minfreq2,maxfreq2,texgrid2))

xarr must be a SpectroscopicAxis instance xoff_v, width are both in km/s

grid_vwidth is the velocity assumed when computing the grid in km/s this is important because tau = modeltau / width (see, e.g., Draine 2011 textbook pgs 219-230) grid_vwidth_scale is True or False: False for LVG, True for Sphere

CHAPTER

SEVEN

EXAMPLES

Check out the flickr gallery.

Want your image or example included? E-mail us.

7.1 Creating a Spectrum from scratch

This example shows the initialization of a pyspeckit object from numpy arrays.

```
import numpy as np
import pyspeckit

xaxis = np.linspace(-50,150,100.)
sigma = 10.
center = 50.
synth_data = np.exp(-(xaxis-center)**2/(sigma**2 * 2.))

# Add noise
stddev = 0.1
noise = np.random.randn(xaxis.size)*stddev
error = stddev*np.ones_like(synth_data)
data = noise+synth_data

# this will give a "blank header" warning, which is fine
sp = pyspeckit.Spectrum(data=data, error=error, xarr=xaxis,
                        xarrkwargs={'unit':'km/s'},
                        unit='erg/s/cm^2/AA')

sp.plotter()

# Fit with automatic guesses
sp.specfit(fittype='gaussian')

# Fit with input guesses
# The guesses initialize the fitter
# This approach uses the 0th, 1st, and 2nd moments
amplitude_guess = data.max()
center_guess = (data*xaxis).sum()/data.sum()
width_guess = (data.sum() / amplitude_guess / (2*np.pi))**0.5
guesses = [amplitude_guess, center_guess, width_guess]
```

(continues on next page)

(continued from previous page)

```
sp.specfit(fittype='gaussian', guesses=guesses)

sp.plotter(errstyle='fill')
sp.specfit.plot_fit()
```

7.2 Fitting a continuum model as a model

This example shows the initialization of a pyspeckit object from numpy arrays, as in *Creating a Spectrum from scratch*, but it adds the twist of including a steep continuum.

We fit the continuum using the polynomial continuum model, which gives access to the error on the polynomial fit parameters. No such parameters are accessible via the `pyspeckit.Spectrum.baseline` tools because they use `numpy.poly1d` to fit the data.

```
import numpy as np
import pyspeckit

xaxis = np.linspace(-50,150,100)
sigma = 10.
center = 50.

baseline = np.poly1d([0.1, 0.25])(xaxis)

synth_data = np.exp(-(xaxis-center)**2/(sigma**2 * 2.)) + baseline

# Add noise
stddev = 0.1
noise = np.random.randn(xaxis.size)*stddev
error = stddev*np.ones_like(synth_data)
data = noise+synth_data

# this will give a "blank header" warning, which is fine
sp = pyspeckit.Spectrum(data=data, error=error, xarr=xaxis,
                        xarrkwargs={'unit':'km/s'},
                        unit='erg/s/cm^2/AA')

sp.plotter()

sp.specfit.Registry.add_fitter('polycontinuum',
                               pyspeckit.models.polynomial_continuum.poly_fitter(),
                               2)

sp.specfit(fittype='polycontinuum', guesses=(0,0), exclude=[30, 70])

# subtract the model fit to create a new spectrum
sp_contsub = sp.copy()
sp_contsub.data -= sp.specfit.get_full_model()
```

(continues on next page)

(continued from previous page)

```
sp_contsub.plotter()

# Fit with automatic guesses
sp_contsub.specfit(fittype='gaussian')

# Fit with input guesses
# The guesses initialize the fitter
# This approach uses the 0th, 1st, and 2nd moments
data = sp_contsub.data
amplitude_guess = data.max()
center_guess = (data*xaxis).sum()/data.sum()
width_guess = (data.sum() / amplitude_guess / (2*np.pi))**0.5
guesses = [amplitude_guess, center_guess, width_guess]
sp_contsub.specfit(fittype='gaussian', guesses=guesses)

sp_contsub.plotter(errstyle='fill')
sp_contsub.specfit.plot_fit()
```

7.3 Cube Fitting: a lightweight gaussian example

```
import pyspeckit
from pyspeckit.cubes.tests.test_cubetools import make_test_cube
import matplotlib.pylab as plt
import numpy as np

# generate a test spectral cube (10x10, with a 100 spectral channels)
make_test_cube((100,10,10), outfile='test.fits')
spc = pyspeckit.Cube('test.fits')

# do a crude noise estimate on the 30 edge channels
rmsmap = np.vstack([spc.cube[:15], spc.cube[85:]]).std(axis=0)

# get a cube of moments
spc.momenteach(vheight=False)

# fit each pixel taking its moment as an initial guess
spc.fiteach(fittype = 'gaussian',
            guesses = spc.momentcube,
            errmap = rmsmap,
            signal_cut = 3, # ignore pixels with SNR<3
            blank_value = np.nan,
            start_from_point=(5,5))
spc.mapplot()
# show the fitted amplitude
spc.show_fit_param(0, cmap='viridis')
plt.show()
```

7.4 Cube Fitting: simple, from scratch example

```
"""
simple cube fitting example
"""
from __future__ import print_function

import numpy as np
import pyspeckit
from spectral_cube import SpectralCube
from astropy import wcs
from astropy import units as u

# Create a new WCS object so we can instantiate the SpectralCube
mywcs = wcs.WCS(naxis=3)

# Set up a tangent projection
# You would normally read this from a file!!
mywcs.wcs.crpix = [1,2,21.0]
mywcs.wcs.cdelt = np.array([-0.066667, 0.066667, 500])
mywcs.wcs.crval = [290.9250, 14.5092, 60000]
mywcs.wcs.ctype = ["RA---TAN", "DEC--TAN", 'VELO']
mywcs.wcs.cunit = ['deg', 'deg', 'm/s']

# Create a synthetic X-dimension in km/s
xarr = np.linspace(50, 70, 41) # km/s

# Define a line width, which will vary across our image
# It will increase from 1 km/s to 4 km/s over the X-direction (RA)
sigma = np.outer(np.linspace(1,1.5,2), np.ones(4)).T

# Define a line center, which will vary in the opposite direction,
# along increasing Y-direction (declination)
centroid = np.outer(np.ones(2), np.linspace(58, 62, 4)).T

data = np.exp(-(np.tile(xarr, (2, 4, 1)).T - centroid)**2 / (2.*sigma**2))
cube = SpectralCube(data=data, wcs=mywcs)

# Sanity checks: do the moments accurately recover the inputs?
assert (np.abs(cube.moment1().to(u.km/u.s).value - centroid).max()) < 1e-5
assert (np.abs(cube.moment2().to(u.km**2/u.s**2).value - sigma**2).max()) < 1e-5

# Create a pyspeckit cube
pcube = pyspeckit.Cube(cube=cube)

# For convenience, convert the X-axis to km/s
# (WCSLIB automatically converts to m/s even if you give it km/s)
pcube.xarr.convert_to_unit(u.km/u.s)
```

(continues on next page)

(continued from previous page)

```
# Set up the fitter by doing a preliminary fit
pcube.specfit(fittype='gaussian', guesses='moments')

# Fit each spectrum with a gaussian
# First, assemble the guesses:
guesses = np.array([cube.max(axis=0).value,
                    cube.moment1(axis=0).to(u.km/u.s).value,
                    (cube.moment2(axis=0)**0.5).to(u.km/u.s).value])
# (the second moment is in m^2/s^2, but we want km/s

# Do the fit!
pcube.fiteach(guesses=guesses, # pass in the guess array
              # tell it where to start the fitting (center pixel in this case)
              start_from_point=(5,5),
              # Paralellize the fits?
              multicore=4,
              fittype='gaussian',
             )

# Then you can access the fits via parcube:
assert np.all(pcube.parcube[0,:,:] == 1)
assert np.all(pcube.parcube[1,:,:] == centroid)
assert np.all(pcube.parcube[2,:,:] == sigma)

# Can also fit non-parallelized
pcube.fiteach(guesses=guesses, # pass in the guess array
              # tell it where to start the fitting (center pixel in this case)
              start_from_point=(5,5),
              # Paralellize the fits?
              multicore=1,
              fittype='gaussian',
             )

assert np.all(pcube.parcube[0,:,:] == 1)
assert np.all(pcube.parcube[1,:,:] == centroid)
assert np.all(pcube.parcube[2,:,:] == sigma)
```

7.5 Radio Fitting: H_2CO RADEX example

Because an LVG model grid is being used as the basis for the fitting in this example, there are fewer free parameters. If you want to create your own model grid, there is a set of tools for creating RADEX model grids (in parallel) at the agpy RADEX page. The model grids used below are available on the pyspeckit bitbucket download page.

```
import pyspeckit
import numpy as np
import astropy.io.fits as pyfits
```

(continues on next page)

(continued from previous page)

```
from pyspeckit.spectrum import models

# create the Formaldehyde Radex fitter
# This step cannot be easily generalized: the user needs to read in their own grids
texgrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_
↪greenscaled_tex1.fits')
taugrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_
↪greenscaled_tau1.fits')
texgrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_
↪greenscaled_tex2.fits')
taugrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_
↪greenscaled_tau2.fits')
hdr = pyfits.getheader('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_
↪greenscaled_tau2.fits')

# this deserves a lot of explanation:
# models.formaldehyde.formaldehyde_radex is the MODEL that we are going to fit
# models.model.SpectralModel is a wrapper to deal with parinfo, multiple peaks,
# and annotations
# all of the parameters after the first are passed to the model function
formaldehyde_radex_fitter = models.model.SpectralModel(
        models.formaldehyde.formaldehyde_radex, 4,
        parnames=['density','column','center','width'],
        parvalues=[4,12,0,1],
        parlimited=[(True,True), (True,True), (False,False), (True,False)],
        parlimits=[(1,8), (11,16), (0,0), (0,0)],
        parsteps=[0.01,0.01,0,0],
        fitunits='Hz',
        texgrid=((4,5,texgrid1),(14,15,texgrid2)), # specify the frequency range over which␣
↪the grid is valid (in GHz)
        taugrid=((4,5,taugrid1),(14,15,taugrid2)),
        hdr=hdr,
        shortvarnames=("n","N","v","\\sigma"), # specify the parameter names (TeX is OK)
        grid_vwidth_scale=False,
        )

# sphere version:
texgrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_
↪sphere_tex1.fits')
taugrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_
↪sphere_tau1.fits')
texgrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_
↪sphere_tex2.fits')
taugrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_
↪sphere_tau2.fits')
hdr = pyfits.getheader('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_sphere_
↪tau2.fits')

formaldehyde_radex_fitter_sphere = models.model.SpectralModel(
        models.formaldehyde.formaldehyde_radex, 4,
        parnames=['density','column','center','width'],
        parvalues=[4,12,0,1],
```

(continues on next page)

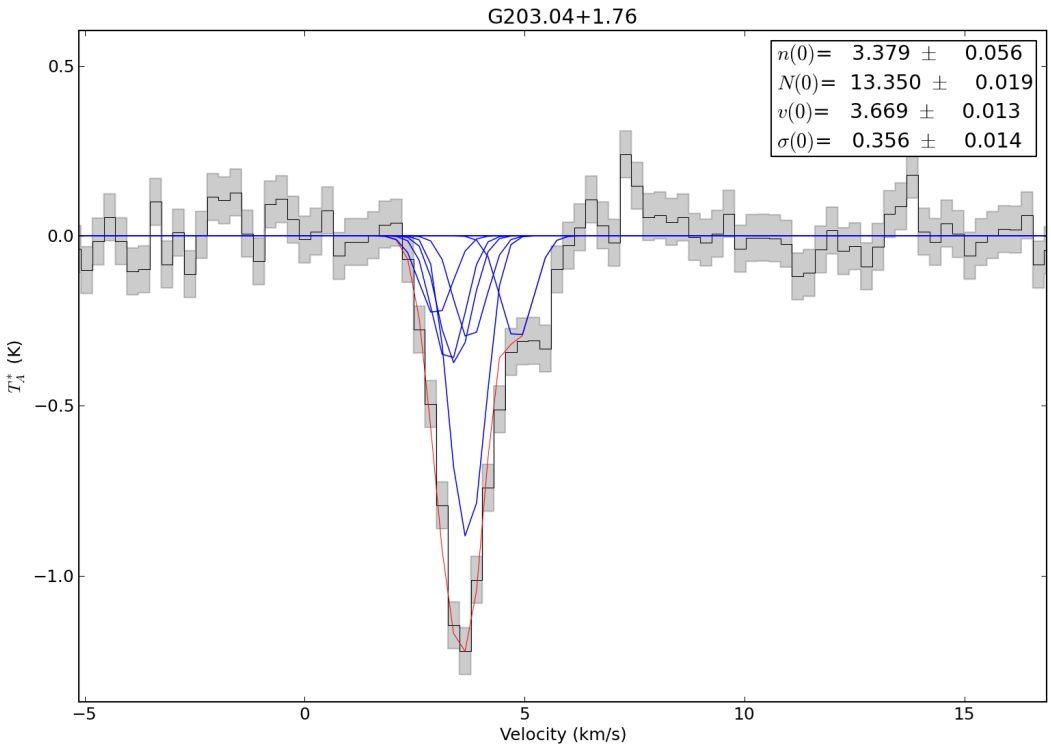
(continued from previous page)

```
        parlimited=[(True,True), (True,True), (False,False), (True,False)],
        parlimits=[(1,8), (11,16), (0,0), (0,0)],
        parsteps=[0.01,0.01,0,0],
        fitunits='Hz',
        texgrid=((4,5,texgrid1),(14,15,texgrid2)),
        taugrid=((4,5,taugrid1),(14,15,taugrid2)),
        hdr=hdr,
        shortvarnames=("n","N","v","\\sigma"),
        grid_vwidth_scale=True,
        )



sp1 = pyspeckit.Spectrum('G203.04+1.76_h2co.fits',wcstype='D',scale_keyword='ETAMB')
sp2 = pyspeckit.Spectrum('G203.04+1.76_h2co_Tastar.fits',wcstype='V',scale_keyword='ETAMB')


sp1.crop(-50,50)
sp1.smooth(3) # match to GBT resolution
sp2.crop(-50,50)

sp1.xarr.convert_to_unit('GHz')
sp1.specfit() # determine errors
sp1.error = np.ones(sp1.data.shape)*sp1.specfit.residuals.std()
sp1.baseline(excludefit=True)
sp2.xarr.convert_to_unit('GHz')
sp2.specfit() # determine errors
sp2.error = np.ones(sp2.data.shape)*sp2.specfit.residuals.std()
sp2.baseline(excludefit=True)
sp = pyspeckit.Spectra([sp1,sp2])

sp.Registry.add_fitter('formaldehyde_radex',
        formaldehyde_radex_fitter,4)
sp.Registry.add_fitter('formaldehyde_radex_sphere',
        formaldehyde_radex_fitter_sphere,4)

sp.plotter()
sp.specfit(fittype='formaldehyde_radex',multifit=None,guesses=[4,12,3.75,0.43],quiet=False)

# these are just for pretty plotting:
sp1.specfit.fitter = sp.specfit.fitter
sp1.specfit.modelpars = sp.specfit.modelpars
sp1.specfit.model = np.interp(sp1.xarr,sp.xarr,sp.specfit.model)
sp2.specfit.fitter = sp.specfit.fitter
sp2.specfit.modelpars = sp.specfit.modelpars
sp2.specfit.model = np.interp(sp2.xarr,sp.xarr,sp.specfit.model)

# previously, xarrs were in GHz to match the fitting scheme
sp1.xarr.convert_to_unit('km/s')
sp2.xarr.convert_to_unit('km/s')

sp1.plotter(xmin=-5,xmax=15,errstyle='fill')
```

(continues on next page)

(continued from previous page)

```
sp1.specfit.plot_fit(show_components=True)
sp2.plotter(xmin=-5,xmax=15,errstyle='fill')
sp2.specfit.plot_fit(show_components=True)


sp.plotter(figure=5)
sp.specfit(fittype='formaldehyde_radex_sphere',multifit=None,guesses=[4,13,3.75,0.43],
↪quiet=False)

# these are just for pretty plotting:
sp1.specfit.fitter = sp.specfit.fitter
sp1.specfit.modelpars = sp.specfit.modelpars
sp1.specfit.model = np.interp(sp1.xarr.as_unit('GHz'),sp.xarr,sp.specfit.model)
sp2.specfit.fitter = sp.specfit.fitter
sp2.specfit.modelpars = sp.specfit.modelpars
sp2.specfit.model = np.interp(sp2.xarr.as_unit('GHz'),sp.xarr,sp.specfit.model)

sp1.plotter(xmin=-5,xmax=15,errstyle='fill',figure=6)
sp1.specfit.plot_fit(show_components=True)
sp2.plotter(xmin=-5,xmax=15,errstyle='fill',figure=7)
sp2.specfit.plot_fit(show_components=True)
```



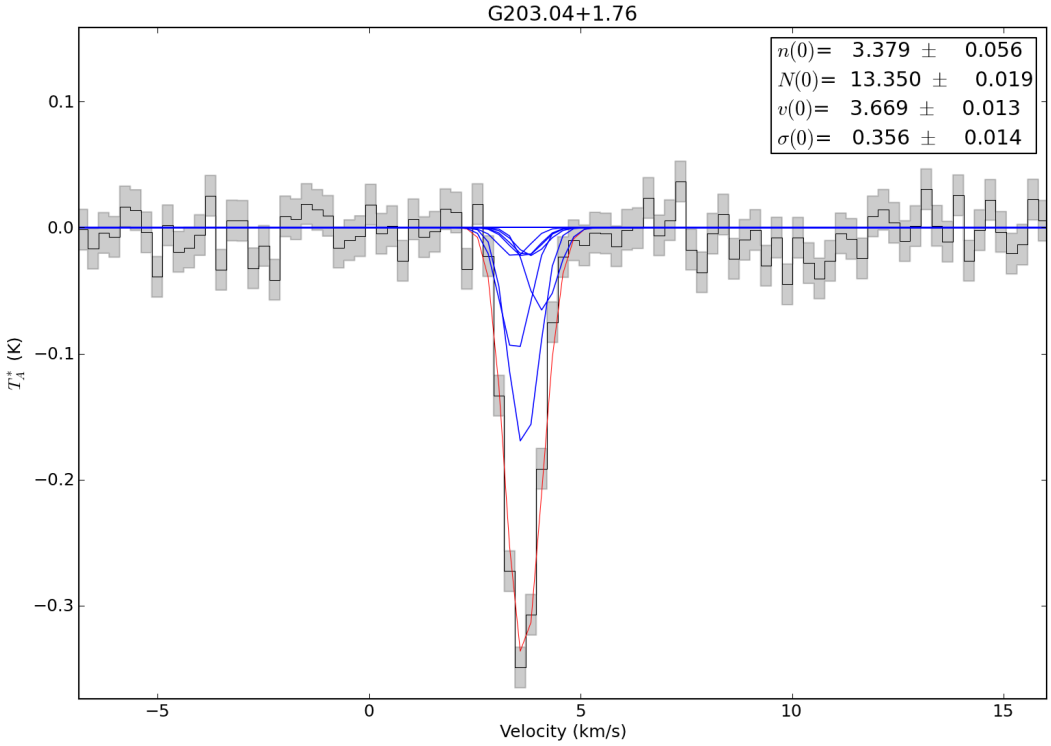


Fig. 1: Both transitions are fit simultaneously using a RADEX model. The input (fitted) parameters are therefore density, column density, width, and velocity.

7.6 Radio Fitting: H_2CO millimeter thermometer lines

Example hyperfine line fitting of a data cube for the H_2CO 303-202, 321-220, and 322-221 lines.

```
import pyspeckit
try:
    import astropy.io.fits as pyfits
except ImportError:
    import pyfits
from pyspeckit.spectrum import models

# create the Formaldehyde Radex fitter
# This step cannot be easily generalized: the user needs to read in their own grids
# Some of these grids can be acquired from:
# https://github.com/keflavich/radex_data_grids
texgrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_321-220_5kms_
↪temperature_para_tex1.fits')
taugrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_321-220_5kms_
↪temperature_para_tau1.fits')
texgrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_321-220_5kms_
↪temperature_para_tex2.fits')
taugrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_321-220_5kms_
↪temperature_para_tau2.fits')
hdr = pyfits.getheader('/Users/adam/work/h2co/radex/thermom/303-202_321-220_5kms_temperature_
↪para_tau2.fits')

texgrid1b = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_322-221_5kms_
↪temperature_para_tex1.fits')
taugrid1b = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_322-221_5kms_
↪temperature_para_tau1.fits')
texgrid2b = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_322-221_5kms_
↪temperature_para_tex2.fits')
taugrid2b = pyfits.getdata('/Users/adam/work/h2co/radex/thermom/303-202_322-221_5kms_
↪temperature_para_tau2.fits')
hdrb = pyfits.getheader('/Users/adam/work/h2co/radex/thermom/303-202_322-221_5kms_
↪temperature_para_tau2.fits')

# this deserves a lot of explanation:
# models.formaldehyde.formaldehyde_radex is the MODEL that we are going to fit
# models.model.SpectralModel is a wrapper to deal with parinfo, multiple peaks,
# and annotations
# all of the parameters after the first are passed to the model function
# This first one fits only the 303-202 and 322-221 lines
formaldehyde_radex_fitter_b = models.model.SpectralModel(
    models.formaldehyde_mm.formaldehyde_mm_radex, 5,
    parnames=['temperature','column','density','center','width'],
    parvalues=[50,12,4.5,0,1],
    parlimited=[(True,True), (True,True), (True,True), (False,False), (True,False)],
    parlimits=[(5,205), (10,17), (2,7), (0,0), (0,0)],
    parsteps=[0.01,0.01,0.1,0,0],
    fitunits='Hz',
    texgrid=((218.2,218.3,texgrid1b),(218.4,218.55,texgrid2b)), # specify the frequency␣
↪range over which the grid is valid (in GHz)
```

(continues on next page)

(continued from previous page)

```
    taugrid=((218.2,218.3,taugrid1b),(218.4,218.55,taugrid2b)),
    hdr=hdrb,
    shortvarnames=("T","N","n","v","\\sigma"), # specify the parameter names (TeX is OK)
    grid_vwidth=5.0,
)

# This second fitter fits only the 303-202 and 321-220 lines
formaldehyde_radex_fitter = models.model.SpectralModel(
    models.formaldehyde_mm.formaldehyde_mm_radex, 5,
    parnames=['temperature','column','density','center','width'],
    parvalues=[50,12,4.5,0,1],
    parlimited=[(True,True), (True,True), (True,True), (False,False), (True,False)],
    parlimits=[(5,205), (10,17), (2,7), (0,0), (0,0)],
    parsteps=[0.01,0.01,0.1,0,0],
    fitunits='Hz',
    texgrid=((218.2,218.3,texgrid1),(218.7,218.8,texgrid2)), # specify the frequency range␣
↪over which the grid is valid (in GHz)
    taugrid=((218.2,218.3,taugrid1),(218.7,218.8,taugrid2)),
    hdr=hdr,
    shortvarnames=("T","N","n","v","\\sigma"), # specify the parameter names (TeX is OK)
    grid_vwidth=5.0,
)

# This third fitter fits all three of the 303-202, 322-221, and 321-220 lines
# Since it has no additional free parameters, it's probably best...
formaldehyde_radex_fitter_both = models.model.SpectralModel(
    models.formaldehyde_mm.formaldehyde_mm_radex, 5,
    parnames=['temperature','column','density','center','width'],
    parvalues=[50,12,4.5,0,1],
    parlimited=[(True,True), (True,True), (True,True), (False,False), (True,False)],
    parlimits=[(5,205), (10,17), (2,7), (0,0), (0,0)],
    parsteps=[0.01,0.01,0.1,0,0],
    fitunits='Hz',
    texgrid=((218.2,218.3,texgrid1b),(218.4,218.55,texgrid2b),(218.7,218.8,texgrid2)), #␣
↪specify the frequency range over which the grid is valid (in GHz)
    taugrid=((218.2,218.3,taugrid1b),(218.4,218.55,taugrid2b),(218.7,218.8,taugrid2)),
    hdr=hdrb,
    shortvarnames=("T","N","n","v","\\sigma"), # specify the parameter names (TeX is OK)
    grid_vwidth=5.0,
)


if __name__ == "__main__":

    import pyspeckit.spectrum.readers.read_class
    sp = pyspeckit.readers.read_class.class_to_spectra('example_h2co_mm_spectrum.apex')
    sp.data *= 1/0.75 # T_A* -> T_MB
    sp.unit = "$T_{MB}$"
    # estimate the error from the data
    sp.error[:] = sp.stats((2.183e2,2.184e2))['std']

    # register the fitters
```

(continues on next page)

(continued from previous page)

```
    sp.Registry.add_fitter('formaldehyde_mm_radex',
                           formaldehyde_radex_fitter,5)
    sp.Registry.add_fitter('formaldehyde_mm_radex_b',
                           formaldehyde_radex_fitter_b,5)
    sp.Registry.add_fitter('formaldehyde_mm_radex_both',
                           formaldehyde_radex_fitter_both,5)

    # make 3 copies so that we can view independent fits
    # This step isn't really necessary, but it's a nice way to compare the fits
    # side-by-side
    sp1 = sp.copy()
    sp2 = sp.copy()
    sp3 = sp.copy()

    sp1.plotter(figure=1)
    sp1.specfit(fittype='formaldehyde_mm_radex',
                multifit=None,
                guesses=[100,13.2,4.5,0,7.0],
                limits=[(20,200),(11,15),(3,5.5),(-5,5),(2,15)],
                limited=[(True,True)]*5,
                fixed=[False,False,True,False,False],
                quiet=False,)
    sp1.plotter.savefig('h2co_mm_fit_303-202_321-220.png')

    sp2.plotter(figure=2)
    sp2.specfit(fittype='formaldehyde_mm_radex_b',
                multifit=None,
                guesses=[100,13.2,4.5,0,7.0],
                limits=[(20,200),(11,15),(3,5.5),(-5,5),(2,15)],
                limited=[(True,True)]*5,
                fixed=[False,False,True,False,False],
                quiet=False,)
    sp2.plotter.savefig('h2co_mm_fit_303-202_322-221.png')

    # Do two versions of the fit with different input guesses
    sp3.plotter(figure=3)
    sp3.specfit(fittype='formaldehyde_mm_radex_both',
                multifit=None,
                guesses=[95,13.2,4.5,0,7.0],
                limits=[(20,200),(11,15),(3,5.5),(-5,5),(2,15)],
                limited=[(True,True)]*5,
                fixed=[True,False,False,False,False],
                quiet=False,)
    sp3.plotter.savefig('h2co_mm_fit_303-202_322-221_and_303-202_321-220_try1.png')

    sp3.plotter(figure=4)
    sp3.specfit(fittype='formaldehyde_mm_radex_both',
                multifit=None,
                guesses=[105,13.2,4.5,0,7.0],
                limits=[(20,200),(11,15),(3,5.5),(-5,5),(2,15)],
                limited=[(True,True)]*5,
                fixed=[False,True,False,False,False],
```

(continues on next page)

(continued from previous page)

```
                quiet=False,)
    sp3.plotter.savefig('h2co_mm_fit_303-202_322-221_and_303-202_321-220_try2.png')

    # An illustration of the degeneracy between parameters
    sp3.plotter(figure=5)
    sp3.specfit.plot_model([95,13.5,4.75,2.89,6.85])
    sp3.specfit.plot_model([165,13.5,7.0,2.89,6.85],composite_fit_color='b')
    sp3.specfit.plot_model([117,13.15,5.25,2.89,6.85],composite_fit_color='g')
    sp3.plotter.savefig("h2co_mm_fit_degeneracy_example.png")
```

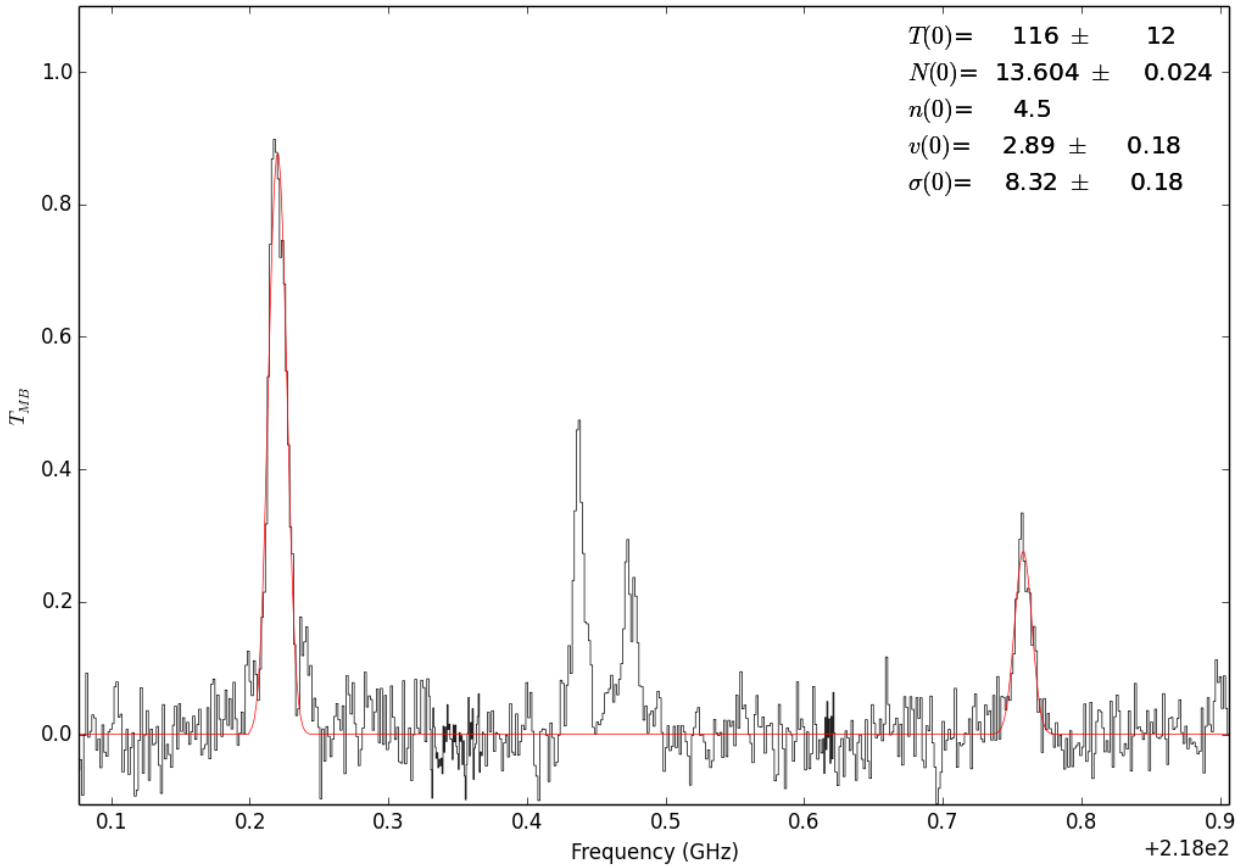


Fig. 2: Fit with the 302-202 and 321-220 lines

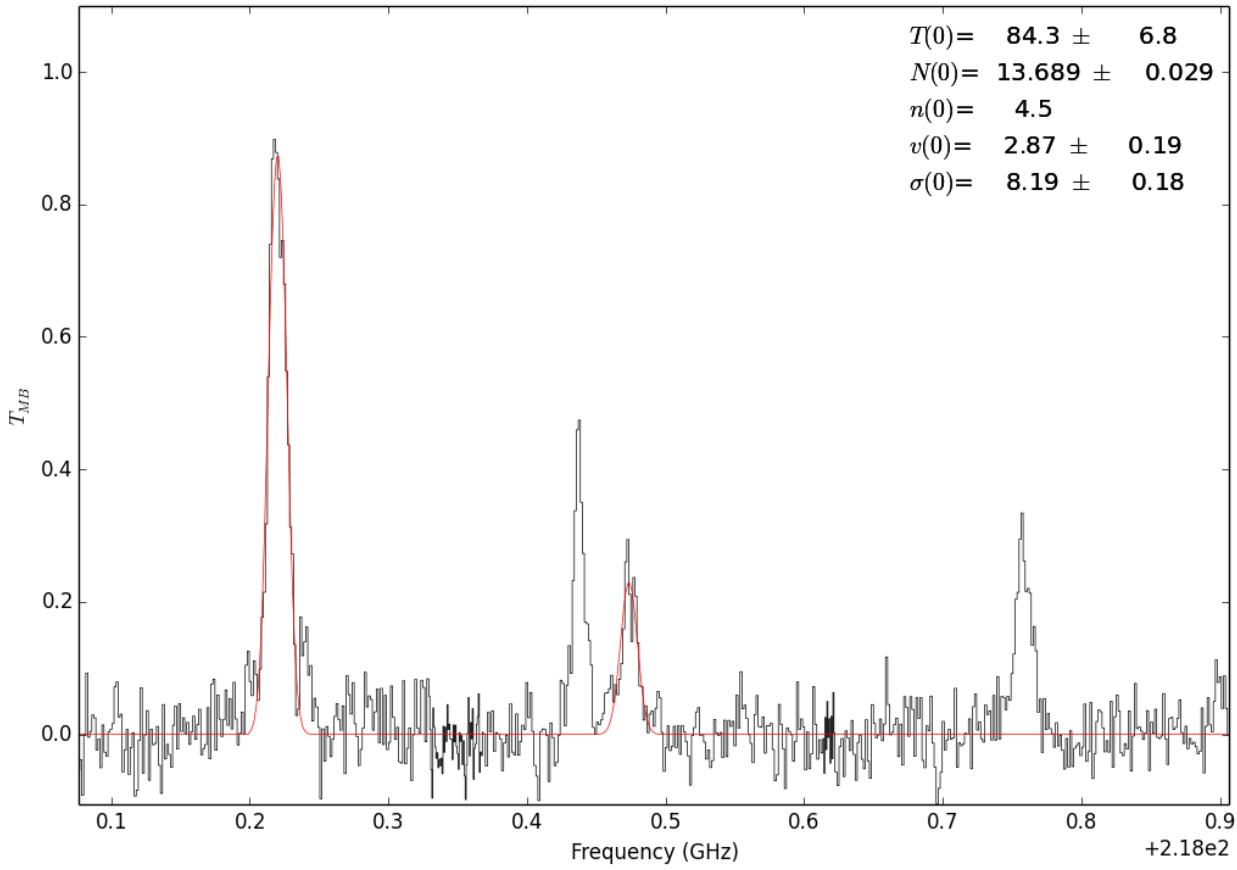


Fig. 3: Fit with the 302-202 and 321-220 lines

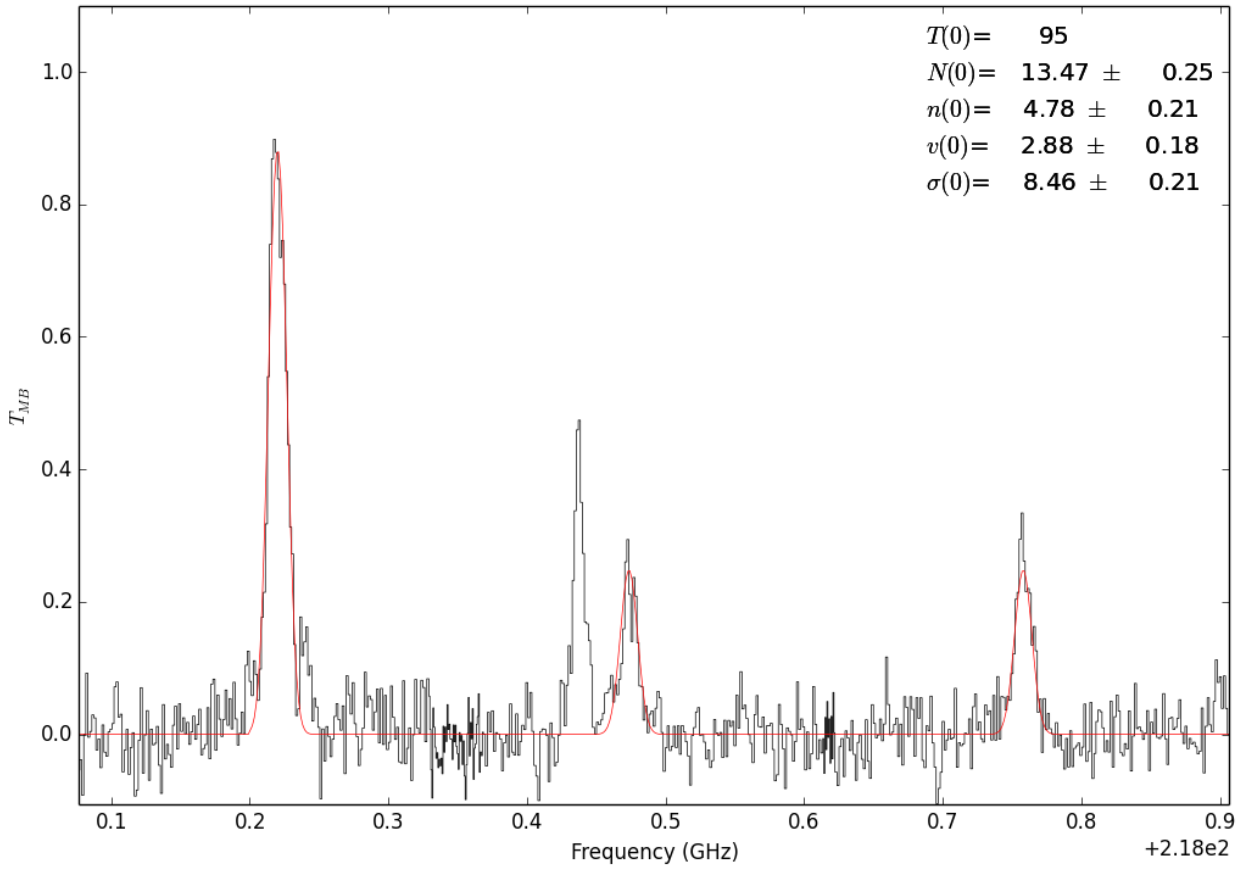


Fig. 4: Fit with all 3 lines. In this case, the temperature is unconstrained.

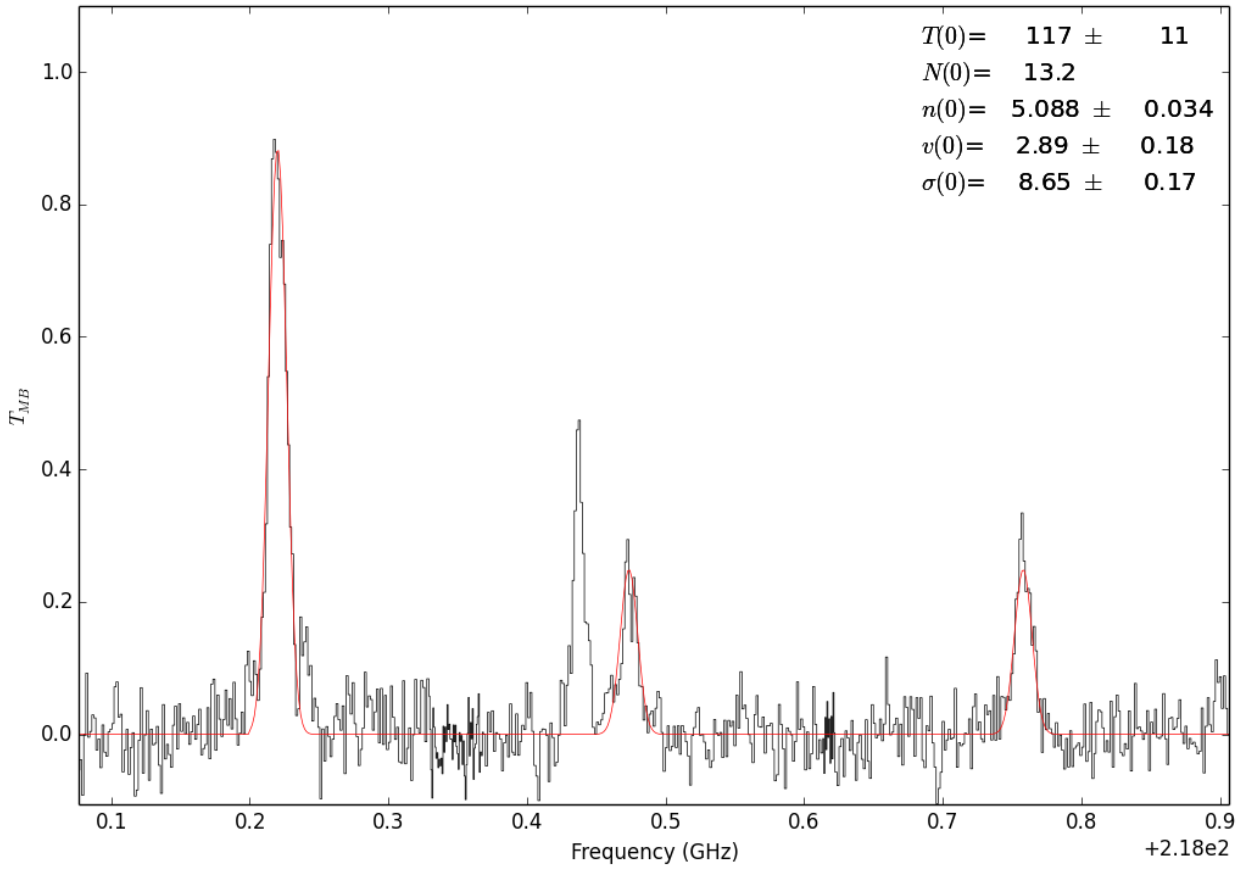


Fig. 5: Fit with all 3 lines. In this case, the column is unconstrained.

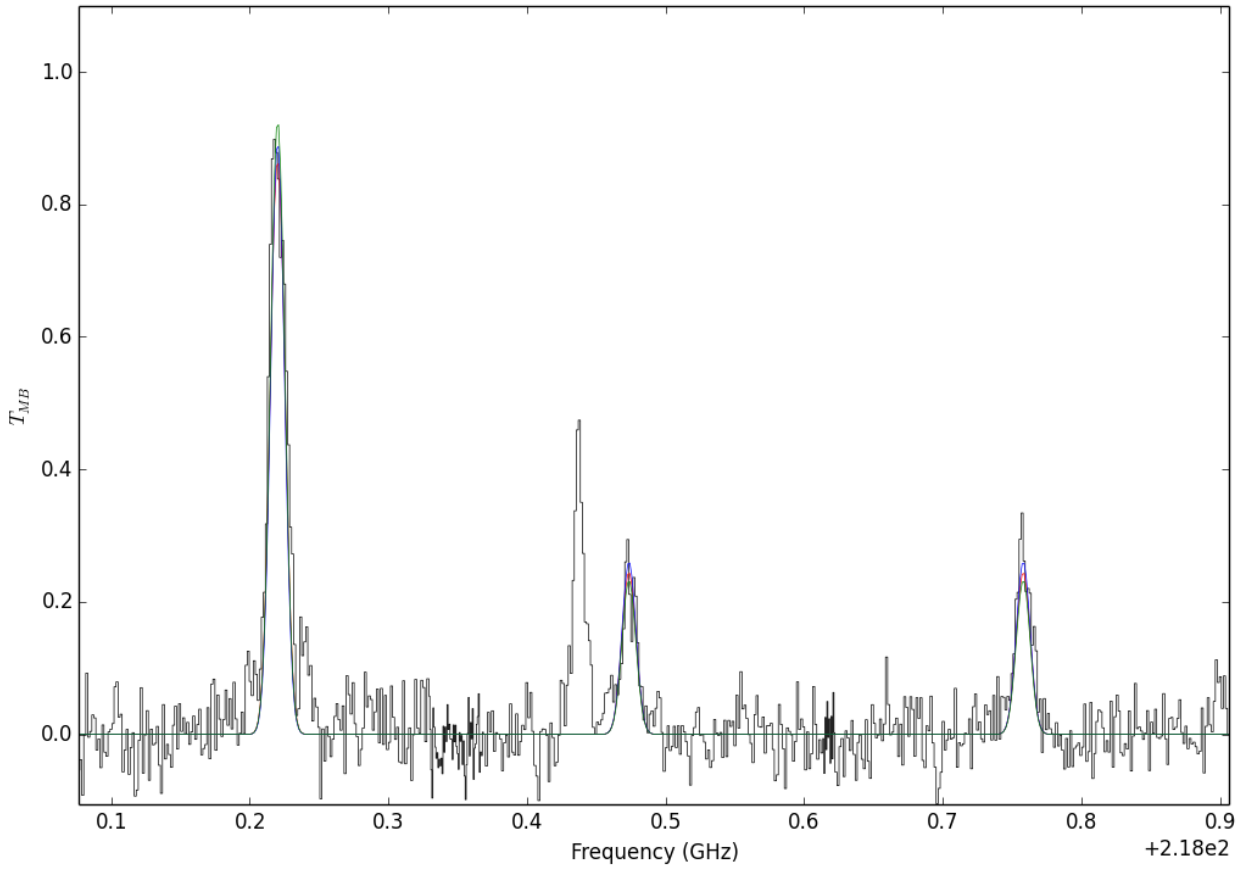


Fig. 6: A demonstration of degenerate parameters. The parameters used are:

- Blue: T,N,n = 95, 13.5, 4.75
- Green: T,N,n = 165, 13.5, 7.0
- Red: T,N,n = 117, 13.15, 5.25

7.7 Radio Fitting: NH_3 example

```
import pyspeckit

# The ammonia fitting wrapper requires a dictionary specifying the transition name
# (one of the four specified below) and the filename.  Alternately, you can have the
# dictionary values be pre-loaded Spectrum instances
filenames = {'oneone':'G032.751-00.071_nh3_11_Tastar.fits',
    'twotwo':'G032.751-00.071_nh3_22_Tastar.fits',
    'threethree':'G032.751-00.071_nh3_33_Tastar.fits',
    'fourfour':'G032.751-00.071_nh3_44_Tastar.fits'}

# Fit the ammonia spectrum with some reasonable initial guesses.  It is
# important to crop out extraneous junk and to smooth the data to make the
# fit proceed at a reasonable pace.
spdict1,spectra1 = pyspeckit.wrappers.fitnh3.fitnh3tkin(filenames,crop=[0,80],tkin=18.65,
↪tex=4.49,column=15.5,fortho=0.9,verbose=False,smooth=6)
```

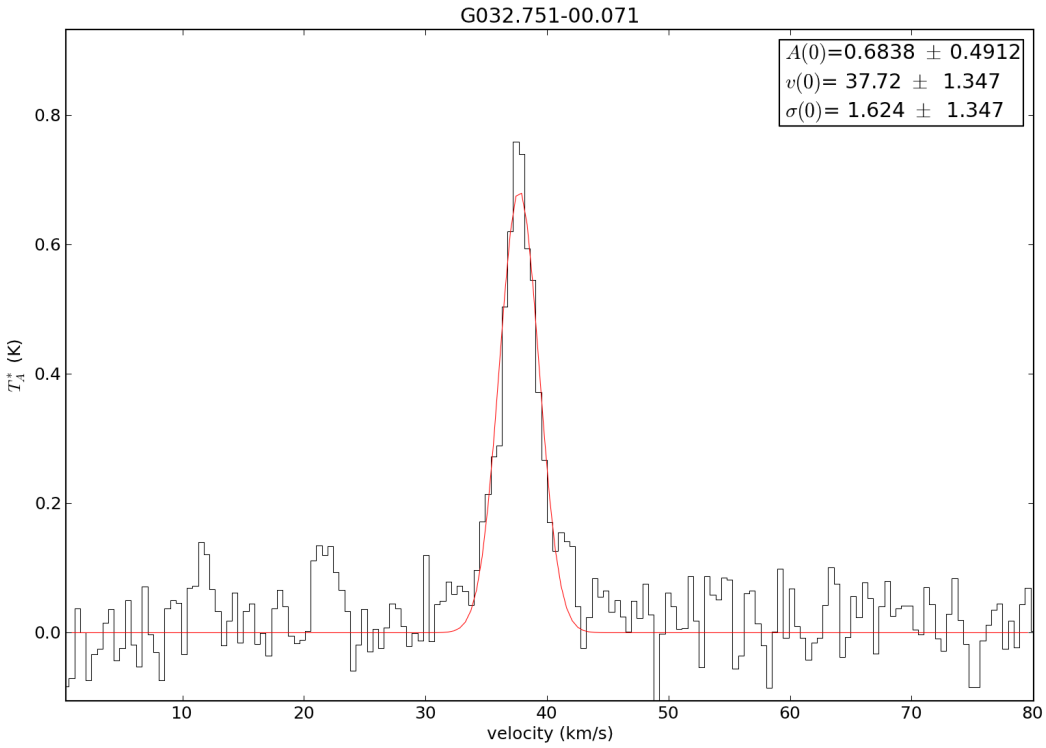


Fig. 7: The 2-2 transition is used to guess the central velocity and width via gaussian fitting because its hyperfine lines are weaker

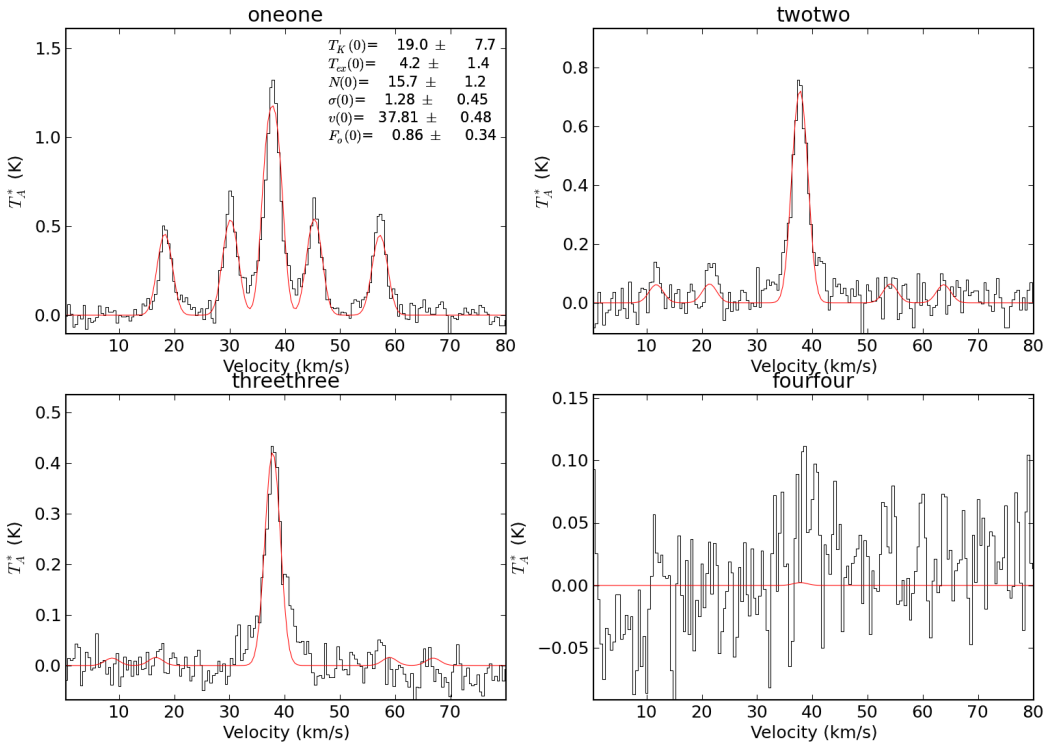


Fig. 8: Then all 4 lines are simultaneously fit. Even upper limits on the 4-4 line can provide helpful constraints on the model

7.8 Radio Fitting: NH_3 CUBE example

```
"""
Fit NH3 Cube
============

Example script to fit all pixels in an NH3 data cube.

This is a bit of a mess, and fairly complicated (intrinsically),
but if you have matched 1-1 + 2-2 + ... NH3 cubes, you should be
able to modify this example and get something useful out.

.. WARNING:: Cube fitting, particularly with a complicated line profile
ammonia, can take a long time.  Test this on a small cube first!

.. TODO:: Turn this example script into a function.  But customizing
    the fit parameters will still require digging into the data manually
    (e.g., excluding bad velocities, or excluding the hyperfine lines from
    the initial guess)
"""
import pyspeckit
import astropy
try:
    import astropy.io.fits as pyfits
except ImportError:
    import pyfits
import numpy as np
import os
from astropy.convolution import convolve_fft,Gaussian2DKernel

# set up CASA-like shortcuts
F=False; T=True

# Some optional parameters for the script
# (if False, it will try to load an already-stored version
# of the file)
fitcube = True

# Mask out low S/N pixels (to speed things up)
mask = pyfits.getdata('hotclump_11_mask.fits')
mask = np.isfinite(mask) * (mask > 0)

# Load the data using a mask
# Then calibrate the data (the data we're loading in this case are in Janskys,
# but we want surface brightness in Kelvin for the fitting process)
cube11 = pyspeckit.Cube('hotclump_11.cube_r0.5.image.fits', maskmap=mask)
cube11.cube *= (13.6 * (300.0 /
    (pyspeckit.spectrum.models.ammonia.freq_dict['oneone']/1e9))**2 *
    1./cube11.header.get('BMAJ')/3600. * 1./cube11.header.get('BMIN')/3600. )
cube11.unit = "K"
cube22 = pyspeckit.Cube('hotclump_22.cube_r0.5_contsub.image.fits', maskmap=mask)
cube22.cube *= (13.6 * (300.0 /
```

(continues on next page)

(continued from previous page)

```
        (pyspeckit.spectrum.models.ammonia.freq_dict['twotwo']/1e9))**2 *
        1./cube22.header.get('BMAJ')/3600. * 1./cube22.header.get('BMIN')/3600. )
cube22.unit = "K"
cube44 = pyspeckit.Cube('hotclump_44.cube_r0.5_contsub.image.fits', maskmap=mask)
cube44.cube *= (13.6 * (300.0 /
        (pyspeckit.spectrum.models.ammonia.freq_dict['fourfour']/1e9))**2 *
        1./cube44.header.get('BMAJ')/3600. * 1./cube44.header.get('BMIN')/3600. )
cube44.unit = "K"

# Compute an error map.  We use the 1-1 errors for all 3 because they're
# essentially the same, but you could use a different error map for each
# frequency
oneonemomentfn = 'hotclump_11.cube_r0.5_rerun.image.moment_linefree.fits'
errmap11 = (pyfits.getdata(oneonemomentfn).squeeze() * 13.6 *
            (300.0 /
             (pyspeckit.spectrum.models.ammonia.freq_dict['oneone']/1e9))**2
            * 1./cube11.header.get('BMAJ')/3600. *
            1./cube11.header.get('BMIN')/3600.)
# Interpolate errors across NaN pixels
errmap11[errmap11 != errmap11] = convolve_fft(errmap11,
                                              Gaussian2DKernel(3),
                                              interpolate_nan=True)[errmap11 != errmap11]

# Stack the cubes into one big cube.  The X-axis is no longer linear: there
# will be jumps from 1-1 to 2-2 to 4-4.
cubes = pyspeckit.CubeStack([cube11,cube22,cube44], maskmap=mask)
cubes.unit = "K"

# Make a "moment map" to contain the initial guesses
# If you've already fit the cube, just re-load the saved version
# otherwise, re-fit it
if os.path.exists('hot_momentcube.fits'):
    momentcubefile = pyfits.open('hot_momentcube.fits')
    momentcube = momentcubefile[0].data
else:
    cube11.mapplot()
    # compute the moment at each pixel
    cube11.momenteach()
    momentcube = cube11.momentcube
    momentcubefile = pyfits.PrimaryHDU(data=momentcube, header=cube11.header)
if astropy.version.major >= 2 or (astropy.version.major==1 and astropy.version.minor>=3):
    momentcubefile.writeto('hot_momentcube.fits',overwrite=True)
else:
    momentcubefile.writeto('hot_momentcube.fits',clobber=True)

# Create a "guess cube".  Because we're fitting physical parameters in this
# case, we want to make the initial guesses somewhat reasonable
# As above, we'll just reload the saved version if it exists
guessfn = 'hot_guesscube.fits'
if os.path.exists(guessfn):
    guesscube = pyfits.open(guessfn)
    guesses = guesscube[0].data
```

(continues on next page)

(continued from previous page)

```
else:
    guesses = np.zeros((6,)+cubes.cube.shape[1:])
    guesses[0,:,:] = 20                    # Kinetic temperature
    guesses[1,:,:] = 5                     # Excitation  Temp
    guesses[2,:,:] = 14.5                  # log(column)
    guesses[3,:,:] = momentcube[3,:,:] / 5 # Line width / 5 (the NH3 moment overestimates␣
↪linewidth)
    guesses[4,:,:] = momentcube[2,:,:]     # Line centroid
    guesses[5,:,:] = 0.5                   # F(ortho) - ortho NH3 fraction (fixed)

    guesscube = pyfits.PrimaryHDU(data=guesses, header=cube11.header)
    if astropy.version.major >= 2 or (astropy.version.major==1 and astropy.version.minor>=3):
        guesscube.writeto(guessfn, overwrite=True)
    else:
        guesscube.writeto(guessfn, clobber=True)

# This bit doesn't need to be in an if statment
if fitcube:
    # excise guesses that fall out of the "good" range
    guesses[4,:,:][guesses[4,:,:] > 100] = 100.0
    guesses[4,:,:][guesses[4,:,:] < 91] = 95

    # do the fits
    # signal_cut means ignore any pixel with peak S/N less than this number
    # In this fit, many of the parameters are limited
    # start_from_point selects the pixel coordinates to start from
    # use_nearest_as_guess says that, at each pixel, the input guesses will be
    # set by the fitted parameters from the nearest pixel with a good fit
    # HOWEVER, because this fitting is done in parallel (multicore=12 means
    # 12 parallel fitting processes will run), this actually means that EACH
    # core will have its own sub-set of the cube that it will search for good
    # fits. So if you REALLY want consistency, you need to do the fit in serial.
    cubes.fiteach(fittype='ammonia', multifit=None, guesses=guesses,
            integral=False, verbose_level=3, fixed=[F,F,F,F,F,T], signal_cut=3,
            limitedmax=[F,F,F,F,T,T],
            maxpars=[0,0,0,0,101,1],
            limitedmin=[T,T,F,F,T,T],
            minpars=[2.73,2.73,0,0,91,0],
            use_nearest_as_guess=True, start_from_point=(94,250),
            multicore=12,
            errmap=errmap11)

    # Save the fitted parameters in a data cube
    fitcubefile = pyfits.PrimaryHDU(data=np.concatenate([cubes.parcube,cubes.errcube]),␣
↪header=cubes.header)
    fitcubefile.header['PLANE1'] = 'TKIN'
    fitcubefile.header['PLANE2'] = 'TEX'
    fitcubefile.header['PLANE3'] = 'COLUMN'
    fitcubefile.header['PLANE4'] = 'SIGMA'
    fitcubefile.header['PLANE5'] = 'VELOCITY'
    fitcubefile.header['PLANE6'] = 'FORTHO'
    fitcubefile.header['PLANE7'] = 'eTKIN'
```

(continues on next page)

(continued from previous page)

```
    fitcubefile.header['PLANE8'] = 'eTEX'
    fitcubefile.header['PLANE9'] = 'eCOLUMN'
    fitcubefile.header['PLANE10'] = 'eSIGMA'
    fitcubefile.header['PLANE11'] = 'eVELOCITY'
    fitcubefile.header['PLANE12'] = 'eFORTHO'
    fitcubefile.header['CDELT3'] = 1
    fitcubefile.header['CTYPE3'] = 'FITPAR'
    fitcubefile.header['CRVAL3'] = 0
    fitcubefile.header['CRPIX3'] = 1
    fitcubefile.writeto("hot_fitcube_try6.fits")
else: # you can read in a fit you've already done!
    cubes.load_model_fit('hot_fitcube_try6.fits', 6, 'ammonia', _temp_fit_loc=(94,250))
    cubes.specfit.parinfo[5]['fixed'] = True


# Now do some plotting things
import pylab as pl

# Set the map-to-plot to be the line centroid
cubes.mapplot.plane = cubes.parcube[4,:,:]
cubes.mapplot(estimator=None,vmin=91,vmax=101)

# Set the reference frequency to be the 1-1 line frequency
cubes.xarr.refX = pyspeckit.spectrum.models.ammonia.freq_dict['oneone']
cubes.xarr.refX_unit='Hz'

# If you wanted to view the spectra in velocity units, use this:
#cubes.xarr.convert_to_unit('km/s')
#cubes.plotter.xmin=55
#cubes.plotter.xmax=135

# Now replace the cube's plotter with a "special" plotter
# The "special" plotter puts the 1-1, 2-2, and 4-4 lines in their own separate
# windows

cubes.plot_special = pyspeckit.wrappers.fitnh3.plotter_override
cubes.plot_special_kwargs = {'fignum':3, 'vrange':[55,135]}
cubes.plot_spectrum(160,99)

# make interactive
pl.ion()
pl.show()

# At this point, you can click on any pixel in the image and see the spectrum
# with the best-fit ammonia profile overlaid.
```

7.9 Radio fitting: NH3 multiple lines with independent tau, Tex

Example hyperfine line fitting for the NH3 1-1 through 3-3 lines, including making up a synthetic spectrum

```
import numpy as np
import pyspeckit
from astropy import units as u
from pyspeckit.spectrum.models import ammonia_constants, ammonia, ammonia_hf
from pyspeckit.spectrum.models.ammonia_constants import freq_dict
from pyspeckit.spectrum.units import SpectroscopicAxis, SpectroscopicAxes

# Step 1. Generate a synthetic spectrum.  Already have a real spectrum?  Skip
# to step 2!
# Generate a synthetic spectrum based off of 3 NH3 lines
# Note that they are converted to GHz first
xarr11 = SpectroscopicAxis(np.linspace(-30, 30, 100)*u.km/u.s,
                           velocity_convention='radio',
                           refX=freq_dict['oneone']).as_unit(u.GHz)
xarr22 = SpectroscopicAxis(np.linspace(-40, 40, 100)*u.km/u.s,
                           velocity_convention='radio',
                           refX=freq_dict['twotwo']).as_unit(u.GHz)
xarr33 = SpectroscopicAxis(np.linspace(-50, 50, 100)*u.km/u.s,
                           velocity_convention='radio',
                           refX=freq_dict['threethree']).as_unit(u.GHz)
# Merge the three X-axes into a single axis
xarr = SpectroscopicAxes([xarr11,xarr22,xarr33])

# Compute a synthetic model that is made of two temperature components with
# identical velocities
synthspec = (ammonia.ammonia(xarr, trot=20, ntot=15, fortho=0.5, xoff_v=0.0,
                             width=1.0) +
             ammonia.ammonia(xarr, trot=50, ntot=14, fortho=0.5, xoff_v=0.0,
                             width=1.0))

# Create the Spectrum object
spectrum = pyspeckit.Spectrum(xarr=xarr, data=synthspec, header={})


# Step 2.  You have a spectrum.
# plot it
spectrum.plotter()

# Use the multi-tex/multi-tau model generator to build up a model function
# You can use any set of oneone, twotwo, ..., eighteight (no 9-9 or higher)
# This sets the number of parameters to be fit: 2+2*(n_transitions)
fitter = ammonia_hf.nh3_vtau_multimodel_generator(['oneone', 'twotwo',
                                                   'threethree'])

# Register the fitter - i.e., tell pyspeckit where it is and how to use it
spectrum.specfit.Registry.add_fitter('nh3_vtau_123', fitter, fitter.npars)
# These are the parameter names, approximately:
# parnames=['center','width','Tex11','tau11','Tex22','tau22','Tex33','tau33'],
```

(continues on next page)

(continued from previous page)

```
# Need to give some input guesses.  We start with something wrong-ish: -5 km/s,
# 1.2 km/s width, and 15 K + 0.5 tau for all 3 lines
guesses = [-5, 1.2, 15, 0.5, 15, 0.5, 15, 0.5,]

# Plot up the guessed model
spectrum.plotter.axis.plot(spectrum.xarr,
                           fitter.n_modelfunc(guesses)(spectrum.xarr), 'b')

# Run the fit!
spectrum.specfit(fittype='nh3_vtau_123', guesses=guesses)

# display the correct and fitted answers
print("Low column version:")
def printthings(ammonia=ammonia.ammonia, xarr=xarr):
    print("Real optical depths of component 1: ",[ammonia(xarr, trot=20,
                                                          ntot=15, fortho=0.5,
                                                          xoff_v=0.0,
                                                          width=1.0,
                                                          return_tau=True)[x]
                                                   for x in ['oneone', 'twotwo',
                                                             'threethree']])
    print("Real optical depths of component 2: ",[ammonia(xarr, trot=50,
                                                          ntot=14, fortho=0.5,
                                                          xoff_v=0.0,
                                                          width=1.0,
                                                          return_tau=True)[x]
                                                   for x in ['oneone', 'twotwo',
                                                             'threethree']])
printthings()

print("Fitted parameters: ",spectrum.specfit.parinfo)

# It works, but the covariances between tex and tau are large.
# So, another example with higher tau (and therefore... less degenerate?)

synthspec = (ammonia.ammonia(xarr, trot=20, ntot=16, fortho=0.5, xoff_v=0.0,
                             width=1.0) +
             ammonia.ammonia(xarr, trot=50, ntot=15, fortho=0.5, xoff_v=0.0,
                             width=1.0))
spectrum2 = pyspeckit.Spectrum(xarr=xarr, data=synthspec, header={})
spectrum2.plotter()
spectrum2.specfit.Registry.add_fitter('nh3_vtau_123', fitter, fitter.npars)
spectrum2.specfit(fittype='nh3_vtau_123', guesses=guesses)

# We can also examine what tau really should have been... kinda.
print("High column version:")
def printthings(ammonia=ammonia.ammonia, xarr=xarr):
    print("Real optical depths of component 1: ",[ammonia(xarr, trot=20,
                                                          ntot=16, fortho=0.5,
                                                          xoff_v=0.0,
                                                          width=1.0,
                                                          return_tau=True)[x]
```

(continues on next page)

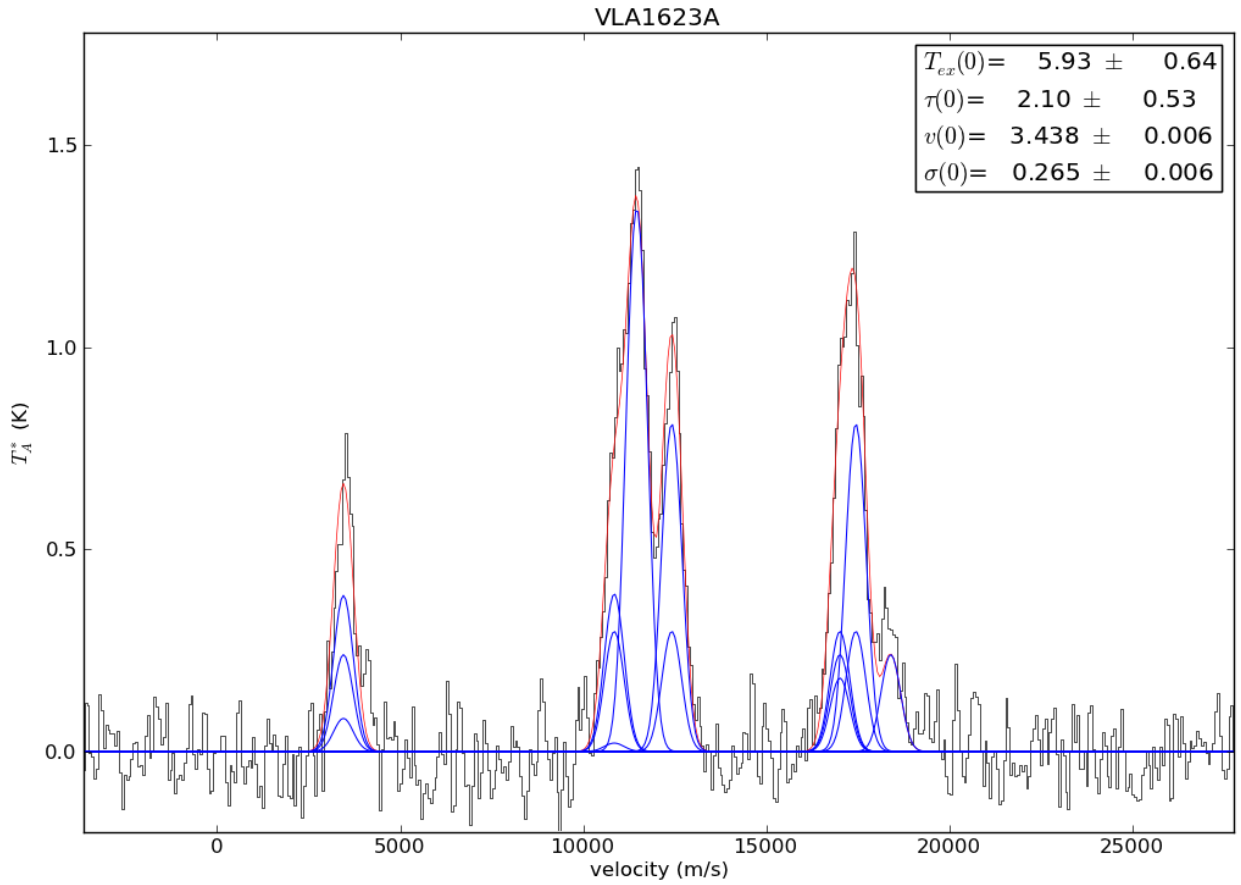
(continued from previous page)

```
                                                for x in ['oneone', 'twotwo',
                                                          'threethree']])
    print("Real optical depths of component 2: ",[ammonia(xarr, trot=50,
                                                           ntot=15, fortho=0.5,
                                                           xoff_v=0.0,
                                                           width=1.0,
                                                           return_tau=True)[x]
                                                for x in ['oneone', 'twotwo',
                                                          'threethree']])
printthings()
print("Fitted parameters: ",spectrum2.specfit.parinfo)
```

7.10 Radio Fitting: N_2H+ example

Example hyperfine line fitting for the N_2H+ 1-0 line.

```
import pyspeckit

# Load the spectrum
sp = pyspeckit.Spectrum('n2hp_opha_example.fits')

# Register the fitter
# The N2H+ fitter is 'built-in' but is not registered by default; this example
# shows how to register a fitting procedure
# 'multi' indicates that it is possible to fit multiple components and a
# background will not automatically be fit 4 is the number of parameters in the
# model (excitation temperature, optical depth, line center, and line width)
sp.Registry.add_fitter('n2hp_vtau',pyspeckit.models.n2hp.n2hp_vtau_fitter,4)
sp.xarr.velocity_convention = 'radio'
# Run the fitter
sp.specfit(fittype='n2hp_vtau',multifit=None,guesses=[15,2,4,0.2])

# Plot the results
sp.plotter()
# Re-run the fitter (to get proper error bars) and show the individual fit components
sp.specfit(fittype='n2hp_vtau', multifit=None, guesses=[15, 2, 4, 0.2],
           show_hyperfine_components=True)

# Save the figure (this step is just so that an image can be included on the web page)
sp.plotter.savefig('n2hp_ophA_fit.png')
```


VLA1623A
$T_{ex}(0)$= 5.93 ± 0.64
$\tau(0)$= 2.10 ± 0.53
$v(0)$= 3.438 ± 0.006
$\sigma(0)$= 0.265 ± 0.006
T_A^* (K)
1.5
1.0
0.5
0.0
0
5000
10000
15000
20000
25000
velocity (m/s)

7.11 Radio Fitting: N_2H+ cube example

Example hyperfine line fitting of a data cube for the N_2H+ 1-0 line.

```
import astropy
import pyspeckit
import os
import astropy.units as u
import warnings
from astropy import wcs

if not os.path.exists('n2hp_cube.fit'):
    import astropy.utils.data as aud
    from astropy.io import fits

    try:
        f = aud.download_file('ftp://cdsarc.u-strasbg.fr/pub/cats/J/A%2BA/472/519/fits/opha_
↪n2h.fit')
    except Exception as ex:
        # this might be any number of different timeout errors (urllib2.URLError, socket.
↪timeout, etc)
        # travis-ci can't handle ftp:
        # https://blog.travis-ci.com/2018-07-23-the-tale-of-ftp-at-travis-ci
        print("Failed to download from ftp.  Exception was: {0}".format(ex))
        f = aud.download_file('http://cdsarc.u-strasbg.fr/ftp/cats/J/A+A/472/519/fits/opha_
↪n2h.fit')

    with fits.open(f) as ff:
        ff[0].header['CUNIT3'] = 'm/s'
        for kw in ['CTYPE4','CRVAL4','CDELT4','CRPIX4','CROTA4']:
            if kw in ff[0].header:
                del ff[0].header[kw]
        ff.writeto('n2hp_cube.fit')

# Load the spectral cube cropped in the middle for efficiency
with warnings.catch_warnings():
    warnings.filterwarnings('ignore', category=wcs.FITSFixedWarning)
    spc = pyspeckit.Cube('n2hp_cube.fit')[:,25:29,12:16]
# Set the velocity convention: in the future, this may be read directly from
# the file, but for now it cannot be.
spc.xarr.refX = 93176265000.0*u.Hz
spc.xarr.velocity_convention = 'radio'
spc.xarr.convert_to_unit('km/s')

# Register the fitter
# The N2H+ fitter is 'built-in' but is not registered by default; this example
# shows how to register a fitting procedure
# 'multi' indicates that it is possible to fit multiple components and a
# background will not automatically be fit 4 is the number of parameters in the
# model (excitation temperature, optical depth, line center, and line width)
spc.Registry.add_fitter('n2hp_vtau', pyspeckit.models.n2hp.n2hp_vtau_fitter, 4)

# Get a measurement of the error per pixel
```

(continues on next page)

(continued from previous page)

```
errmap = spc.slice(20, 28, unit='km/s').cube.std(axis=0)

# A good way to write a cube fitter is to have it load from disk if the cube
# fit was completed successfully in the past
if os.path.exists('n2hp_fitted_parameters.fits'):
    spc.load_model_fit('n2hp_fitted_parameters.fits', npars=4, npeaks=1)
else:
    # Run the fitter
    # Estimated time to completion ~ 2 minutes
    spc.fiteach(fittype='n2hp_vtau',
                guesses=[5,0.5,3,1], # Tex=5K, tau=0.5, v_center=12, width=1 km/s
                signal_cut=3, # minimize the # of pixels fit for the example
                start_from_point=(2,2), # start at a pixel with signal
                errmap=errmap,
                use_neighbor_as_guess=True,
                multicore=4,
                )
    # There are a huge number of parameters for the fiteach procedure.  See:
    # http://pyspeckit.readthedocs.org/en/latest/example_nh3_cube.html
    # http://pyspeckit.readthedocs.org/en/latest/cubes.html?highlight=fiteach#pyspeckit.
↪cubes.SpectralCube.Cube.fiteach
    #
# Unfortunately, a complete tutorial on this stuff is on the to-do list;
# right now the use of many of these parameters is at a research level.
# However, pyspeckit@gmail.com will support them!  They are being used
# in current and pending publications

# Save the fitted parameters to a FITS file, and overwrite one if one exists
spc.write_fit('n2hp_fitted_parameters.fits', overwrite=True)

# Show an image of the fitted tex
spc.mapplot(estimator=1, vmin=0, vmax=10)
# you can click on any pixel to see its spectrum & fit

# plot one of the fitted spectra
spc.plot_spectrum(2, 2, plot_fit=True)
# spc.parcube[:,27,14] = [ 14.82569198,   1.77055642,   3.15740051,   0.16035407]
# Note that the optical depth is the "total" optical depth, which is
# distributed among 15 hyperfine components.  You can see this in
# pyspeckit.spectrum.models.n2hp.line_strength_dict
# As a sanity check, you can see that the brightest line has 0.259 of the total
# optical depth, so the peak line brightness is:
# (14.825-2.73) * (1-np.exp(-1.77 * 0.259)) = 4.45
# which matches the peak of 4.67 pretty well

# Show an image of the best-fit velocity
spc.mapplot.plane = spc.parcube[2,:,:]
spc.mapplot(estimator=None)

# running in script mode, the figures won't show by default on some systems
# import pylab as pl
# pl.draw()
```

(continues on next page)

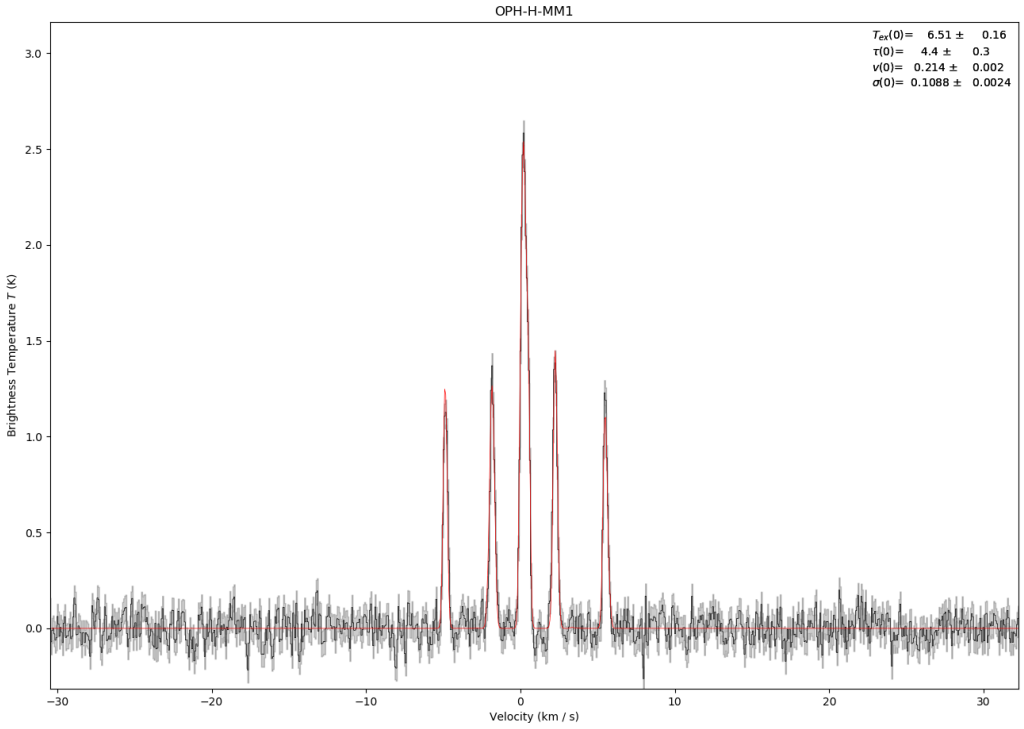
(continued from previous page)

```
# pl.show()
```

7.12 Radio Fitting: ortho-NH_2D example

Example hyperfine line fitting for the ortho-NH_2D line.

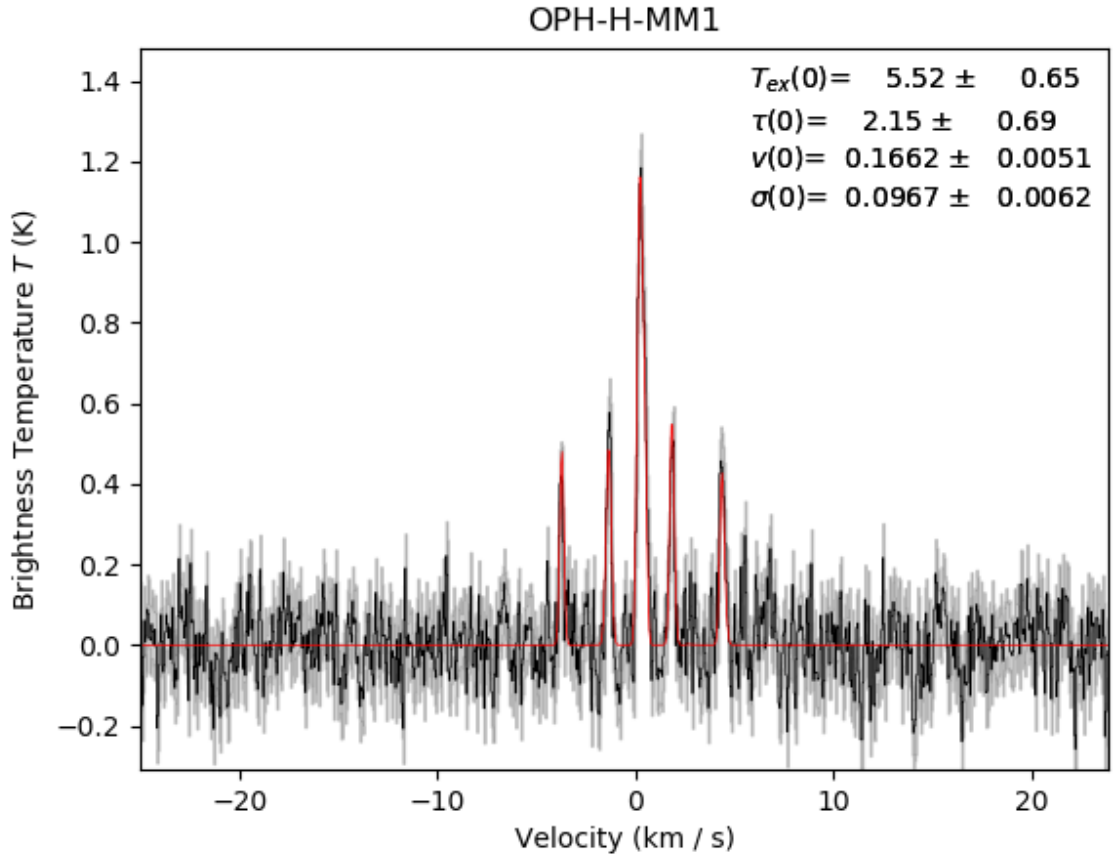
```
import pyspeckit
import os
from pyspeckit.spectrum.models import nh2d
import numpy as np

import astropy.units as u

if not os.path.exists('o-nh2d_spec.fits'):
    import astropy.utils.data as aud
    from astropy.io import fits
    f = aud.download_file('https://github.com/pyspeckit/pyspeckit-example-files/raw/master/o-
↪nh2d_spec.fits')
    with fits.open(f) as ff:
        ff.writeto('o-nh2d_spec.fits')

# Load the spectrum
spec = pyspeckit.Spectrum('o-nh2d_spec.fits')
# Determine rms from line free section and load into cube
rms = np.std(spec.data[10:370])
spec.error[:] = rms
# setup spectral axis
spec.xarr.refX = 85.92627*u.GHz
spec.xarr.velocity_convention = 'radio'
spec.xarr.convert_to_unit('km/s')
# define useful shortcuts for True and False
F=False
T=True
# Setup of matplotlib
import matplotlib.pyplot as plt
plt.ion()
# Add NH2D fitter
spec.Registry.add_fitter('nh2d_vtau', pyspeckit.models.nh2d.nh2d_vtau_fitter,4)
# run spectral fit using some reasonable guesses
spec.specfit(fittype='nh2d_vtau', guesses=[6.51, 4.4, 0.214, 0.1088],
    verbose_level=4, signal_cut=1.5, limitedmax=[F,T,T,T], limitedmin=[T,T,T,T],
    minpars=[0, 0, -1, 0.05], maxpars=[30.,50.,1,0.5], fixed=[F,F,F,F])
# plot best fit
spec.plotter(errstyle='fill')
spec.specfit.plot_fit()
#save figure
plt.savefig('example_o-NH2D.png')
```


OPH-H-MM1
$T_{ex}(0)$= 6.51 ± 0.16
$\tau(0)$= 4.4 ± 0.3
$v(0)$= 0.214 ± 0.002
$\sigma(0)$= 0.1088 ± 0.0024
Brightness Temperature T (K)
Velocity (km / s)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−30
−20
−10
0
10
20
30

7.13 Radio Fitting: para-NH_2D example

Example hyperfine line fitting for the para-NH_2D line.

```
import pyspeckit
import os
from pyspeckit.spectrum.models import nh2d
import numpy as np

import astropy.units as u

if not os.path.exists('p-nh2d_spec.fits'):
    import astropy.utils.data as aud
    from astropy.io import fits
    f = aud.download_file('https://github.com/pyspeckit/pyspeckit-example-files/raw/master/p-
↪nh2d_spec.fits')
    with fits.open(f) as ff:
        ff.writeto('p-nh2d_spec.fits')

# Load the spectrum
spec = pyspeckit.Spectrum('p-nh2d_spec.fits')
# Determine rms from line free section and load into cube
rms = np.std(spec.data[10:340])
spec.error[:] = rms
# setup spectral axis
spec.xarr.refX = 110.153594*u.GHz
spec.xarr.velocity_convention = 'radio'
spec.xarr.convert_to_unit('km/s')
# define useful shortcuts for True and False
F=False
T=True
# Setup of matplotlib
import matplotlib.pyplot as plt
plt.ion()
# Add NH2D fitter
spec.Registry.add_fitter('nh2d_vtau', pyspeckit.models.nh2d.nh2d_vtau_fitter,4)
# run spectral fit using some reasonable guesses
spec.specfit(fittype='nh2d_vtau', guesses=[5.52, 2.15, 0.166, 0.09067],
    verbose_level=4, signal_cut=1.5, limitedmax=[F,T,T,T], limitedmin=[T,T,T,T],
    minpars=[0, 0, -1, 0.05], maxpars=[30.,50.,1,0.5], fixed=[F,F,F,F])
# plot best fit
spec.plotter(errstyle='fill')
spec.specfit.plot_fit()
#save figure
plt.savefig('example_p-NH2D.png')
```


OPH-H-MM1
$T_{ex}(0)$= 5.52 ± 0.65
$\tau(0)$= 2.15 ± 0.69
$v(0)$= 0.1662 ± 0.0051
$\sigma(0)$= 0.0967 ± 0.0062
Brightness Temperature T (K)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−20
−10
0
10
20
Velocity (km / s)

7.14 Radio Fitting: HCN example with freely varying hyperfine amplitudes

Example hyperfine line fitting for the HCN 1-0 line.

```
from __future__ import print_function
import pyspeckit
import pylab as pl
import astropy.units as u

# Load the spectrum & properly identify the units
# The data is from http://adsabs.harvard.edu/abs/1999A%26A...348..600P
sp = pyspeckit.Spectrum('02232_plus_6138.txt')
sp.xarr.set_unit(u.km/u.s)
sp.xarr.refX = 88.63184666e9 * u.Hz
sp.xarr.velocity_convention = 'radio'
sp.xarr.xtype='velocity'
sp.unit='$T_A^*$'

# set the error array based on a signal-free part of the spectrum
sp.error[:] = sp.stats((-35,-25))['std']
# Register the fitter
# The HCN fitter is 'built-in' but is not registered by default; this example
# shows how to register a fitting procedure
# 'multi' indicates that it is possible to fit multiple components and a
# background will not automatically be fit
# 5 is the number of parameters in the model (line center,
# line width, and amplitude for the 0-1, 2-1, and 1-1 lines)
sp.Registry.add_fitter('hcn_varyhf',pyspeckit.models.hcn.hcn_varyhf_amp_fitter,5)

# This one is the same, but with fixed relative ampltidue hyperfine components
sp.Registry.add_fitter('hcn_fixedhf',pyspeckit.models.hcn.hcn_amp,3)

# Plot the results
sp.plotter()
# Run the fixed-ampltiude fitter and show the individual fit components
sp.specfit(fittype='hcn_fixedhf',
           multifit=None,
           guesses=[1,-48,0.6],
           show_hyperfine_components=True)
# Now plot the residuals offset below the original
sp.specfit.plotresiduals(axis=sp.plotter.axis,clear=False,yoffset=-1,color='g',label=False)
sp.plotter.reset_limits(ymin=-2)

# Save the figure (this step is just so that an image can be included on the web page)
sp.plotter.savefig('hcn_fixedhf_fit.png')

# Run the variable-ampltiude fitter and show the individual fit components
# Note the different order of the arguments (velocity, width, then three amplitudes)
sp.specfit(fittype='hcn_varyhf',
           multifit=None,
           guesses=[-48,1,0.2,0.6,0.3],
```

(continues on next page)

(continued from previous page)

```
           show_hyperfine_components=True,
           clear=True)

# Again plot the residuals
sp.specfit.plotresiduals(axis=sp.plotter.axis,clear=False,yoffset=-1,color='g',label=False)
sp.plotter.reset_limits(ymin=-2)

# Save the figure
sp.plotter.savefig('hcn_freehf_fit.png')



# now do the same thing, but allow the widths to vary too
# there are 7 parameters:
# 1. the centroid
# 2,3,4 - the amplitudes of the 0-1, 2-1, and 1-1 lines
# 5,6,7 - the widths of the 0-1, 2-1, and 1-1 lines
sp.Registry.add_fitter('hcn_varyhf_width',pyspeckit.models.hcn.hcn_varyhf_amp_width_fitter,7)

# Run the fitter
sp.specfit(fittype='hcn_varyhf_width',
           multifit=None,
           guesses=[-48,0.2,0.6,0.3,1,1,1],
           show_hyperfine_components=True,
           clear=True)

# print the fitted parameters:
print(sp.specfit.parinfo)
# Param #0      CENTER0 =      -51.865 +/-       0.0525058
# Param #1    AMP10-010 =      1.83238 +/-       0.0773993   Range:   [0,inf)
# Param #2    AMP12-010 =      5.26566 +/-       0.0835981   Range:   [0,inf)
# Param #3    AMP11-010 =      3.02621 +/-       0.0909095   Range:   [0,inf)
# Param #4  WIDTH10-010 =      2.16711 +/-        0.118651   Range:   [0,inf)
# Param #5  WIDTH12-010 =      1.90987 +/-       0.0476163   Range:   [0,inf)
# Param #6  WIDTH11-010 =      1.64409 +/-        0.076998   Range:   [0,inf)

sp.specfit.plotresiduals(axis=sp.plotter.axis,clear=False,yoffset=-1,color='g',label=False)
sp.plotter.reset_limits(ymin=-2)

# Save the figure (this step is just so that an image can be included on the web page)
sp.plotter.savefig('hcn_freehf_ampandwidth_fit.png')

# Finally, how well does a 2-component fit work?
sp.specfit(fittype='hcn_fixedhf',
           multifit=None,
           guesses=[1,-48,0.6,0.1,-46,0.6],
           show_hyperfine_components=True,
           clear=True)
sp.specfit.plotresiduals(axis=sp.plotter.axis,clear=False,yoffset=-1,color='g',label=False)
sp.plotter.reset_limits(ymin=-2)

# Save the figure (this step is just so that an image can be included on the web page)
```

(continues on next page)

(continued from previous page)

```
sp.plotter.savefig('hcn_fixedhf_fit_2components.png')
```

The green lines in the following figures all show the residuals to the fit

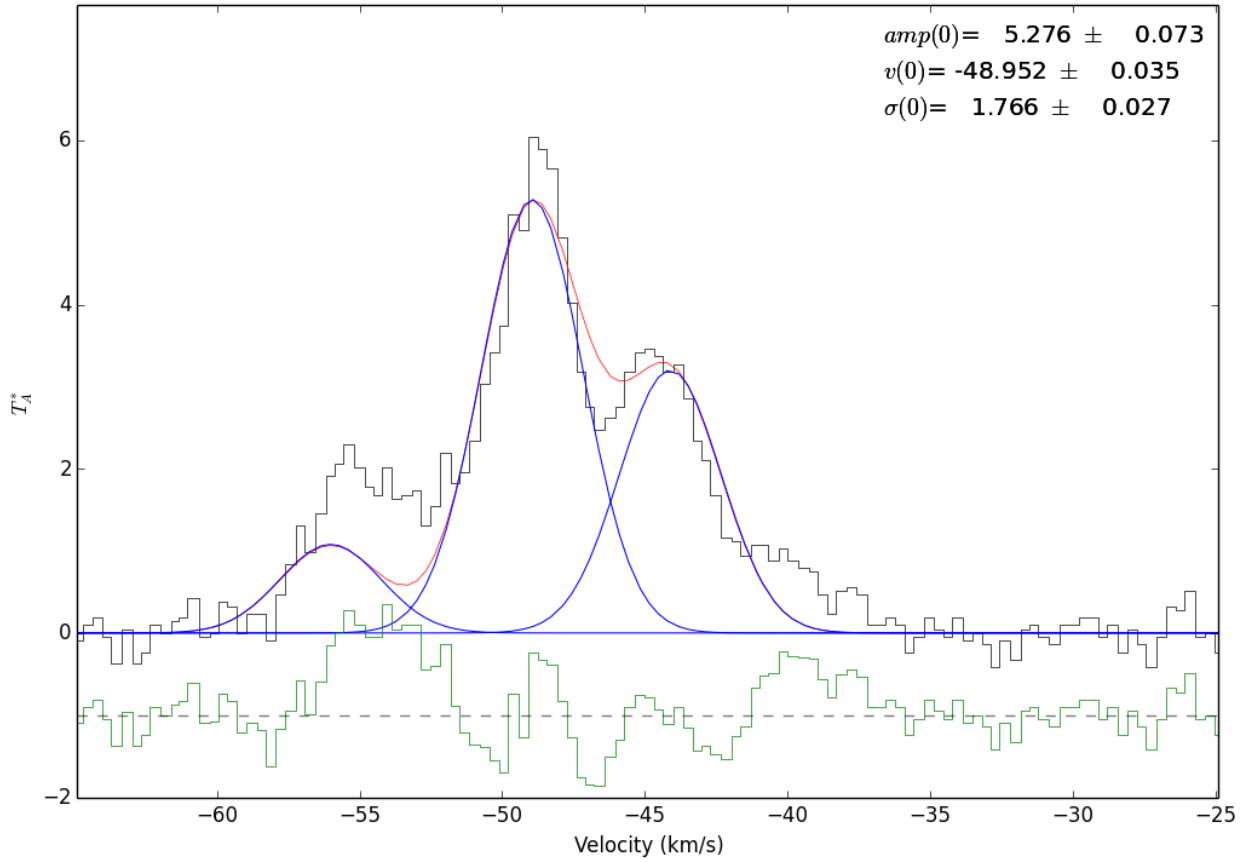


Fig. 9: Fit to the 3 hyperfine components of HCN 1-0 simultaneously with fixed amplitudes. The (0)'s indicate that this is the 0'th velocity component being fit (though that velocity corresponds to the 12-01 component of the line)

7.15 Simple Radio Fitting: HCO+ example

```
import pyspeckit

# load a FITS-compliant spectrum
spec = pyspeckit.Spectrum('10074-190_HCOp.fits')
# The units are originally frequency (check this by printing spec.xarr.units).
# I want to know the velocity.  Convert!
# Note that this only works because the reference frequency is set in the header
spec.xarr.convert_to_unit('km/s')
```

(continues on next page)

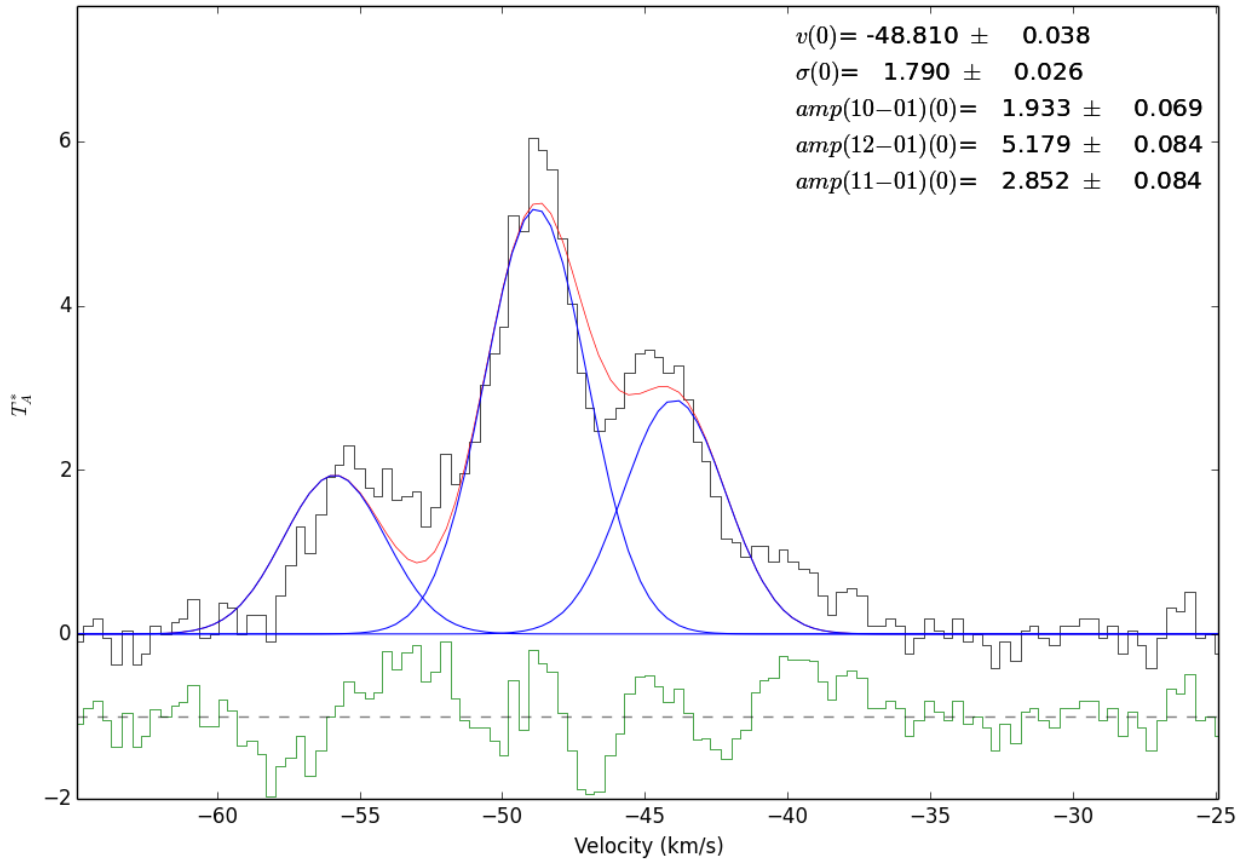


Fig. 10: Fit to the 3 hyperfine components of HCN 1-0 simultaneously with freely varying amplitudes. The (0)'s indicate that this is the 0'th velocity component being fit

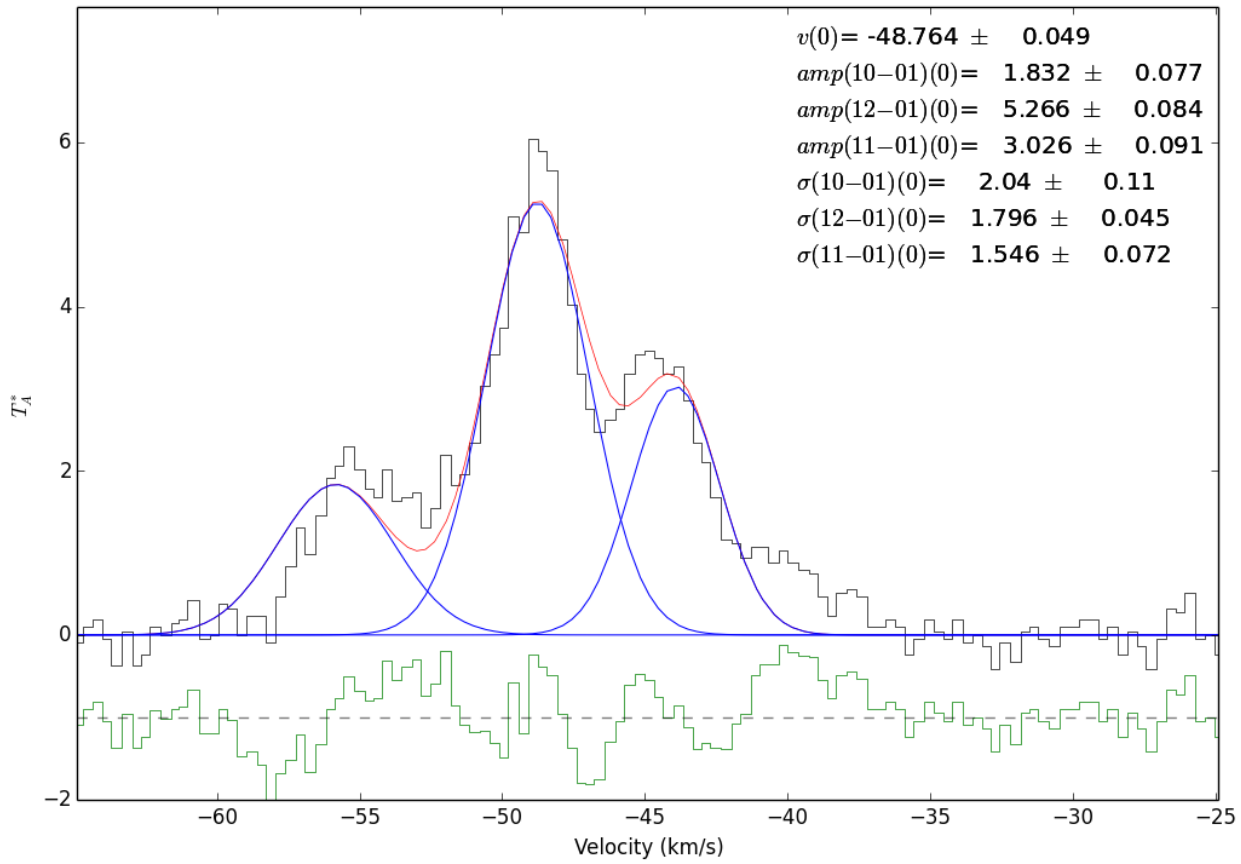


Fig. 11: Fit to the 3 hyperfine components of HCN 1-0 simultaneously. The widths are allowed to vary in this example.

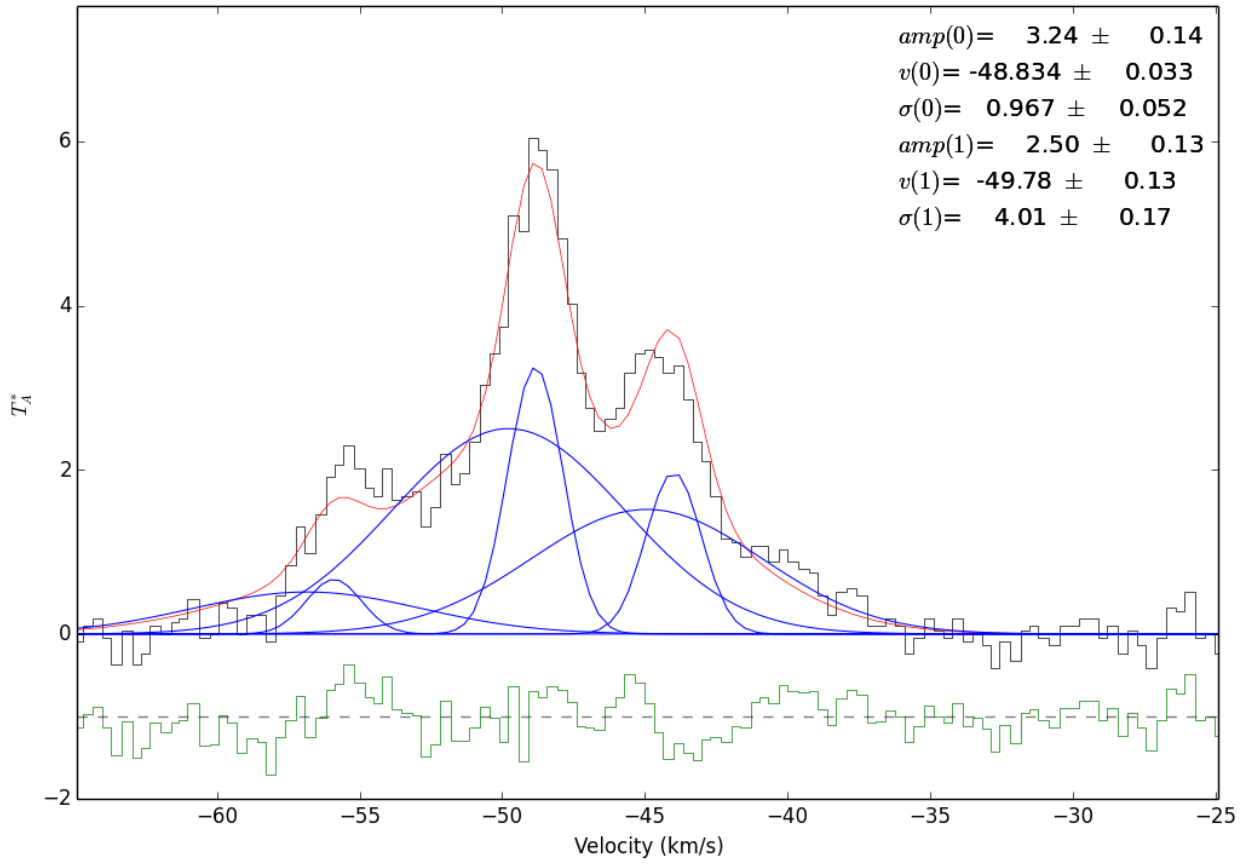


Fig. 12: A two-component fit to the same spectrum. It appears to be a much better fit, hinting that there are indeed two components (though the fit is probably not unique)

(continued from previous page)

```
# plot it
spec.plotter()

# Subtract a baseline
spec.baseline()

# Fit a gaussian.  We know it will be an emission line, so we force a positive guess
spec.specfit(negamp=False)

# Note that the errors on the fits are larger than the fitted parameters.
# That's because this spectrum did not have an error assigned to it.
# Let's use the residuals:
spec.specfit.plotresiduals()

# Now, refit with error determined from the residuals:
# (we pass in guesses to save time / make sure nothing changes)
spec.specfit(guesses=spec.specfit.modelpars)

# Save the figures to put on the web....
spec.plotter.figure.savefig("simple_fit_example_HCOp.png")
spec.specfit.residualaxis.figure.savefig("simple_fit_example_HCOp_residuals.png")

# Also, let's crop out stuff we don't want...
spec.crop(-100,100)
# replot after cropping (crop doesn't auto-refresh)
spec.plotter()
# replot the fit without re-fitting
spec.specfit.plot_fit()
# show the annotations again
spec.specfit.annotate()

spec.plotter.figure.savefig("simple_fit_example_HCOp_cropped.png")
```

7.16 LTE Line Model example: Line identification

```
import numpy as np

# retrieve the data
import requests

# 10 MB file: stream it, but it should be able to load in memory
response = requests.get('https://dataverse.harvard.edu/api/access/datafile/:persistentId?
↪persistentId=doi:10.7910/DVN/8QJT3K/POCX2W', stream=True)

with open('temp_cube.fits', 'wb') as fh:
    fh.write(response.content)
```

(continues on next page)

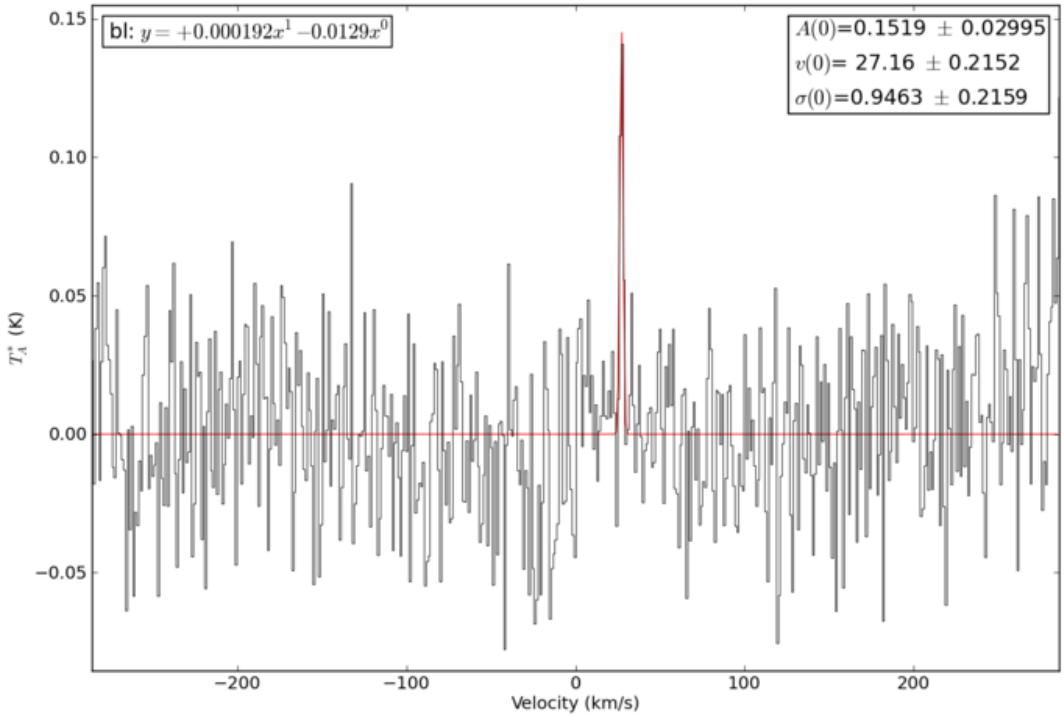


Fig. 13: Sample HCO+ spectrum fitted with a gaussian

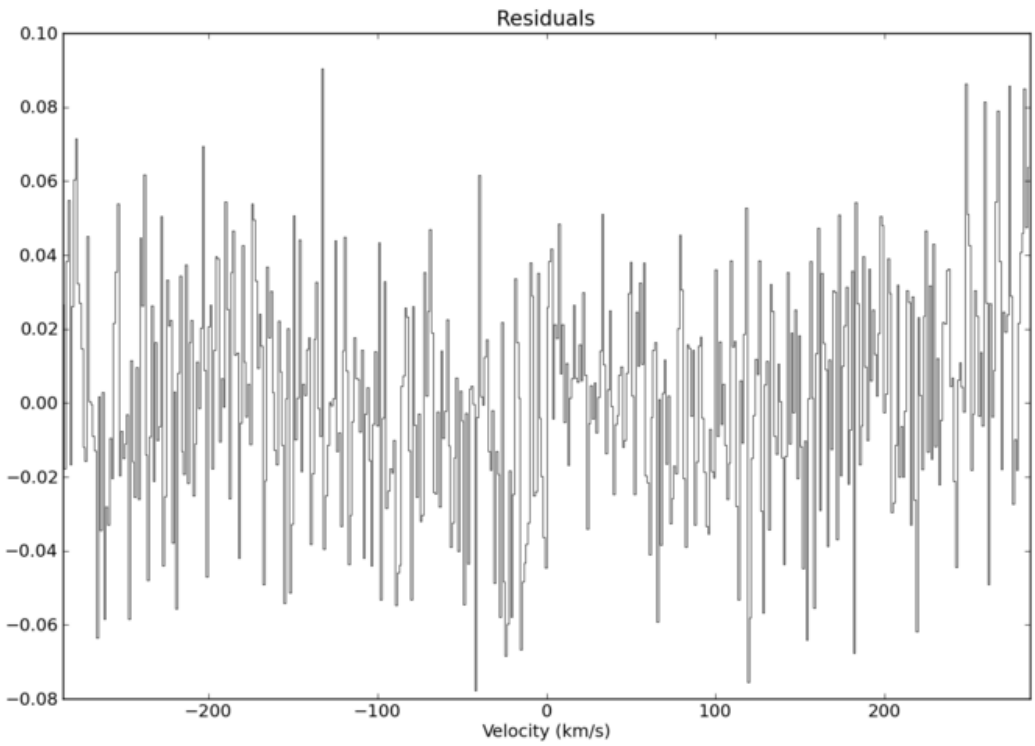


Fig. 14: Residuals of the gaussian fit from the previous figure

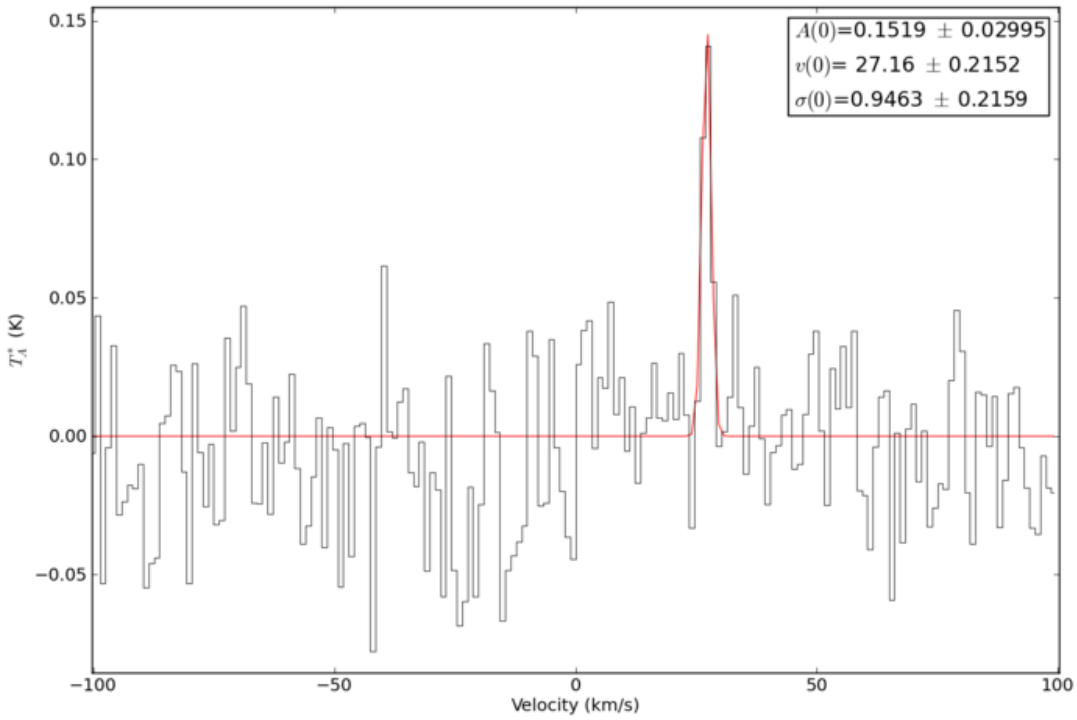


Fig. 15: A zoomed-in, cropped version of the spectrum. With the 'crop' command, the excess data is discarded.

(continued from previous page)

```
from spectral_cube import SpectralCube
from astropy import units as u
# there is some junk at low frequency
cube = SpectralCube.read('temp_cube.fits').spectral_slab(218.14*u.GHz, 218.54*u.GHz)

# this isn't really right b/c of beam issues, but it's Good Enough
meanspec = cube.mean(axis=(1,2))
# have to hack this, which is _awful_ and we need to fix it
meanspec_K = meanspec.value * cube.beam.jtok(cube.spectral_axis)

import pyspeckit

sp = pyspeckit.Spectrum(xarr=cube.spectral_axis, data=meanspec_K)



# The LTE model doesn't include an explicit filling factor, so we impose one
fillingfactor = 0.05

# there's continuum in our spectrum, but it's also not modeled
offset = np.nanpercentile(sp.data.filled(np.nan), 20)

# look at a couple species
# Be cautious! Not all line catalogs have all of these species, some species
# can result in multiple matches from splatalogue, and definitely not all
# species have the same column and excitation
species_list = ('CH3OH', 'HCCCN', 'H2CO', 'CH3OCHO', 'CH3OCH3', 'C2H5CN',)
# 'C2H3CN' is pretty definitely not here at the constant assumed column

# make a set of subplots:
import pylab as pl
fig, axes = pl.subplots(len(species_list)+1, 1, figsize=(18,14), num=1)
pl.draw()

from pyspeckit.spectrum.models import lte_molecule
mods = []
for species, axis in zip(species_list, axes):
    # search for lines within +/- 0.5 GHz of the min/max (to account for velocity offset,␣
↪generously)
    freqs, aij, deg, EU, partfunc = lte_molecule.get_molecular_parameters(species,
                                                                          fmin=sp.xarr.min()-
↪0.5*u.GHz,
                                                                          fmax=sp.xarr.
↪max()+0.5*u.GHz)

    mod = lte_molecule.generate_model(xarr=sp.xarr, vcen=50*u.km/u.s,
                                      width=3*u.km/u.s, tex=100*u.K,
                                      column=1e17*u.cm**-2, freqs=freqs,
                                      aij=aij, deg=deg, EU=EU,
                                      partfunc=partfunc)
    mods.append(mod)
```

(continues on next page)

(continued from previous page)

```
    sp.plotter(axis=axis, figure=fig, clear=False)
    sp.plotter.axis.plot(sp.xarr, mod*fillingfactor+offset, label=species, zorder=-1)
    sp.plotter.axis.text(0.5,0.9,species,transform=axis.transAxes)

    if axis != axes[-1]:
        axis.set_xlabel("")
        axis.set_xticklabels([])
    if axis != axes[len(species_list) // 2]:
        axis.set_ylabel("")


axis = axes[-1]
sp.plotter(axis=axis, figure=fig, clear=False)
sp.plotter.axis.plot(sp.xarr, np.nansum(mods,axis=0)*fillingfactor+offset, label='Sum',␣
↪zorder=-1)
sp.plotter.axis.text(0.5,0.9,'Sum',transform=axis.transAxes)
```

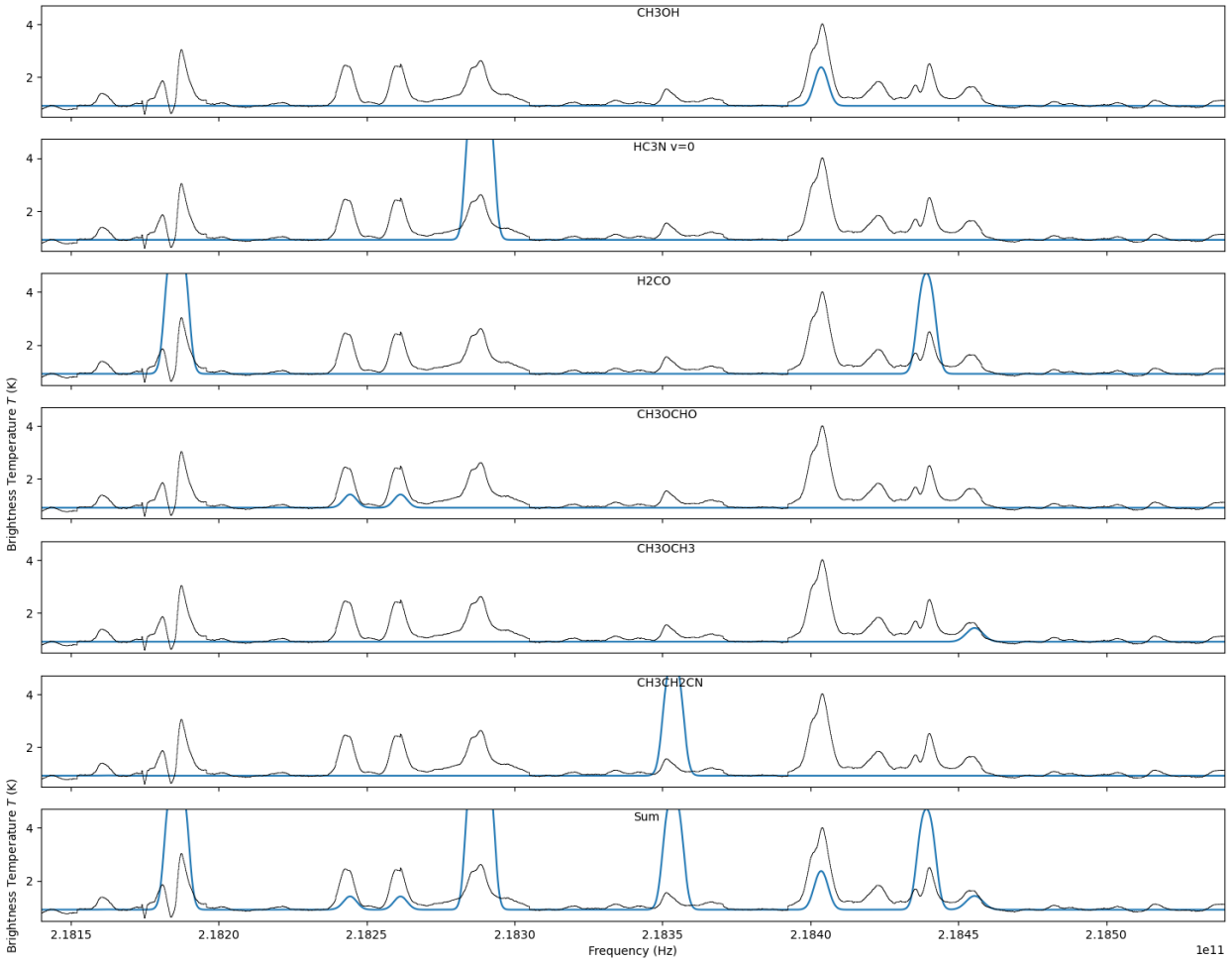


Fig. 16: W51 spectrum with LTE lines at T_X = 100, N=10^17

7.17 Fitting H2 1-0 S(1) line in NIR data cube

```
import astropy
import pyspeckit
import pylab as pl
import numpy as np
import astropy.io.fits as pyfits


# load the cube as a pyspeckit Cube
# I usually get an error here if cube.fits' header doesn't have 'CTYPE3' = 'WAVE'
cube = pyspeckit.Cube('cube.fits')

# Slice the cube over the wavelength range you'd like to fit
cube_h2 = cube.slice(21145,21250,unit='Angstrom')

# rescale data to make fitting nicer
# I add an offset to avoid negative values
cube_h2.cube *= 1e17
cube_h2.cube += 30.

# Do an initial plot & fit of a single spectrum
# at a pixel with good S/N
cube_h2.plot_spectrum(100,100)

### The following command is just for setup!  The actual fitting occurs below. ###
# Here I'm fitting two gaussians with 4 parameters each (background offset,
# amplitude, wavelength centroid, linewidth).
# I find that if I let both backgrounds be free parameters, pyspeckit returns
# unrealistic values for both backgrounds, so I fix the 2nd gaussian's background
# level to 0.  The actual command to fix the parameter comes in the fiteach call.
cube_h2.specfit(fittype='vheightgaussian',guesses=[3,1,2.1220e4,2,0,50,21232.,2],quiet=False,
↪save=False)

# Get ready for the interactive plots that come up after fiteach finishes
cube_h2.mapplot.makeplane(estimator=np.nansum)

# For my cube.fits, (21145,21195) covers a part of the spectrum that is free of
# spectral lines.  The std variable will be used to estimate the S/N of a line
# during fiteach.
std = cube_h2.stats((21145,21195))['std']

#### Here's where all the fitting happens.
## With the "parlimited" and "parlimits" keywords, I have restricted
## the range for the wavelength centroid and linewidth parameters.
## With the "fixed" keyword, I have held the 2nd gaussian's background level
## to zero, and the "signal_cut" keyword rejects fits for voxels below a
## user-specified S/N threshold.
cube_h2.fiteach(use_nearest_as_guess=False,
```

(continues on next page)

(continued from previous page)

```
                guesses=[3,1,2.1220e4,2,0,50,21232.,2],
                fittype='vheightgaussian',
                integral=False,
                multicore=4,
                negamp=False,
                verbose_level=2,
                errspec=np.ones(cube_h2.shape)*std,
                parlimited=[(False,False), (False,False), (True,True),
                            (True,True), (False,False), (True,True),
                            (True,True), (True,True)],
                parlimits=[(0.9,1.4), (0,16), (21210,21225), (0.5,5),
                           (0.9,1.4), (0,100), (21227.,21236), (0.5,5)],
                fixed=[False, False, False, False, True, False, False,
                       False],
                signal_cut=20,
                start_from_point=(100,100))

# plot the fits as images (you can click on background image to see the spectra + fits)
amp_max = np.max(cube_h2.parcube[1,:,:])
cube_h2.mapplot(estimator=1,vmax=amp_max,vmin=0)
cube_h2.mapplot.axis.set_title("Amplitude")

cube_h2.mapplot.figure=pl.figure(5)
cube_h2.mapplot(estimator=3, vmax=5, vmin=0)
cube_h2.mapplot.axis.set_title("Line Width")

cube_h2.mapplot.figure=pl.figure(6)
cube_h2.mapplot(estimator=2,vmin=21215,vmax=21225)
cube_h2.mapplot.axis.set_title("Line Center")

cube_h2.mapplot.figure=pl.figure(7)
cube_h2.mapplot(estimator=0,vmax=100,vmin=0)
cube_h2.mapplot.axis.set_title("Background")
pl.show()

## Create the image
background = (cube_h2.parcube[0,:,:] - 30.) / 1e17
amplitude = cube_h2.parcube[1,:,:] / 1e17
linecenter = cube_h2.parcube[2,:,:]
sigma = cube_h2.parcube[3,:,:] / 3e5 * h2_linecenter
image = np.sqrt(2*np.pi)*h2_amplitude*h2_sigma

# Clean up the header
# (this is a bit of a hacky way to do it, but it works)
cube_h2.header['NAXIS'] = 2
del cube_h2.header['NAXIS3']
# a nicer way is to use WCS:
from astropy import wcs
newheader = wcs.WCS(cube_h2.header).sub([wcs.WCSSUB_CELESTIAL]).to_header()
cube2.header = newheader
# however, this approach may lose other important header keywords
```

(continues on next page)

(continued from previous page)

```
# Write the image to file
h2filename = input_filename.replace("cube.fits","h2_1-0S1.fits")
h2fits = pyfits.PrimaryHDU(data=h2_image,header=cube_h2.header)
if astropy.version.major >= 2 or (astropy.version.major==1 and astropy.version.minor>=3):
     h2fits.writeto(h2filename,overwrite=True)
else:
     h2fits.writeto(h2filename,clobber=True)

# Write pyspeckit parcube and errcube to file
pyspeckit_fits_filename = input_filename.replace("cube.fits",
                                                 "pyspeckitfits_h2_1-0S1.fits")
cube_h2.write_fit(pyspeckit_fits_filename,overwrite=True)
```

7.18 Optical fitting: The H-[NII] complex of a type-I Seyfert galaxy

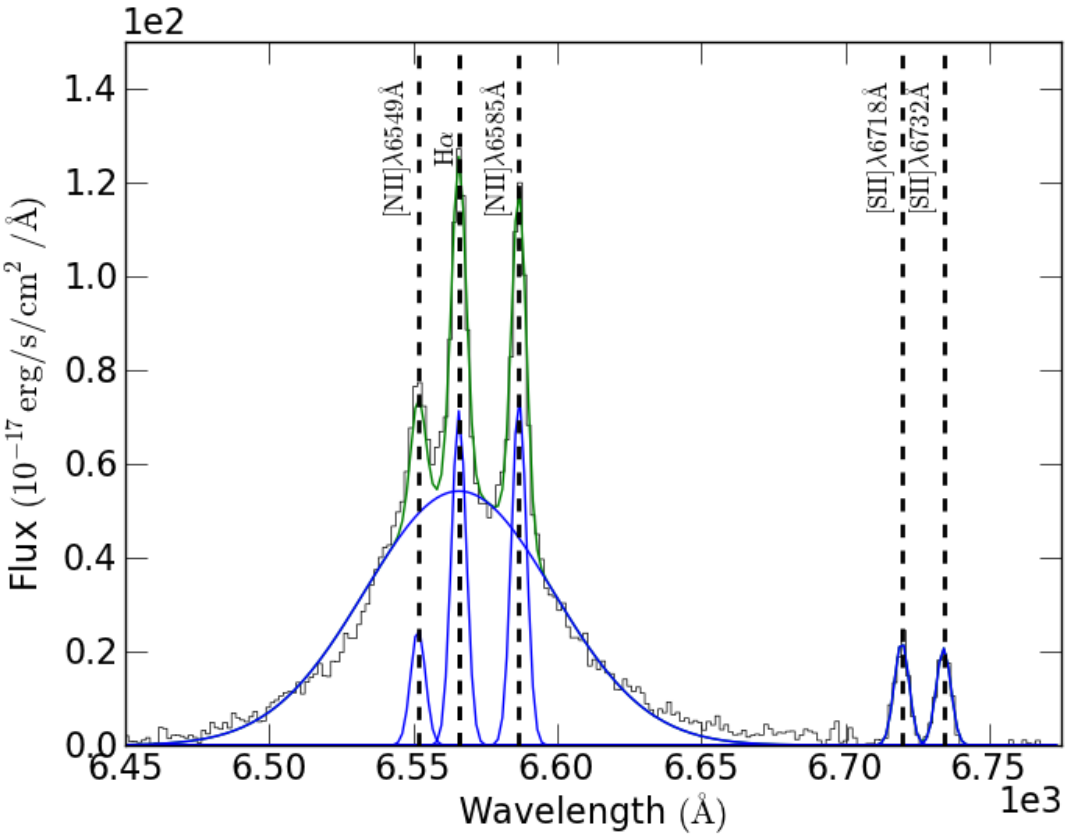
Source file for this example is here: https://github.com/pyspeckit/pyspeckit/blob/master/examples/sn_example.py

```
import pyspeckit

# Rest wavelengths of the lines we are fitting - use as initial guesses
NIIa = 6549.86
NIIb = 6585.27
Halpha = 6564.614
SIIa = 6718.29
SIIb = 6732.68

# Initialize spectrum object and plot region surrounding Halpha-[NII] complex
spec = pyspeckit.Spectrum('sample_sdss.txt', errorcol=2)
spec.plotter(xmin = 6450, xmax = 6775, ymin = 0, ymax = 150)

# We fit the [NII] and [SII] doublets, and allow two components for Halpha.
# The widths of all narrow lines are tied to the widths of [SII].
guesses = [50, NIIa, 5, 100, Halpha, 5, 50, Halpha, 50, 50, NIIb, 5, 20, SIIa,
           5, 20, SIIb, 5]
tied = ['', '', 'p[17]', '', '', 'p[17]', '', 'p[4]', '', '3 * p[0]', '',
        'p[17]', '', '', 'p[17]', '', '', '']

# Actually do the fit.
spec.specfit(guesses = guesses, tied = tied, annotate = False)
spec.plotter.refresh()

# Let's use the measurements class to derive information about the emission
# lines.  The galaxy's redshift and the flux normalization of the spectrum
# must be supplied to convert measured fluxes to line luminosities.  If the
# spectrum we loaded in FITS format, 'BUNITS' would be read and we would not
# need to supply 'fluxnorm'.
```

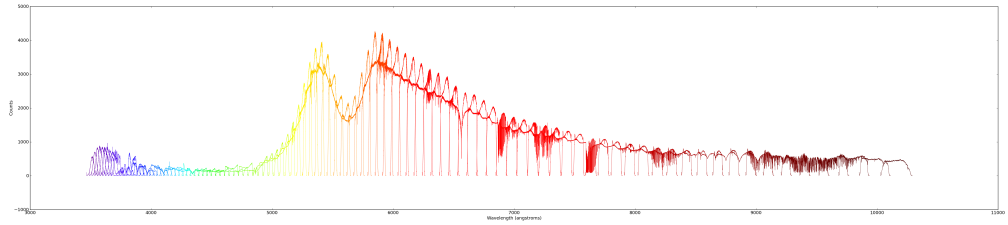
(continues on next page)

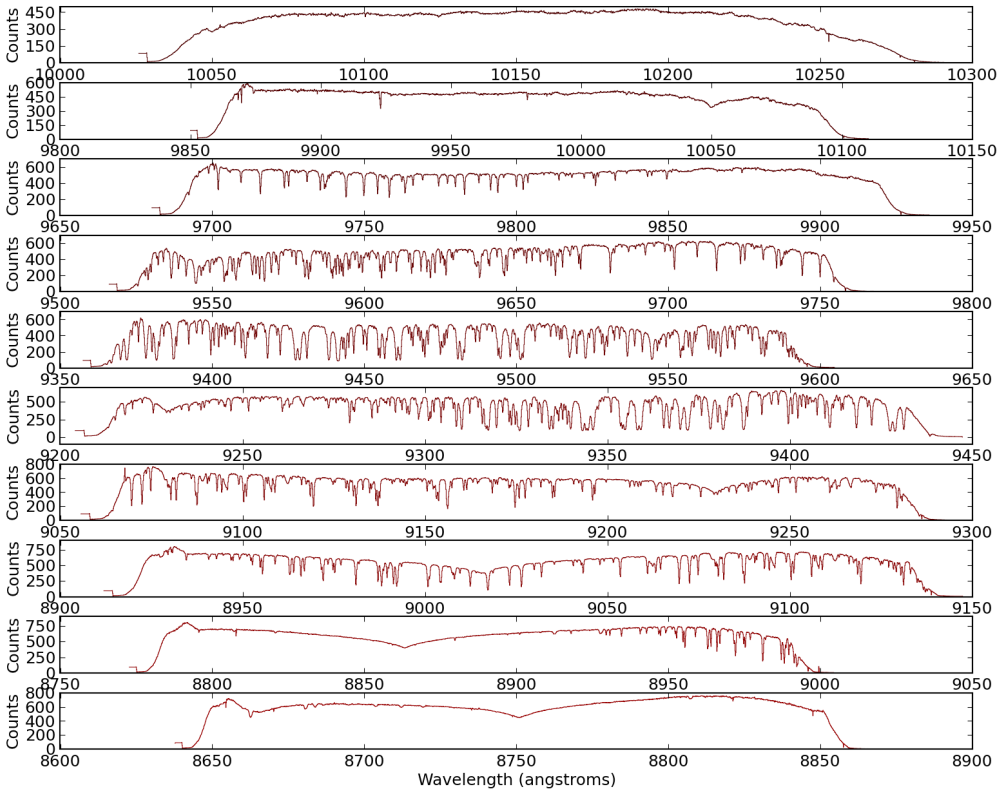
(continued from previous page)

```
spec.measure(z = 0.05, fluxnorm = 1e-17)

# Now overplot positions of lines and annotate

y = spec.plotter.ymax * 0.85    # Location of annotations in y

for i, line in enumerate(spec.measurements.lines.keys()):

    # If this line is not in our database of lines, don't try to annotate it
    if line not in spec.speclines.optical.lines.keys(): continue

    x = spec.measurements.lines[line]['modelpars'][1]   # Location of the emission line
    # Draw dashed line to mark its position
    spec.plotter.axis.plot([x]*2, [spec.plotter.ymin, spec.plotter.ymax],
                           ls='--', color='k')
    # Label it
    spec.plotter.axis.annotate(spec.speclines.optical.lines[line][-1], (x, y),
                               rotation = 90, ha = 'right', va = 'center')
# Make some nice axis labels
spec.plotter.axis.set_xlabel(r'Wavelength $(\AA)$')
spec.plotter.axis.set_ylabel(r'Flux $(10^{-17} \mathrm{erg/s/cm^2/\AA})$')
spec.plotter.refresh()

# Print out spectral line information
print("Line   Flux (erg/s/cm^2)     Amplitude (erg/s/cm^2)"
      "    FWHM (Angstrom)   Luminosity (erg/s)")
for line in spec.measurements.lines.keys():
    print(line, spec.measurements.lines[line]['flux'],
          spec.measurements.lines[line]['amp'],
          spec.measurements.lines[line]['fwhm'],
          spec.measurements.lines[line]['lum'])

# Had we not supplied the objects redshift (or distance), the line
# luminosities would not have been measured, but integrated fluxes would
# still be derived.  Also, the measurements class separates the broad and
# narrow H-alpha components, and identifies which lines are which. How nice!

spec.specfit.plot_fit()

# Save the figure
spec.plotter.figure.savefig("sdss_fit_example.png")
```


1e2
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Flux (10^{-17} erg/s/cm^2 /Å)
[NII]λ6549Å
Hα
[NII]λ6585Å
[SII]λ6718Å
[SII]λ6732Å
6.45
6.50
6.55
6.60
6.65
6.70
6.75
1e3
Wavelength (Å)

7.19 Optical Plotting - Echelle spectrum of Vega (in color!)





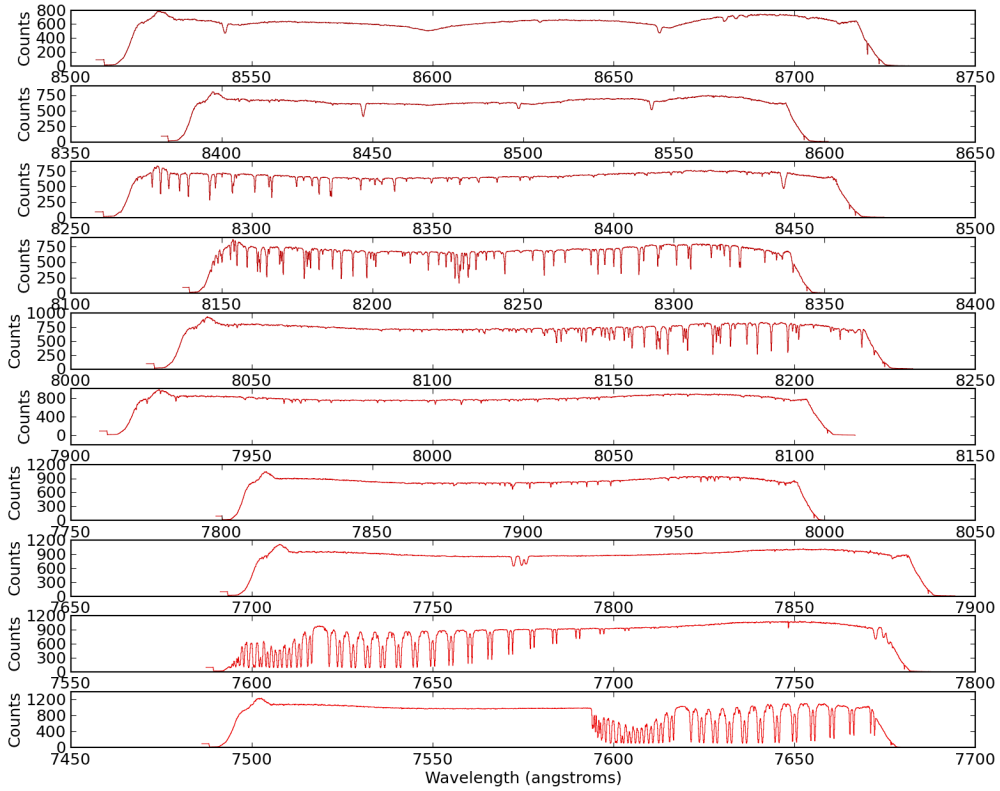
7.20 A guide to interactive fitting

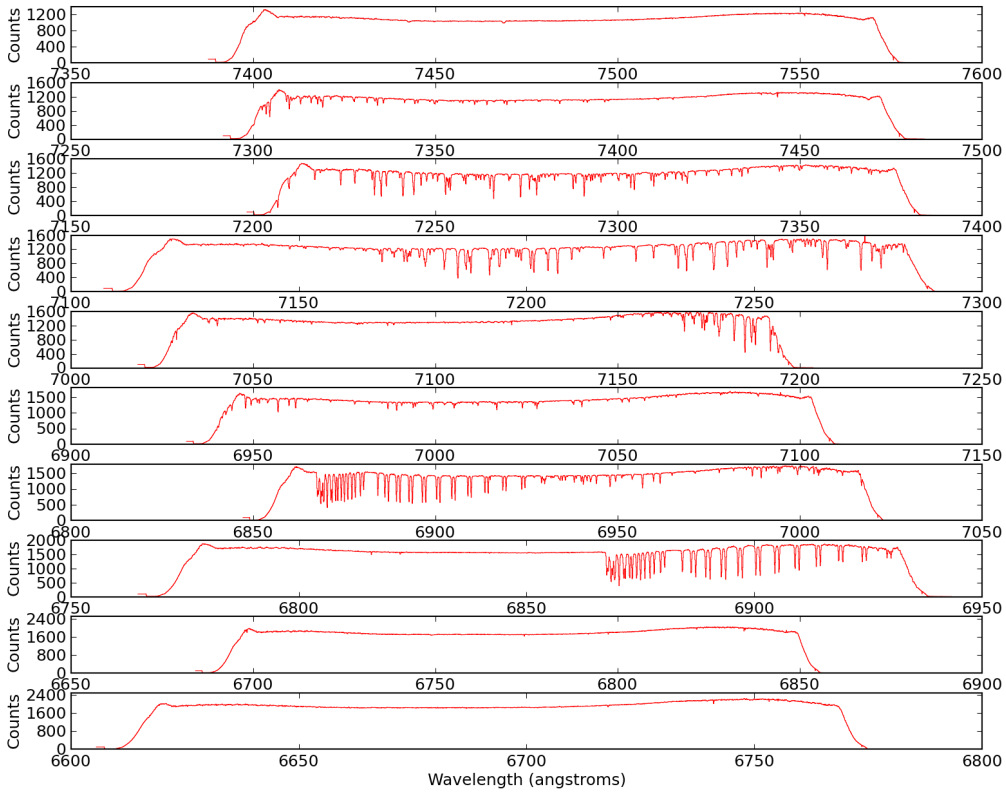
A step-by-step example of how to use the interactive fitter.

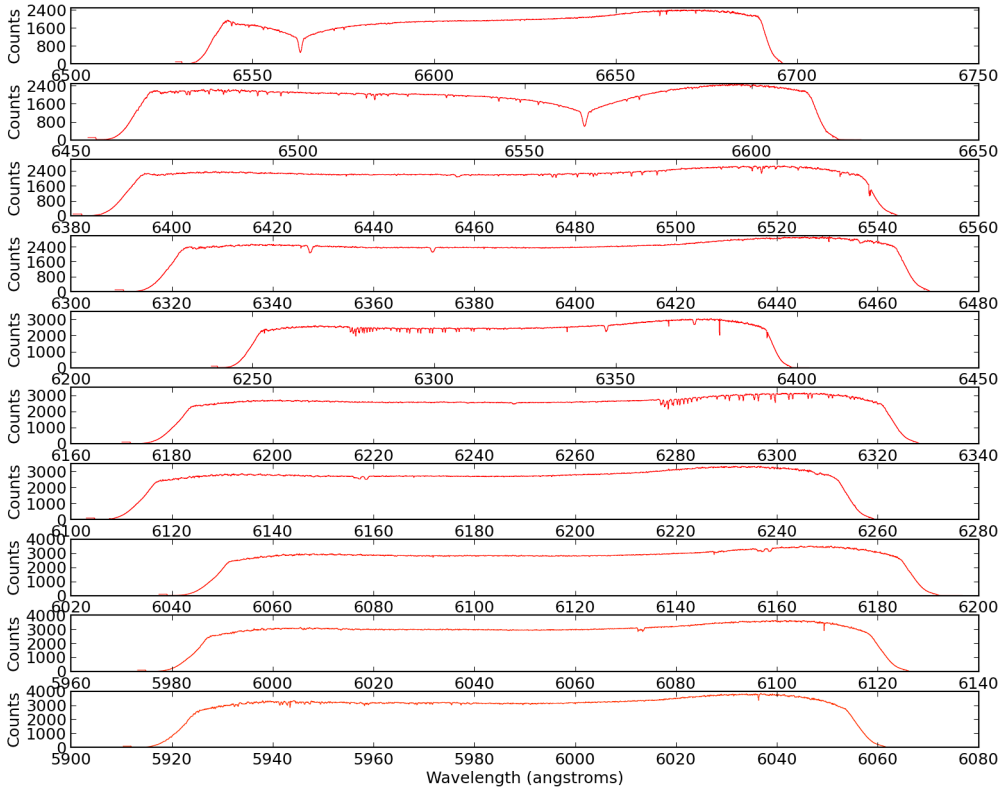
In short, we will do the following:

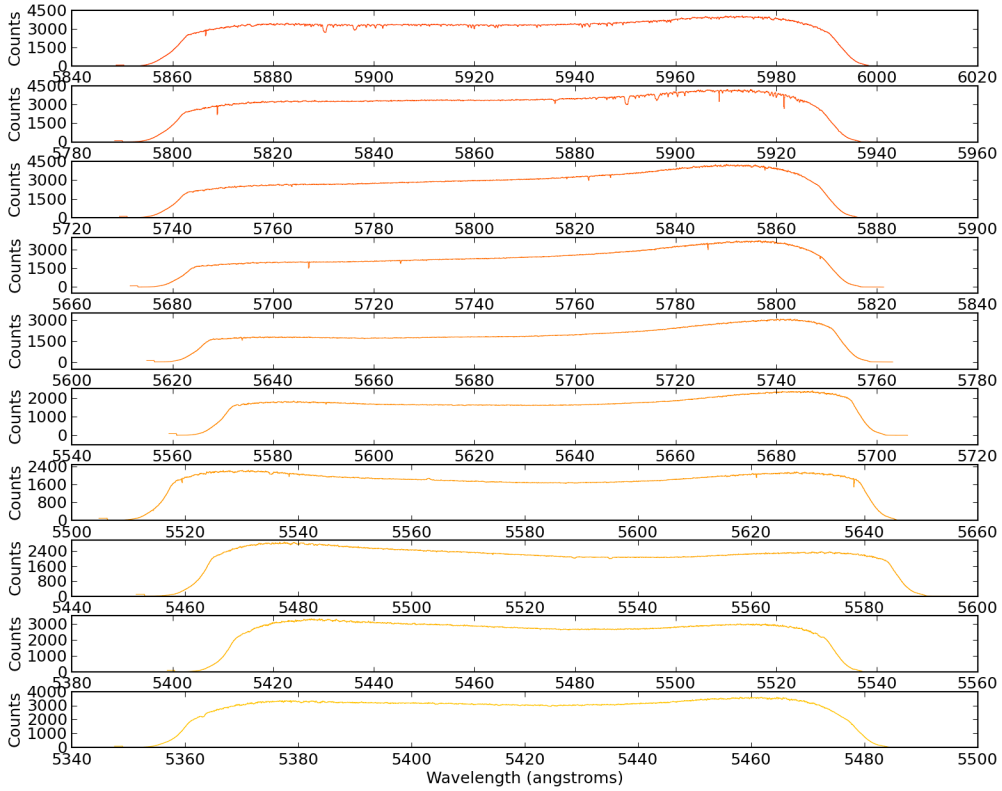
```
# 1. Load the spectrum
sp = pyspeckit.Spectrum('hr2421.fit')

# 2. Plot a particular spectral line
```

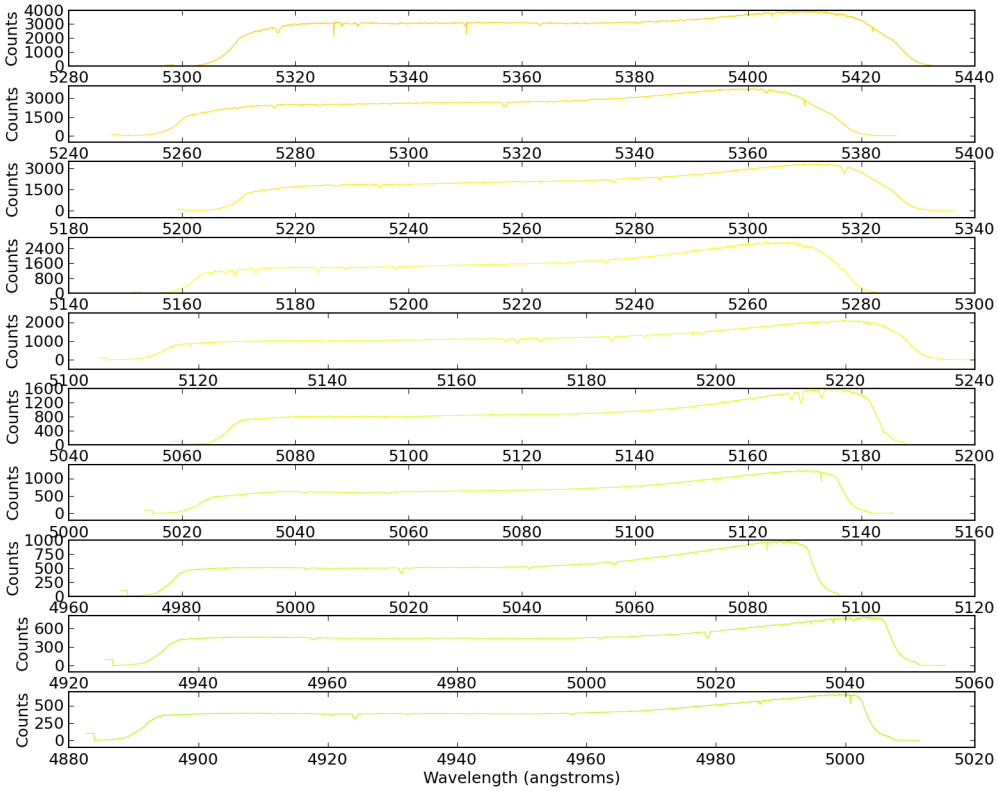
(continues on next page)

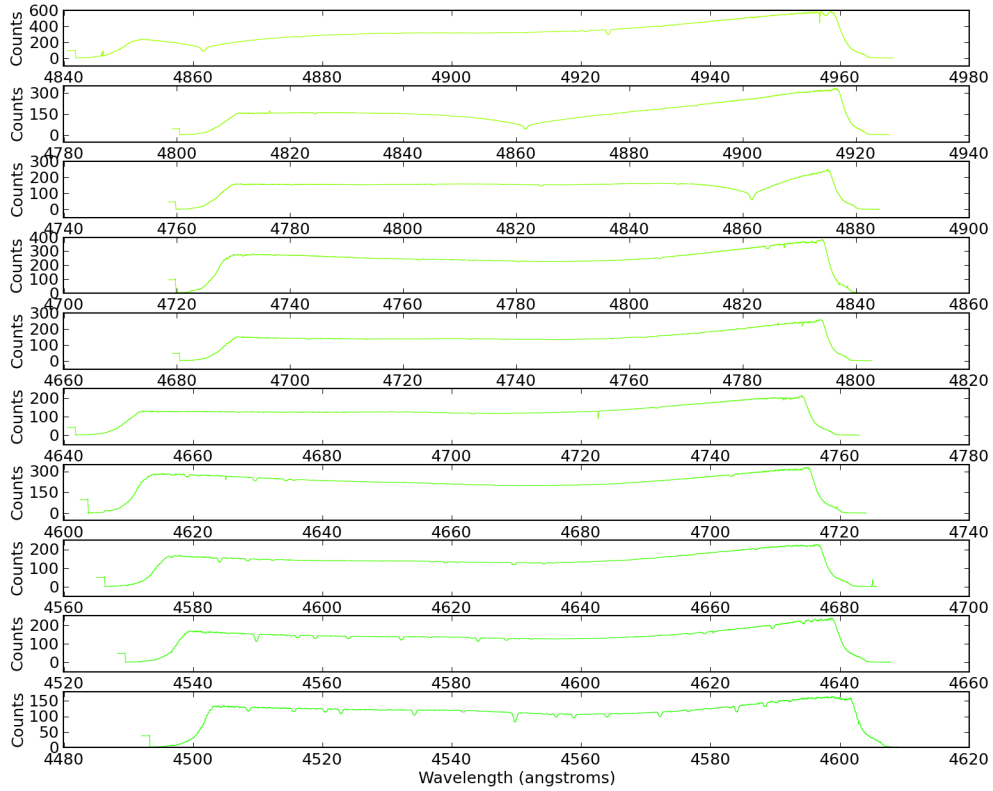

Counts
Wavelength (angstroms)

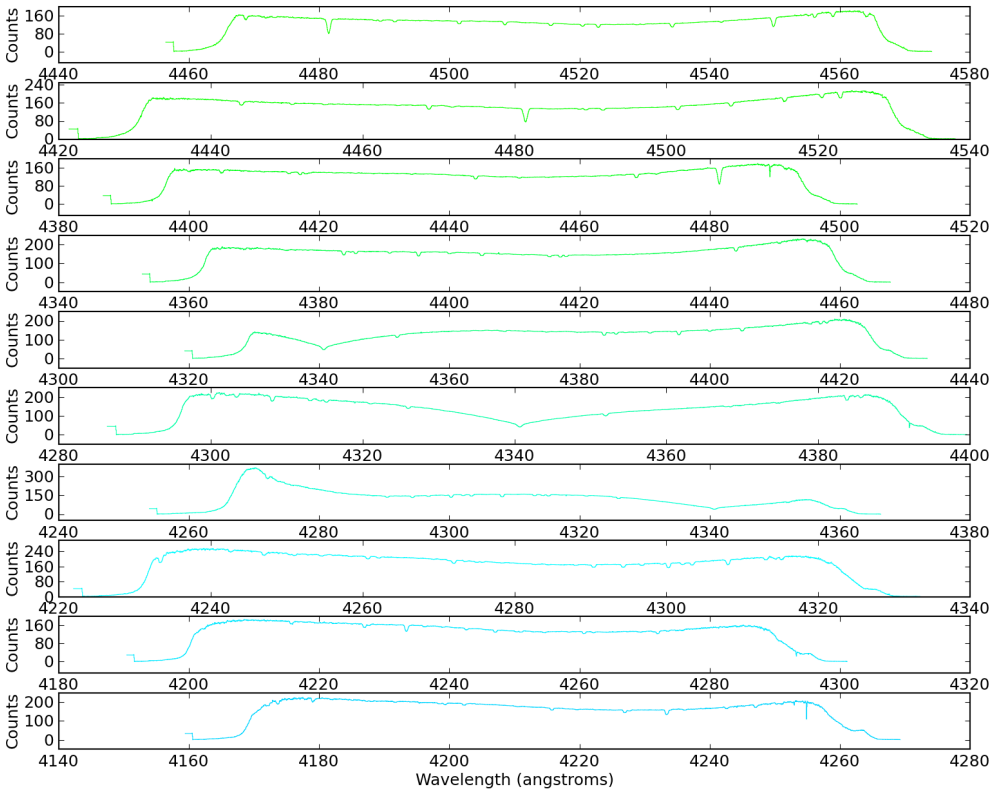

Counts
Wavelength (angstroms)

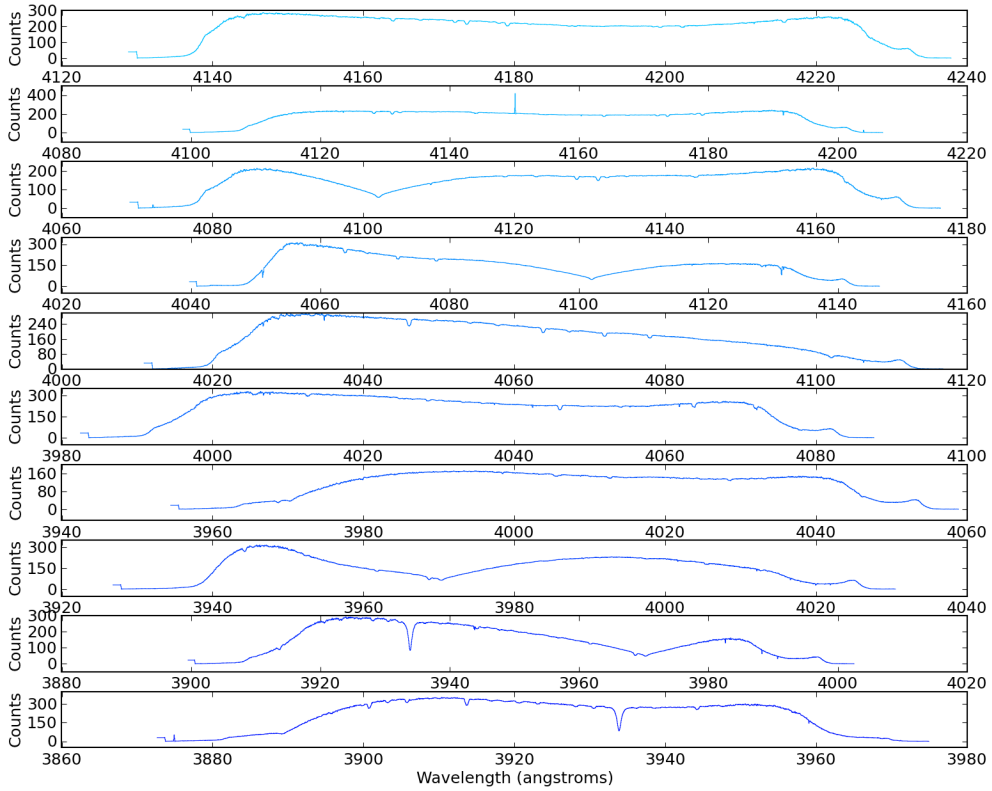

Counts
Wavelength (angstroms)

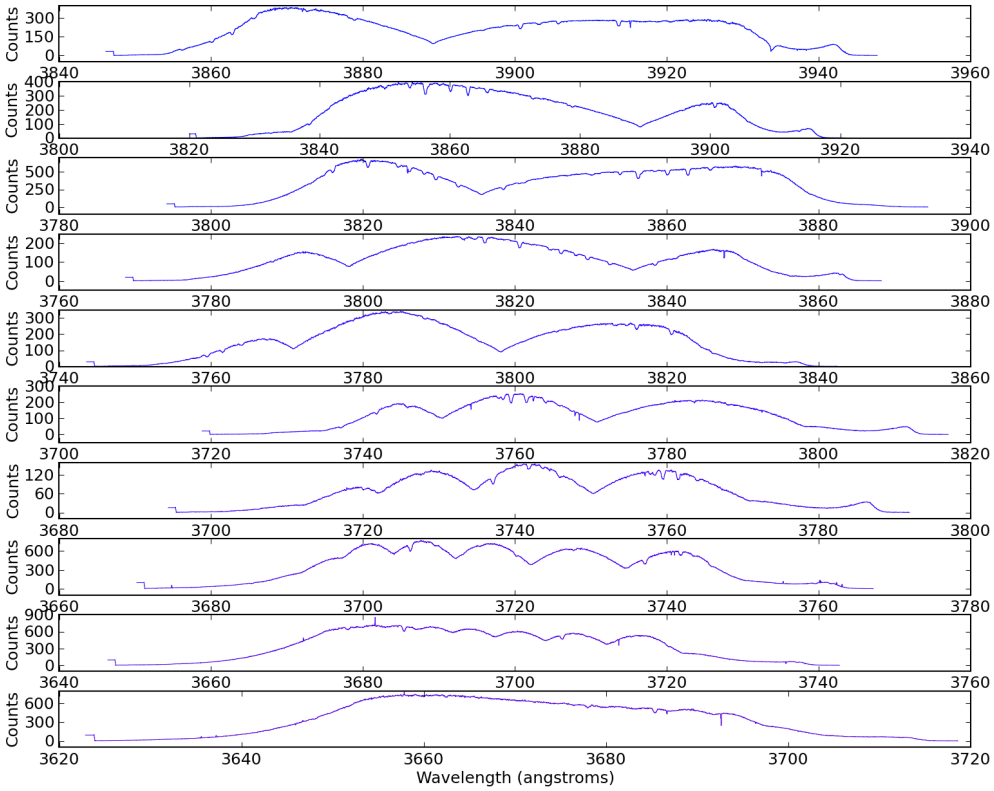

Counts
Wavelength (angstroms)


Counts
Wavelength (angstroms)


Counts
Wavelength (angstroms)


Counts
Wavelength (angstroms)


Counts
Wavelength (angstroms)



(continued from previous page)

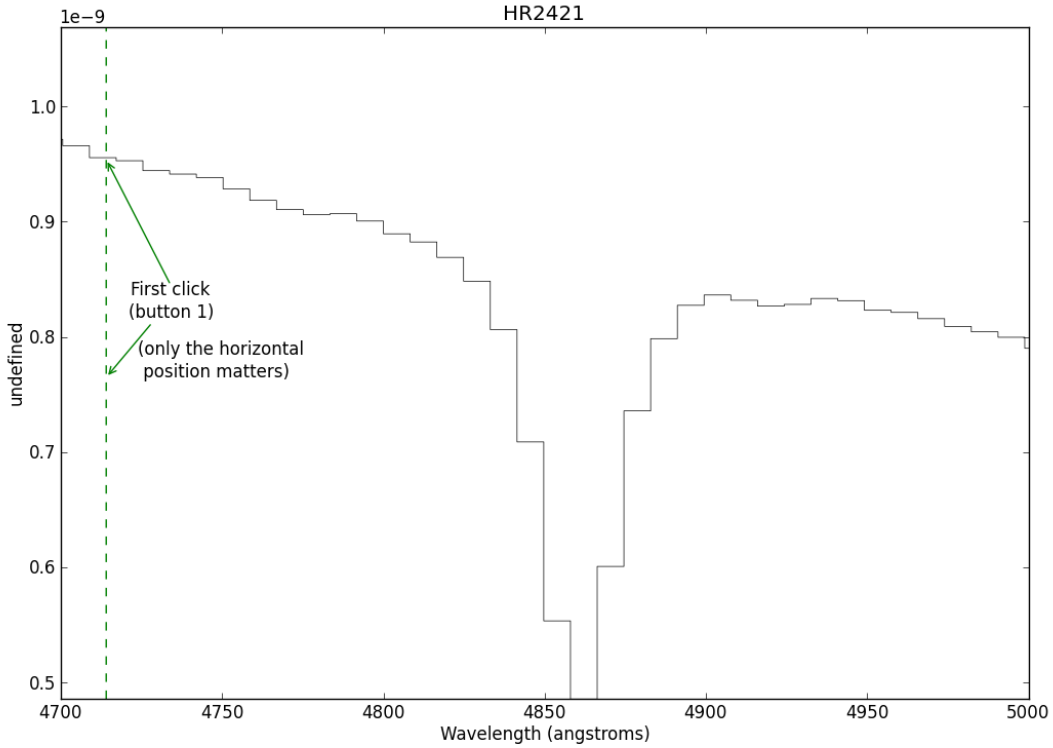
```
sp.plotter(xmin=4700,xmax=5000)

# 3. Need to fit the continuum first
sp.baseline(interactive=True, subtract=False)

# DO INTERACTIVE THINGS HERE
# 4... (much work takes place interactively at this stage)

# 5. Start up an interactive line-fitting session
# (do not run this line until *after* completing the baseline fitting)
#sp.specfit(interactive=True)
```

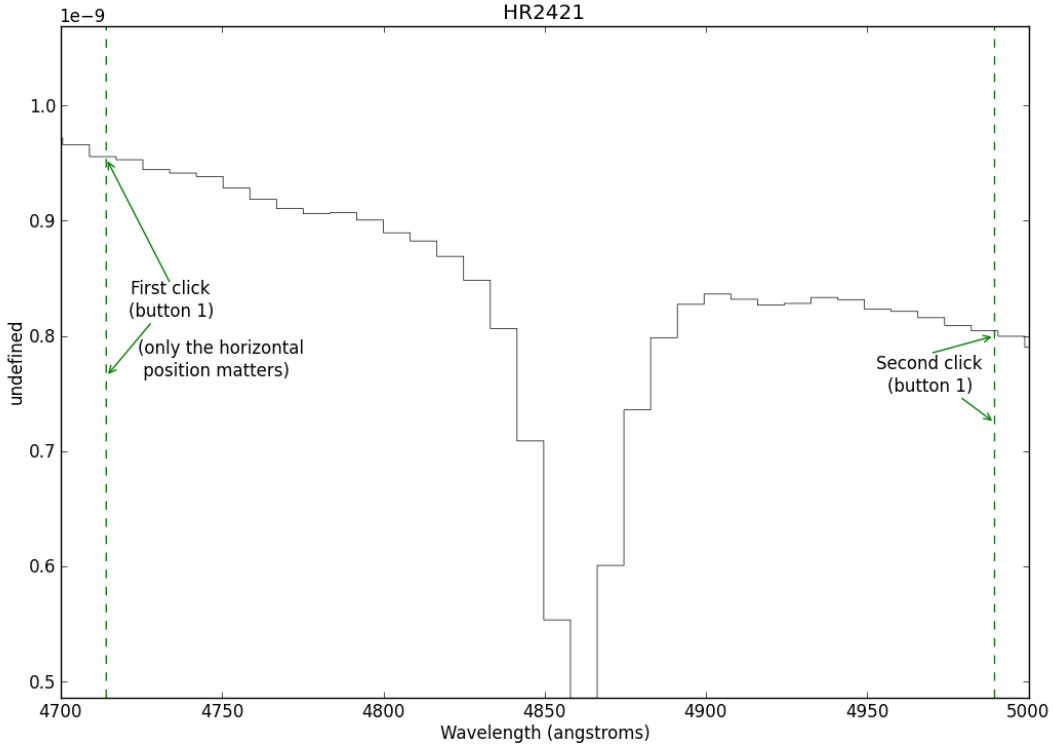
Note: If you don't see a plot window after step #2 above, make sure you're using matplotlib in interactive mode. This may require starting ipython as `ipython --pylab`

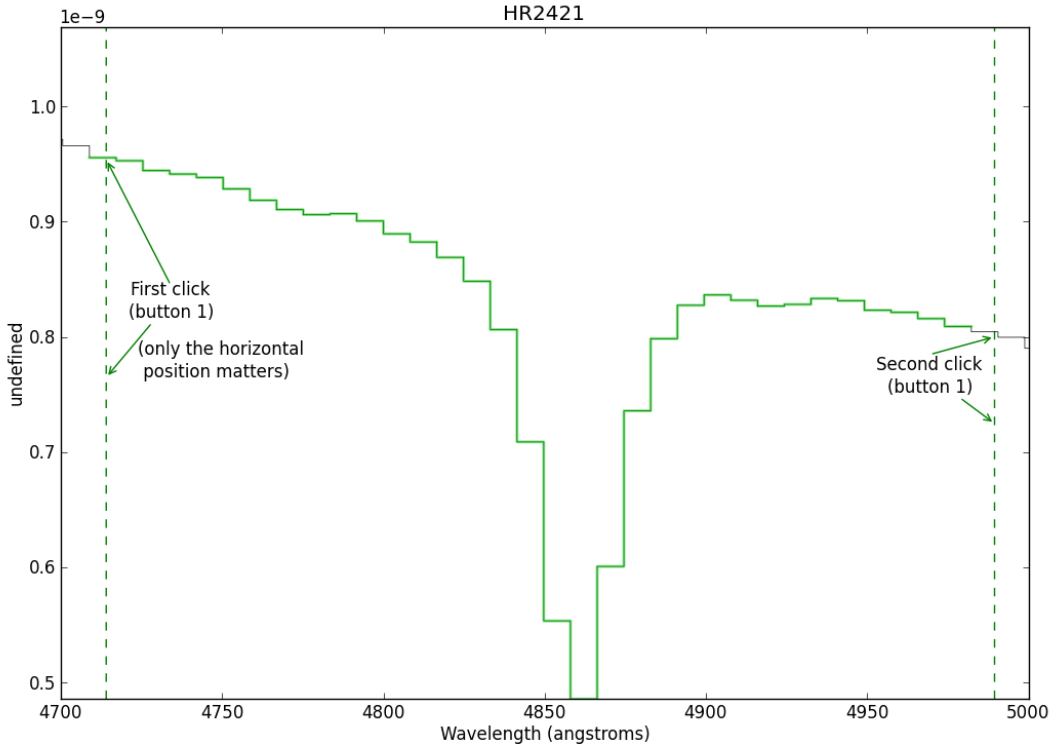
You can also start the interactive fitter by pressing 'b' for baseline or 'f' for fit when a plot window is active, then press '?' to get help. These commands will not work if you have the "zoom" or "pan" tools active, though!

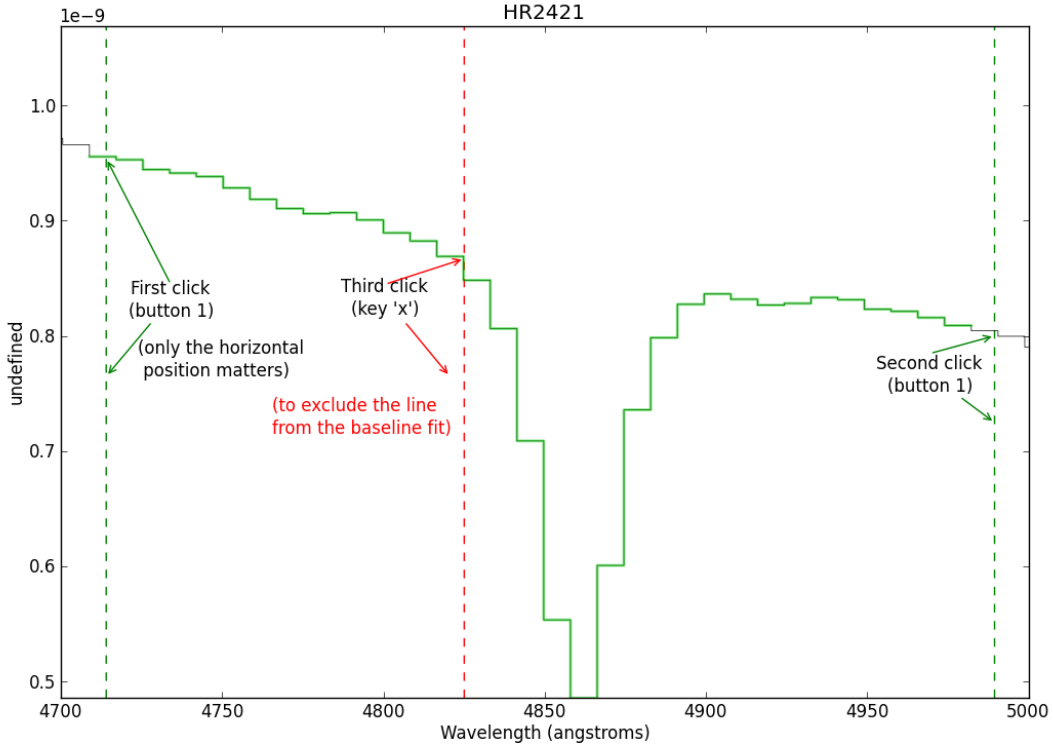
This is where you start the line-fitter:

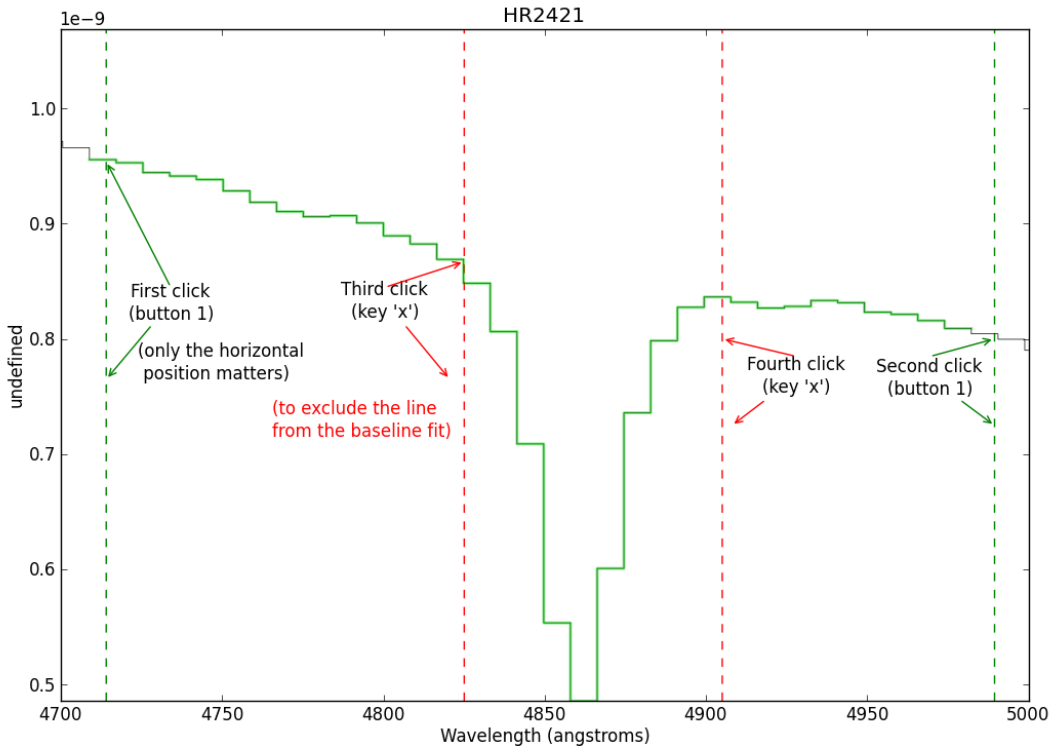
```
# Start up an interactive line-fitting session
sp.specfit(interactive=True)
```

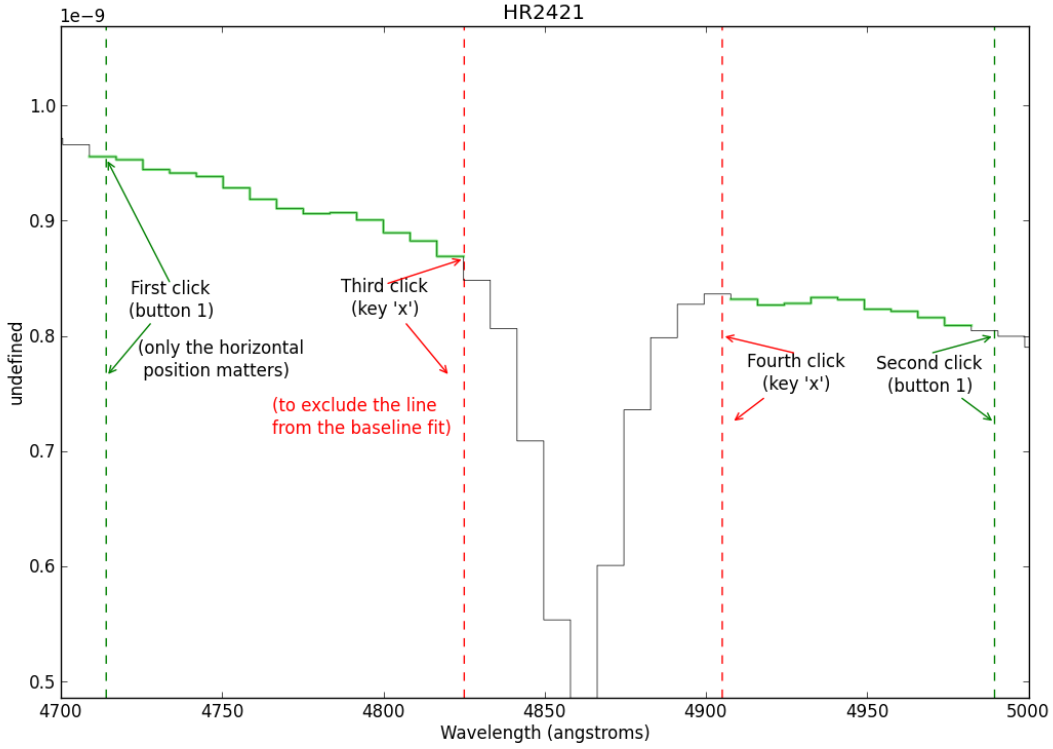

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

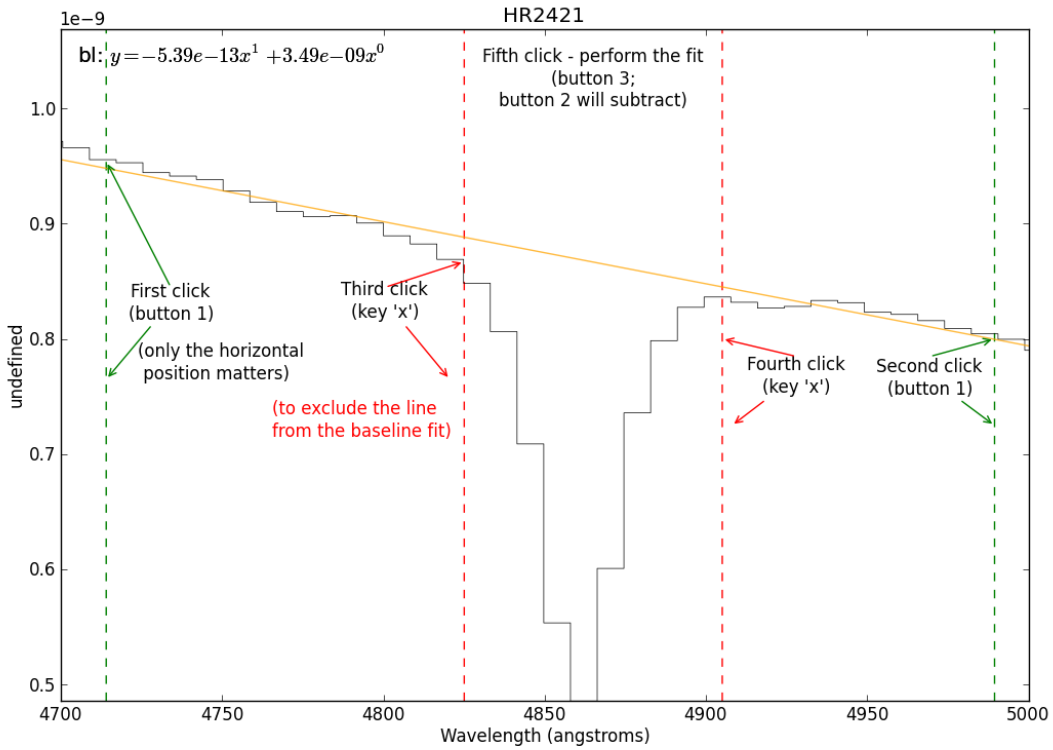

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

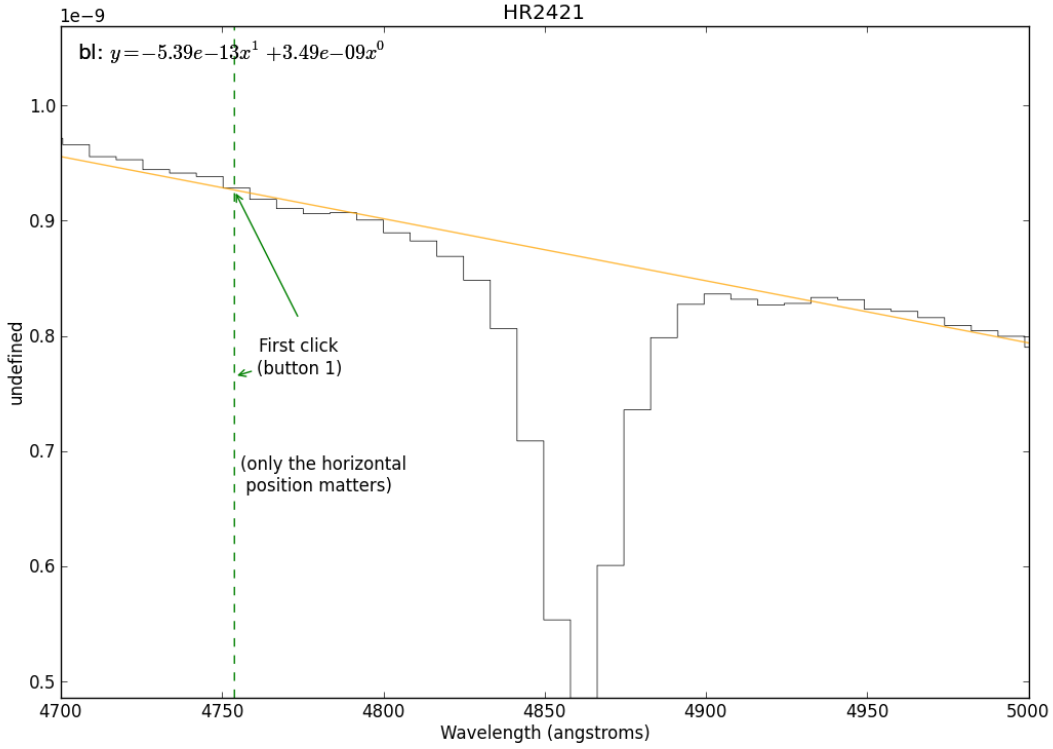

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

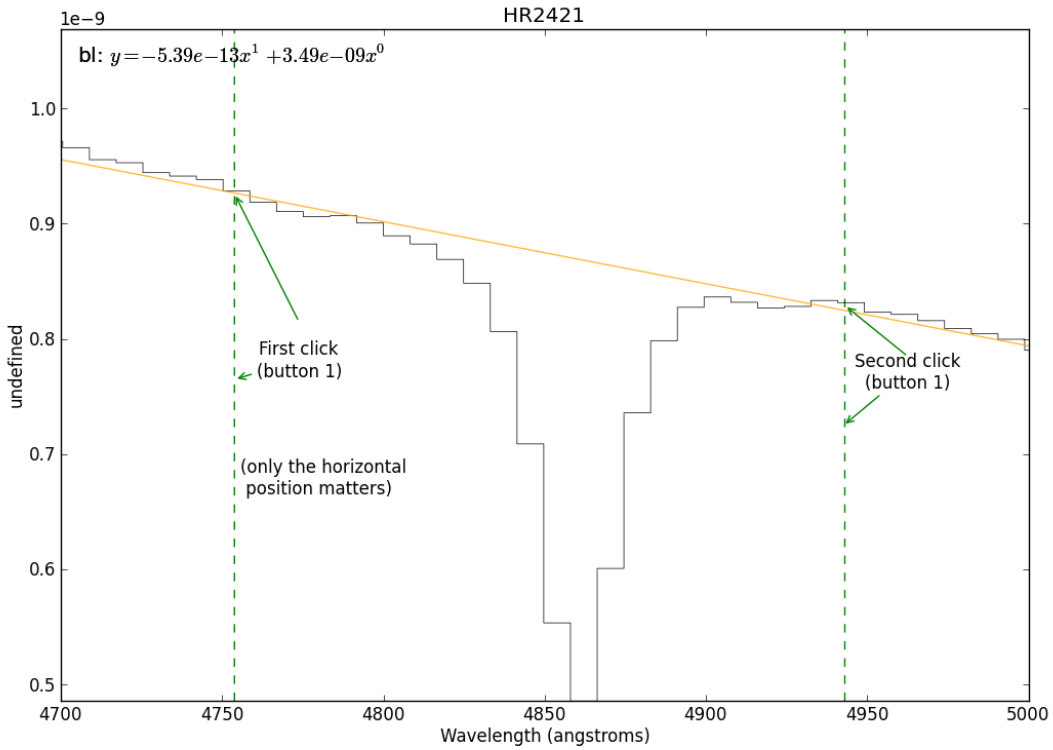

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

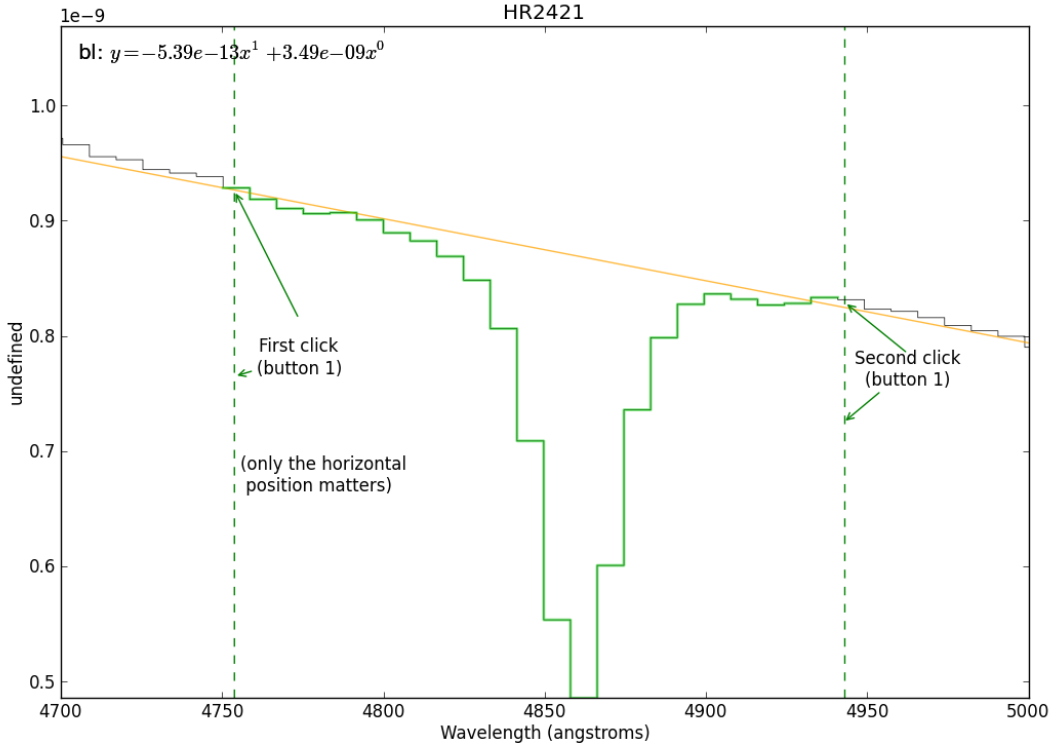

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)

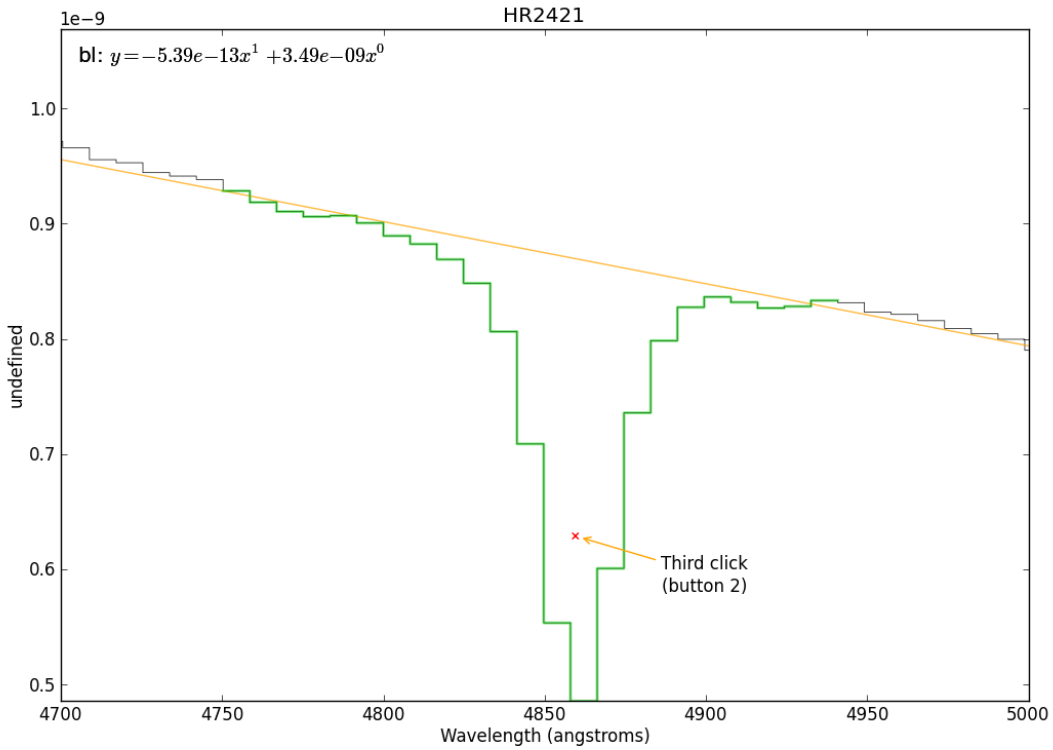

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)

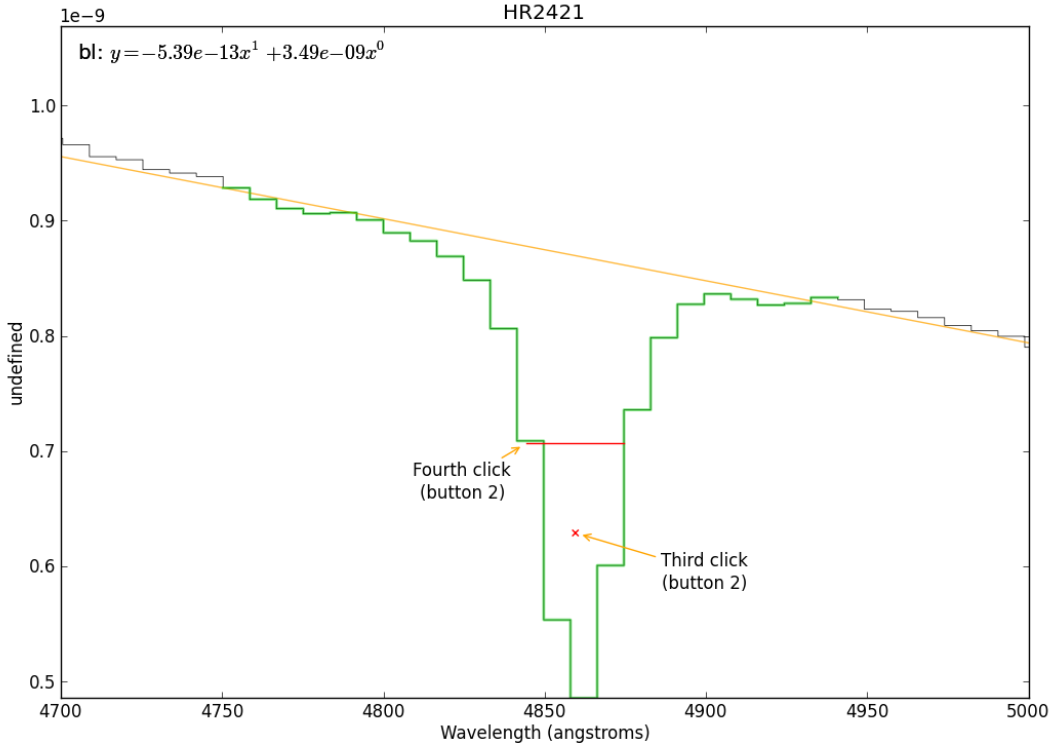

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
Fifth click - perform the fit
(button 3;
button 2 will subtract)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)
undefined
Wavelength (angstroms)

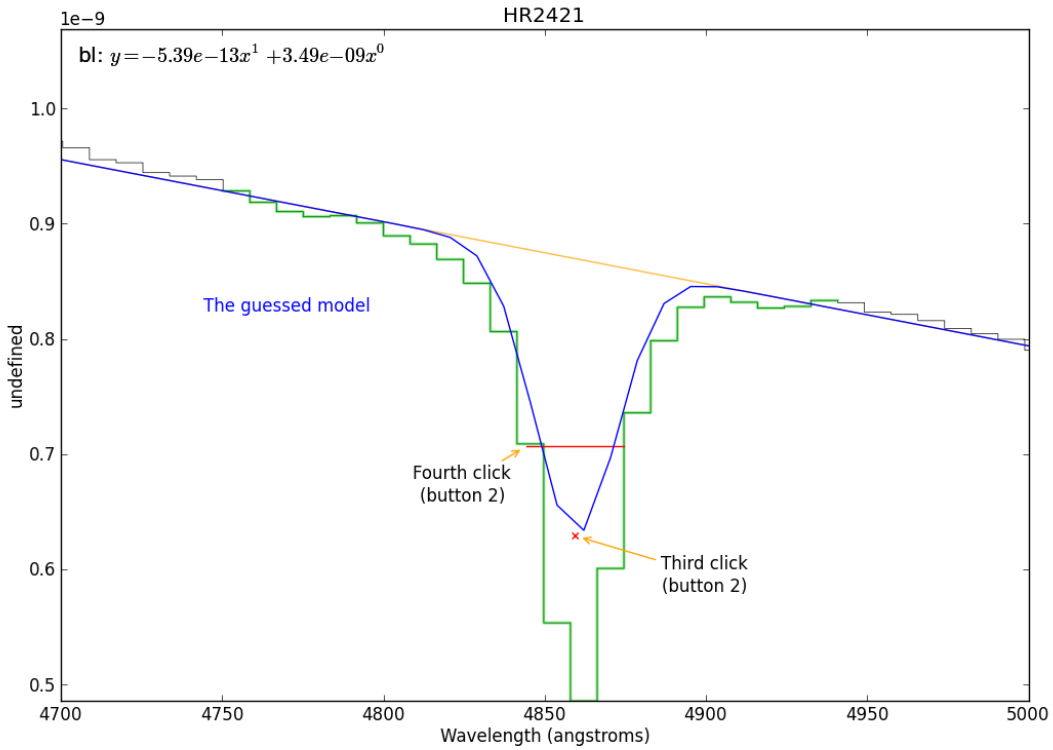

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
(only the horizontal
position matters)
undefined
Wavelength (angstroms)

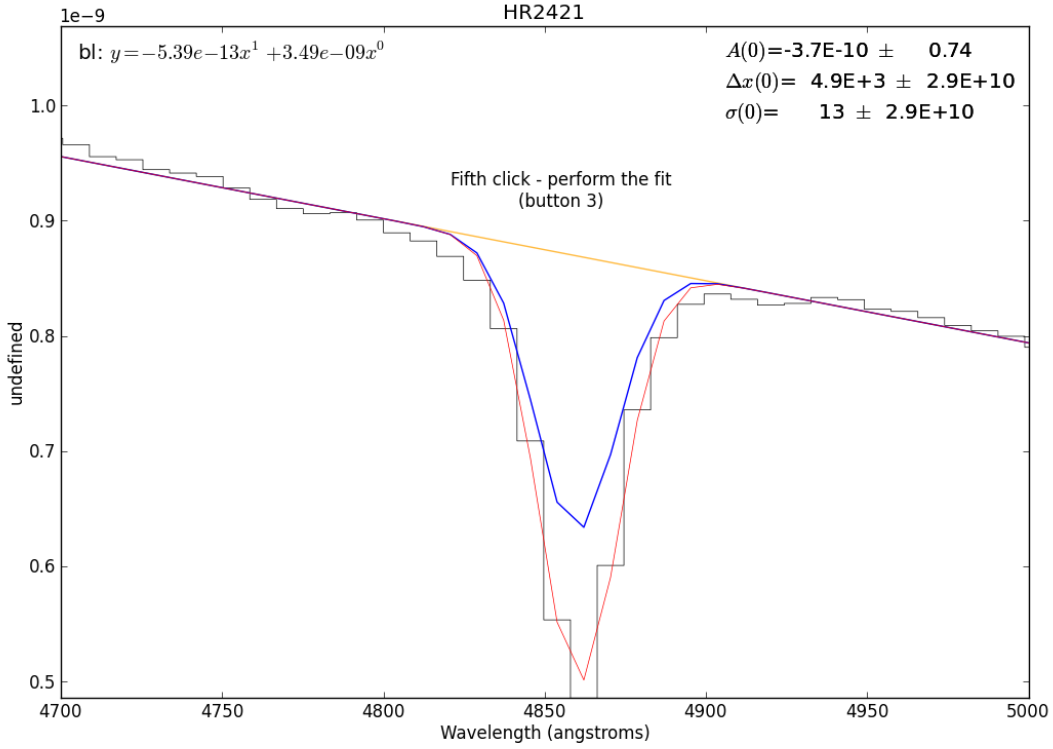

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
Second click
(button 1)
(only the horizontal
position matters)
undefined
Wavelength (angstroms)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
Second click
(button 1)
(only the horizontal
position matters)
undefined
Wavelength (angstroms)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
1.0
0.9
0.8
0.7
0.6
0.5
undefined
Third click
(button 2)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
1.0
0.9
0.8
0.7
0.6
0.5
undefined
Fourth click
(button 2)
Third click
(button 2)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
The guessed model
Fourth click
(button 2)
Third click
(button 2)
undefined
Wavelength (angstroms)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
$A(0)$=-3.7E-10 ± 0.74
$\Delta x(0)$= 4.9E+3 ± 2.9E+10
$\sigma(0)$= 13 ± 2.9E+10
Fifth click - perform the fit
(button 3)
undefined
Wavelength (angstroms)

7.21 Complicated H-alpha Line Fitting

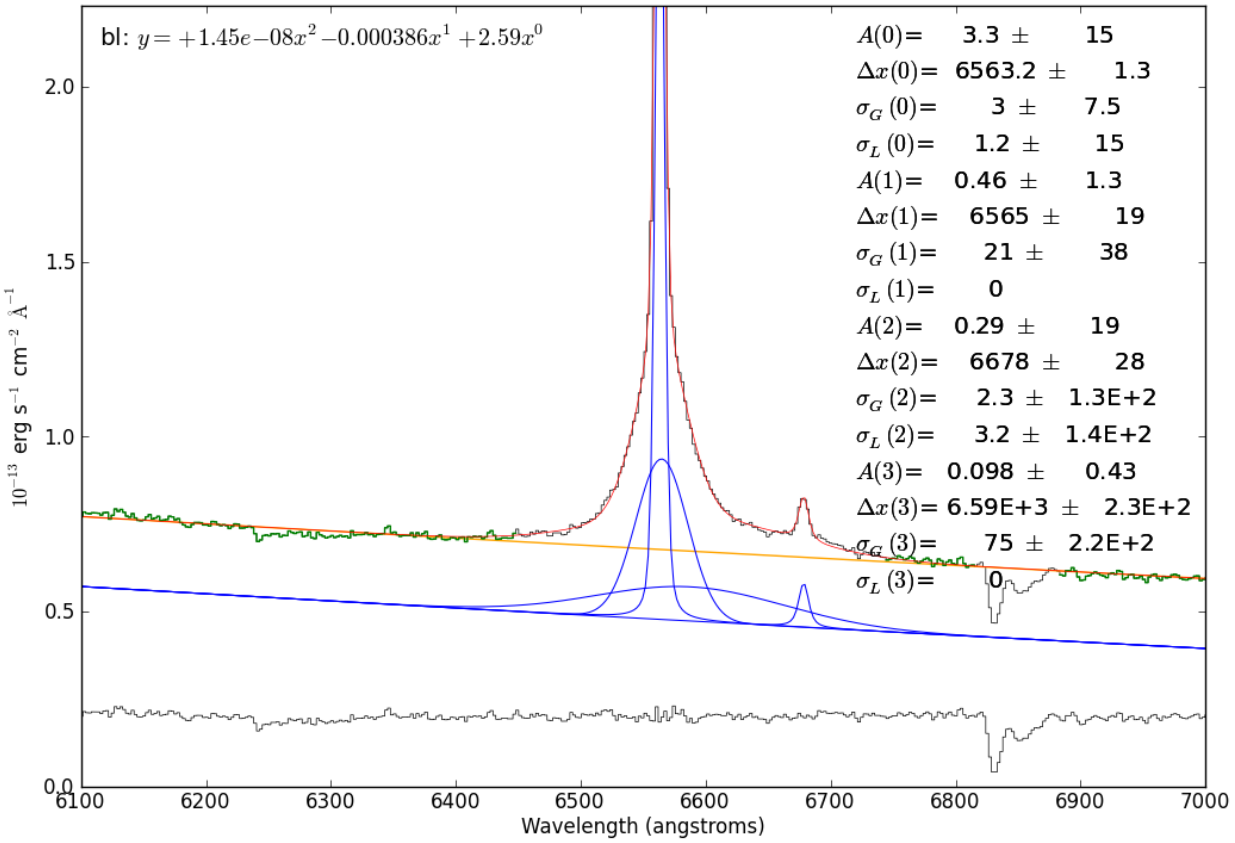
Source code for this example

FITS file for this example

```
"""
Example demonstrating how to fit a complex H-alpha profile after
subtracting off a satellite line (in this case, He I 6678.151704)
"""
import pyspeckit

sp = pyspeckit.Spectrum('sn2009ip_halpha.fits')

# start by plotting a small region around the H-alpha line
sp.plotter(xmin=6100,xmax=7000,ymax=2.23,ymin=0)

# the baseline (continuum) fit will be 2nd order, and excludes "bad"
# parts of the spectrum
# The exclusion zone was selected interactively
# (i.e., cursor hovering over the spectrum)
sp.baseline(xmin=6100, xmax=7000,
        exclude=[6450,6746,6815,6884,7003,7126,7506,7674,8142,8231],
        subtract=False, reset_selection=True, highlight_fitregion=True,
        order=2)

# Fit a 4-parameter voigt (figured out through a series if guess and check fits)
sp.specfit(guesses=[2.4007096541802202,  6563.2307968382256, 3.5653446153950314,  1,
                    0.53985149324131965, 6564.3460908526877, 19.443226155616617,  1,
                    0.11957267912208754, 6678.3853431367716, 4.1892742162283181,  1,
                    0.10506431180136294, 6589.9310414408683, 72.378997529374672,  1,],
                    fittype='voigt')

# Now overplot the fitted components with an offset so we can see them
# the add_baseline=True bit means that each component will be displayed
# with the "Continuum" added
# If this was off, the components would be displayed at y=0
# the component_yoffset is the offset to add to the continuum for plotting
# only (a constant)
sp.specfit.plot_components(add_baseline=True,component_yoffset=-0.2)

# Now overplot the residuals on the same graph by specifying which axis to overplot it on
# clear=False is needed to keep the original fitted plot drawn
# yoffset is the offset from y=zero
sp.specfit.plotresiduals(axis=sp.plotter.axis,clear=False,yoffset=0.20,label=False)

# save the figure
sp.plotter.savefig("SN2009ip_UT121002_Halpha_voigt_zoom.png")
```

```
# print the fit results in table form
# This includes getting the equivalent width for each component using sp.specfit.EQW
print(" ".join(["%15s %15s" % (s,s+"err") for s in sp.specfit.parinfo.parnames]),
      " ".join(["%15s" % ("EQW"+str(i)) for i,w in enumerate(sp.specfit.
```

(continues on next page)

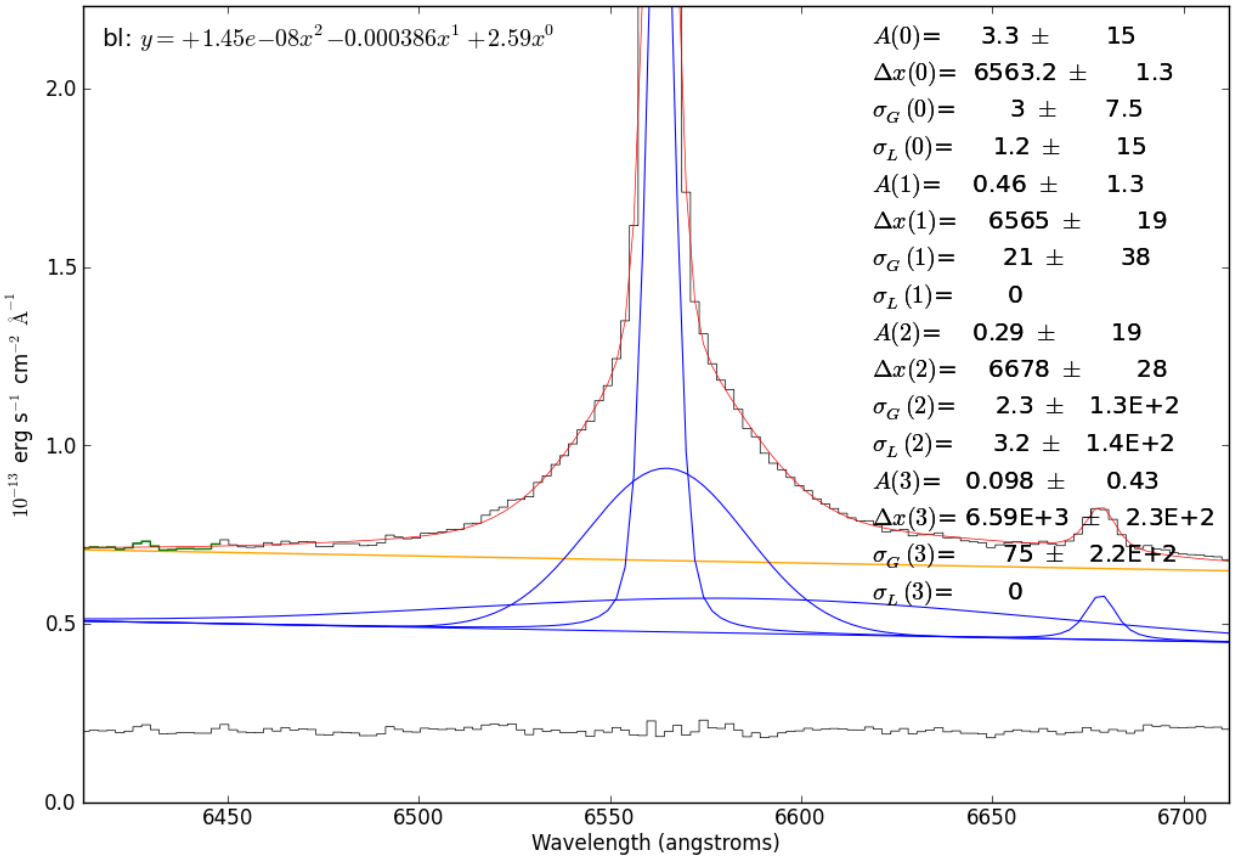


(continued from previous page)

```
↪EQW(components=True))]))
print(" ".join(["%15g %15g" % (par.value,par.error) for par in sp.specfit.parinfo]),
      " ".join(["%15g" % w for w in sp.specfit.EQW(components=True)]))

# zoom in further for a detailed view of the profile fit
sp.plotter.axis.set_xlim(6562-150,6562+150)
sp.plotter.savefig("SN2009ip_UT121002_Halpha_voigt_zoomzoom.png")
```

```
# now we'll re-do the fit with the He I line subtracted off
# first, create a copy of the spectrum
just_halpha = sp.copy()

# Second, subtract off the model fit for the He I component
# (identify it by looking at the fitted central wavelengths)
just_halpha.data -= sp.specfit.modelcomponents[2,:]

# re-plot
just_halpha.plotter(xmin=6100,xmax=7000,ymax=2.00,ymin=-0.3)

# this time, subtract off the baseline - we're now confident that the continuum
# fit is good enough
just_halpha.baseline(xmin=6100, xmax=7000,
        exclude=[6450,6746,6815,6884,7003,7126,7506,7674,8142,8231],
```

(continues on next page)


bl: $y = +1.45e{-}08x^2 - 0.000386x^1 + 2.59x^0$
$A(0)$= 3.3 ± 15
$\Delta x(0)$= 6563.2 ± 1.3
$\sigma_G\,(0)$= 3 ± 7.5
$\sigma_L\,(0)$= 1.2 ± 15
$A(1)$= 0.46 ± 1.3
$\Delta x(1)$= 6565 ± 19
$\sigma_G\,(1)$= 21 ± 38
$\sigma_L\,(1)$= 0
$A(2)$= 0.29 ± 19
$\Delta x(2)$= 6678 ± 28
$\sigma_G\,(2)$= 2.3 ± 1.3E+2
$\sigma_L\,(2)$= 3.2 ± 1.4E+2
$A(3)$= 0.098 ± 0.43
$\Delta x(3)$= 6.59E+3 ± 2.3E+2
$\sigma_G\,(3)$= 75 ± 2.2E+2
$\sigma_L\,(3)$= 0
10^{-13} erg s^{-1} cm^{-2} Å^{-1}
0.0
0.5
1.0
1.5
2.0
6450
6500
6550
6600
6650
6700
Wavelength (angstroms)

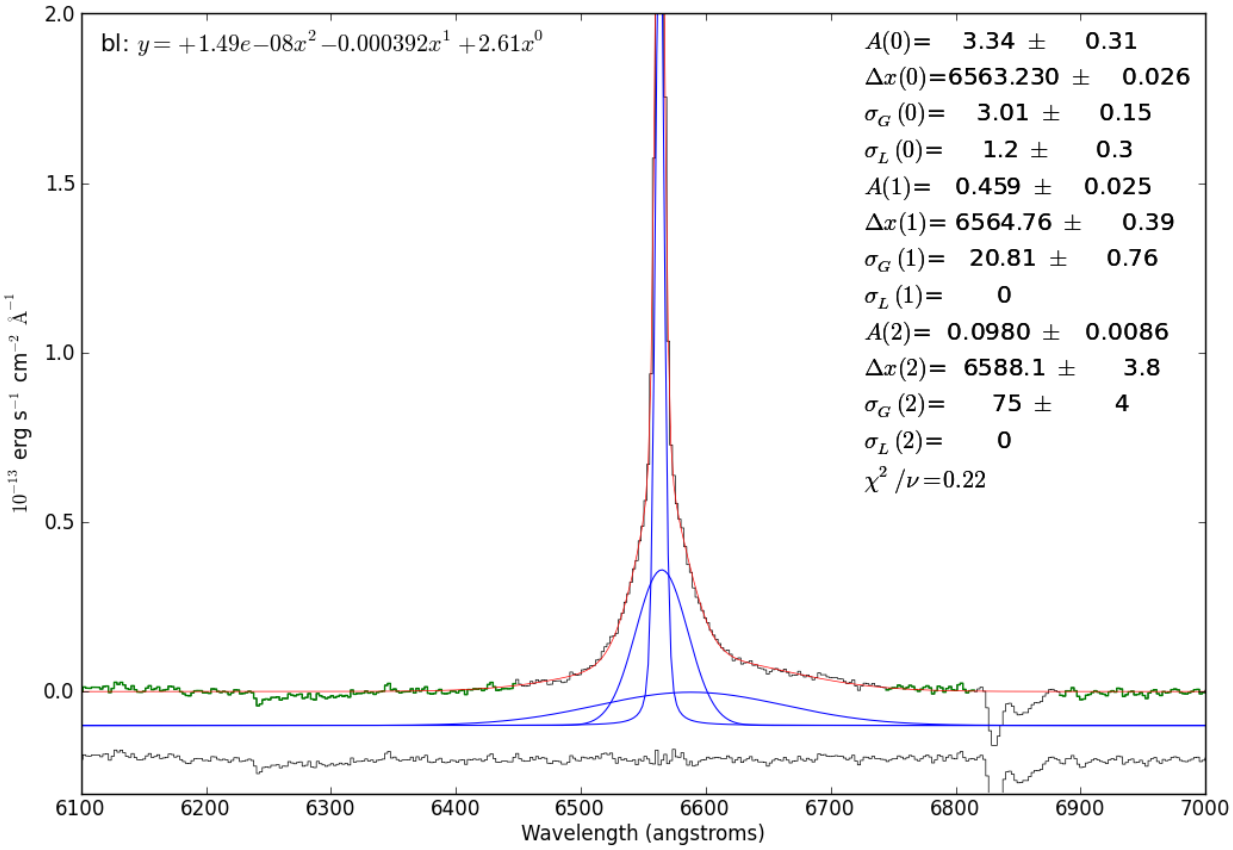
(continued from previous page)

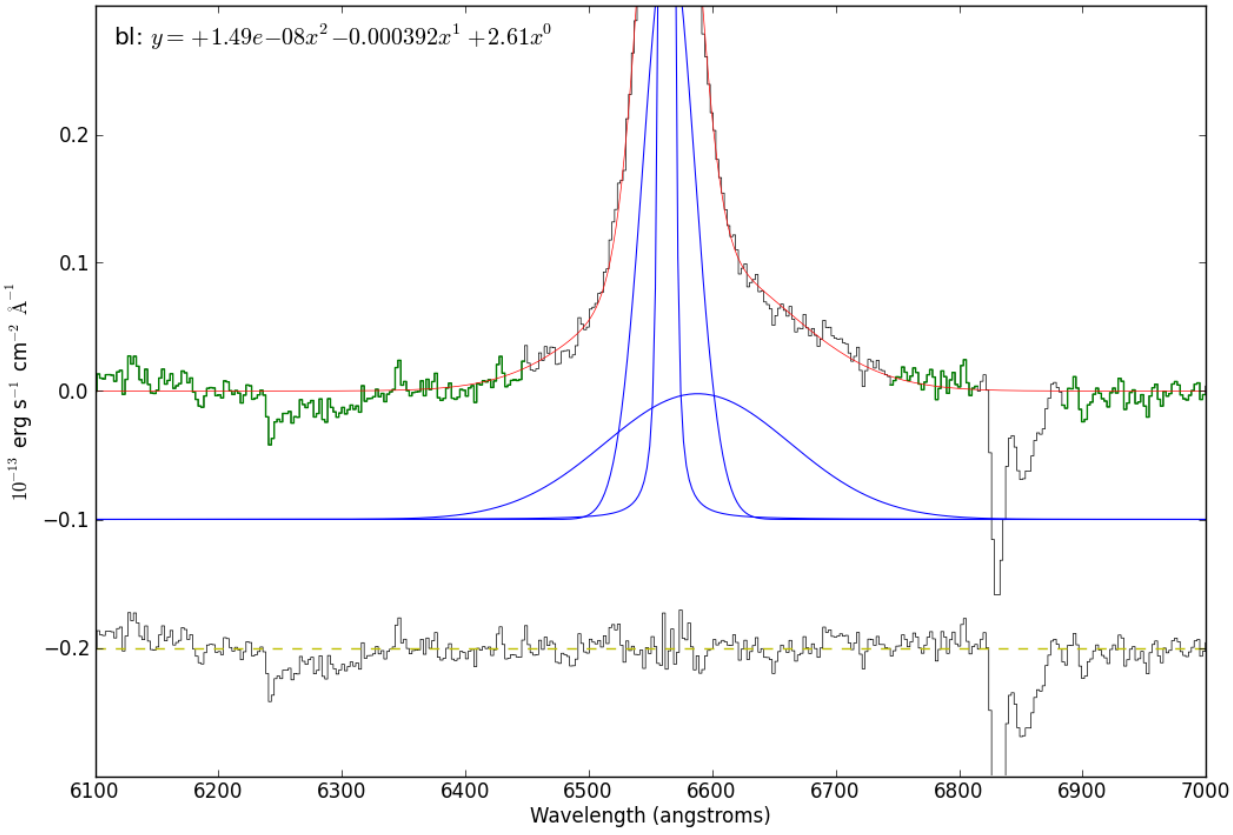
```
        subtract=True, reset_selection=True, highlight_fitregion=True, order=2)

# Do a 3-component fit now that the Helium line is gone
# I've added some limits here because I know what parameters I expect of my fitted line
just_halpha.specfit(guesses=[2.4007096541802202, 6563.2307968382256, 3.5653446153950314, 1,
                    0.53985149324131965, 6564.3460908526877, 19.443226155616617,  1,
                    0.10506431180136294, 6589.9310414408683, 50.378997529374672,  1,],
                    fittype='voigt',
                    xmin=6100,xmax=7000,
                    limitedmax=[False,False,True,True]*3,
                    limitedmin=[True,False,True,True]*3,
                    limits=[(0,0),(0,0),(0,100),(0,100)]*3)

# overplot the components and residuals again
just_halpha.specfit.plot_components(add_baseline=False,component_yoffset=-0.1)
just_halpha.specfit.plotresiduals(axis=just_halpha.plotter.axis,
                                  clear=False, yoffset=-0.20, label=False)

# The "optimal chi^2" isn't a real statistical concept, it's something I made up
# However, I think it makes sense (but post an issue if you disagree!):
# It uses the fitted model to find all pixels that are above the noise in the spectrum
# then computes chi^2/n using only those pixels
just_halpha.specfit.annotate(chi2='optimal')

# save the figure
just_halpha.plotter.savefig("SN2009ip_UT121002_Halpha_voigt_threecomp.png")
```

```
# A new zoom-in figure
import pylab

# now hide the legend
just_halpha.specfit.fitleg.set_visible(False)
# overplot a y=0 line through the residuals (for reference)
pylab.plot([6100,7000],[-0.2,-0.2],'y--')
# zoom vertically
pylab.gca().set_ylim(-0.3,0.3)
# redraw & save
pylab.draw()
just_halpha.plotter.savefig("SN2009ip_UT121002_Halpha_voigt_threecomp_zoom.png")
```

```
# Part of the reason for doing the above work is to demonstrate that a
# 3-component fit is better than a 2-component fit
#
# So, now we do the same as above with a 2-component fit

just_halpha.plotter(xmin=6100,xmax=7000,ymax=2.00,ymin=-0.3)
just_halpha.specfit(guesses=[2.4007096541802202, 6563.2307968382256, 3.5653446153950314, 1,
                    0.53985149324131965, 6564.3460908526877, 19.443226155616617,  1],
                    fittype='voigt')
just_halpha.specfit.plot_components(add_baseline=False,component_yoffset=-0.1)
just_halpha.specfit.plotresiduals(axis=just_halpha.plotter.axis,
```

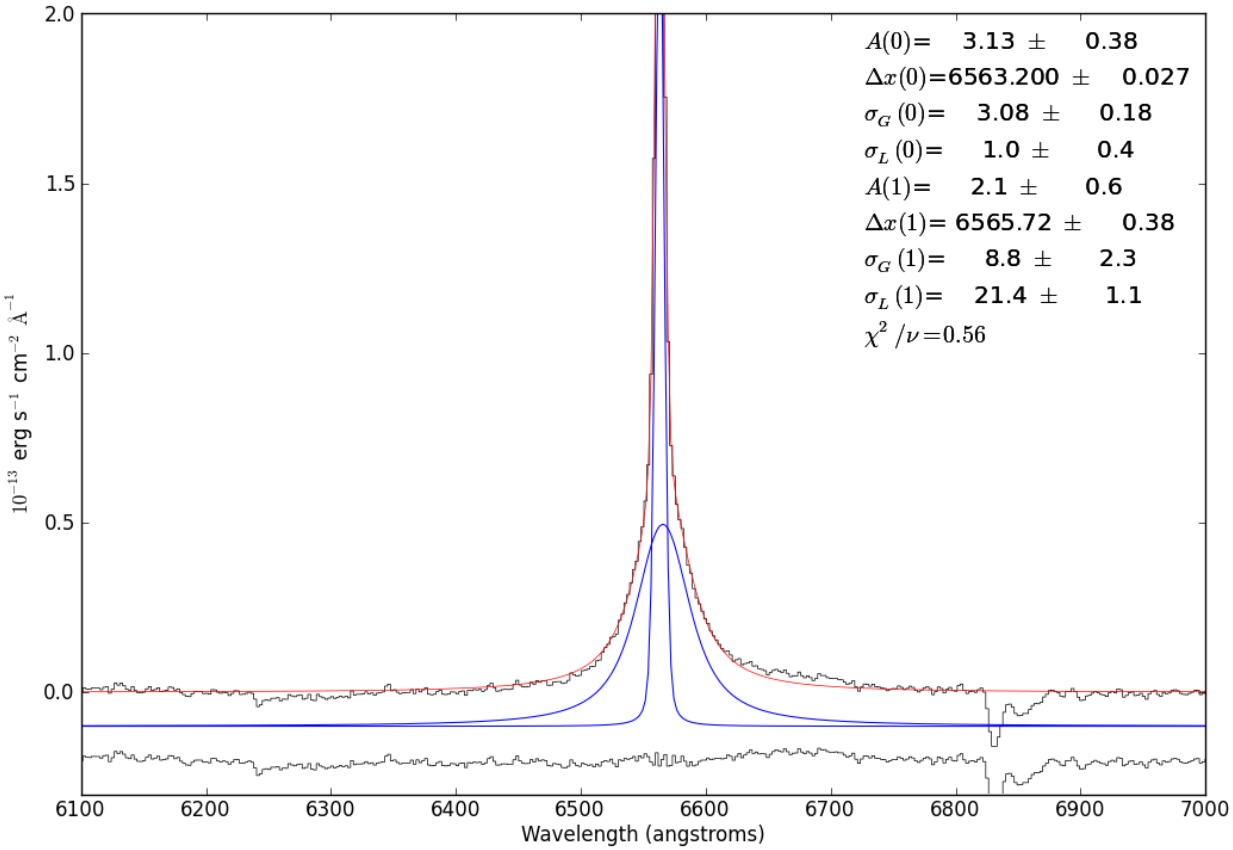
(continues on next page)


bl: $y = +1.49e{-}08x^2 - 0.000392x^1 + 2.61x^0$
$A(0)$= 3.34 ± 0.31
$\Delta x(0)$=6563.230 ± 0.026
$\sigma_G(0)$= 3.01 ± 0.15
$\sigma_L(0)$= 1.2 ± 0.3
$A(1)$= 0.459 ± 0.025
$\Delta x(1)$= 6564.76 ± 0.39
$\sigma_G(1)$= 20.81 ± 0.76
$\sigma_L(1)$= 0
$A(2)$= 0.0980 ± 0.0086
$\Delta x(2)$= 6588.1 ± 3.8
$\sigma_G(2)$= 75 ± 4
$\sigma_L(2)$= 0
χ^2/ν=0.22
10^{-13} erg s^{-1} cm^{-2} Å^{-1}
Wavelength (angstroms)


bl: $y = +1.49e{-}08x^2 - 0.000392x^1 + 2.61x^0$
10^{-13} erg s^{-1} cm^{-2} Å^{-1}
Wavelength (angstroms)

(continued from previous page)

```
                                clear=False, yoffset=-0.20,label=False)
just_halpha.specfit.annotate(chi2='optimal')
just_halpha.plotter.savefig("SN2009ip_UT121002_Halpha_voigt_twocomp.png")
```



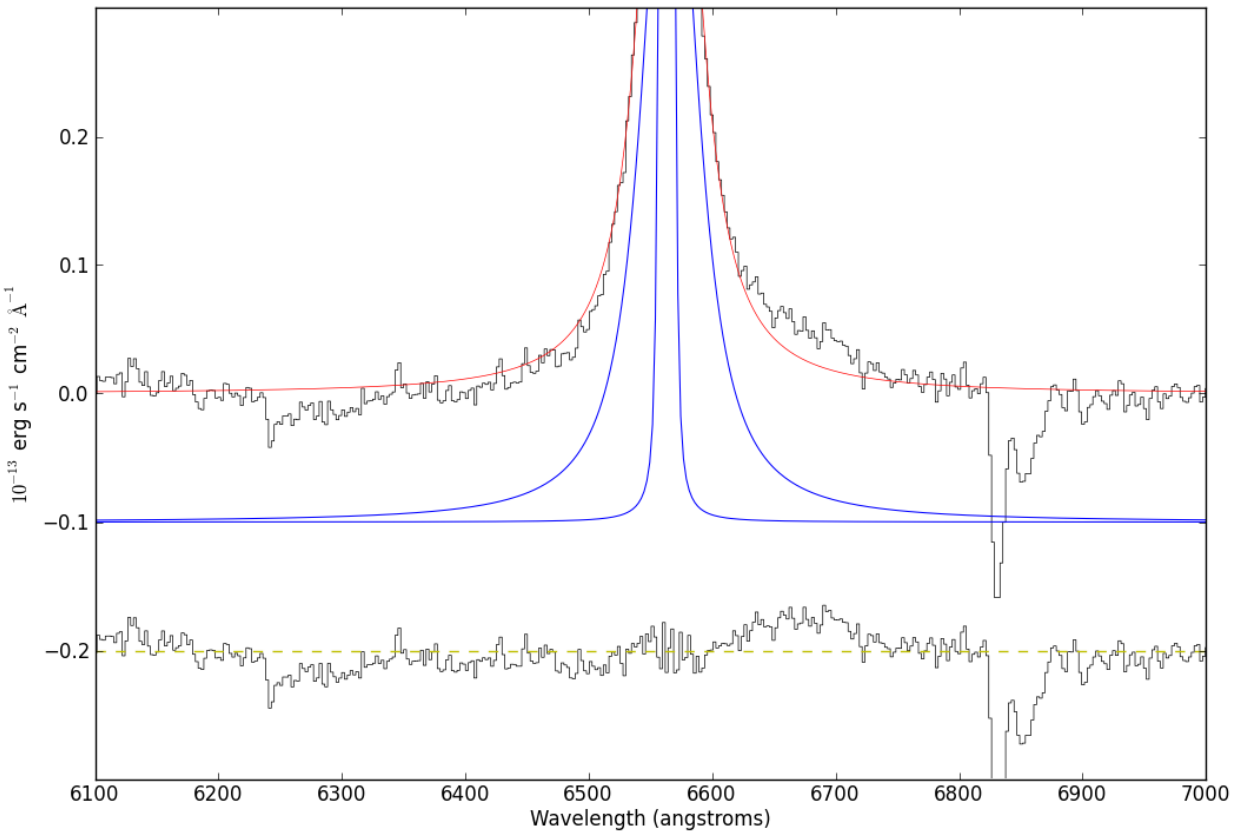

```
just_halpha.specfit.fitleg.set_visible(False)
pylab.plot([6100,7000],[-0.2,-0.2],'y--')
pylab.gca().set_ylim(-0.3,0.3)
pylab.draw()
just_halpha.plotter.savefig("SN2009ip_UT121002_Halpha_voigt_twocomp_zoom.png")
```

7.22 MUSE cube fitting

An example analysis of a MUSE data cube using both spectral_cube and pyspeckit

```
from astropy.io import fits
from astropy import units as u
import numpy as np
import spectral_cube
```

(continues on next page)



(continued from previous page)

```
import pyspeckit
from spectral_cube.spectral_axis import vac_to_air

lines = {'OI6300': 6302.046,
         'SIII6313': 6313.8,
         'OI6363': 6365.536,
         'NII6548': 6549.85,
         'HAlpha': 6564.61,
         'HBeta': 4862.69,
         'NII6584': 6585.28,
         'HeI6678': 6679.99556,
         'SII6716': 6718.29,
         'SII6731': 6732.67,
         'pa20': 8394.71,
         'pa19': 8415.63,
         'pa18': 8440.27,
         'pa17': 8469.59,
         'pa16': 8504.83,
         'pa15': 8547.73,
         'pa14': 8600.75,
         'pa13': 8667.40,
         'pa12': 8752.86,
         'pa11': 8865.32,
         'pa10': 9017.8 ,
```

(continues on next page)

(continued from previous page)

```
         'pa9' : 9232.2 ,
         'HeI7067': 7067.138,
         'HeI7283': 7283.355,
         'ArIII7135': 7137.8,
         'ArIII7751': 7753.2,
         'SIII9071':9071.1,
         'NeII5756':5756.24,
         'HeI5877':5877.3,
         'OIII5008': 5008.24,
         'OII4960': 4960.3,
}

# use spectral_cube to read the data
cube = spectral_cube.SpectralCube.read('CUBEec_nall.fits', hdu=1)
cont = cube.spectral_slab(6380*u.AA, 6500*u.AA).apply_numpy_function(np.mean, axis=0)

# "slabs" are velocity-cutouts of the cube
# Cube velocity conversion should use vacuum wavelengths
slabs = {line:
         cube.with_spectral_unit(u.km/u.s, velocity_convention='optical',
                                 rest_value=wl*u.AA)
             .spectral_slab(-200*u.km/u.s, 250*u.km/u.s)
         for line,wl in lines.items()}

# Compute 1st moments (intensity-weighted velocity)
for line,slab in slabs.items():
    print "kms",line
    mom1 = slab.moment1(axis=0)
    mom1.write('moments/moment1_125_{0}_kms.fits'.format(line), overwrite=True)
mean_moment = np.mean([fits.getdata('moments/moment1_125_{0}_kms.fits'.format(line)) for
                       line in slabs], axis=0)
hdr = fits.getheader('moments/moment1_125_{0}_kms.fits'.format(line))
outfilename = 'moments/moment1_125_mean_kms.fits'
if astropy.version.major >= 2 or (astropy.version.major==1 and astropy.version.minor>=3):
    fits.PrimaryHDU(data=mean_moment, header=hdr).writeto(outfilename, overwrite=True)
else:
    fits.PrimaryHDU(data=mean_moment, header=hdr).writeto(outfilename, clobber=True)

# Create a cube of many different lines all in velocity
# This will allow us to measure a velocity more accurate than is possible
# with one line alone (assuming all lines have the same velocity) because
# MUSE undersamples its line-spread-function a little
newcube_shape = (sum(s.shape[0] for s in slabs.values()),) + slabs.values()[0].shape[1:]
newcube_spaxis = np.concatenate([s.spectral_axis
                                 for s in slabs.values()]).value*u.km/u.s
sortvect = newcube_spaxis.argsort()
sortspaxis = newcube_spaxis[sortvect]

newcube = np.empty(newcube_shape)

# normalize each spectrum
ind = 0
```

(continues on next page)

(continued from previous page)

```
for ii,slab in enumerate(slabs.values()):
    data = (slab.filled_data[:] - cont) / (slab.sum(axis=0) - cont*slab.shape[0])
    newcube[ind:ind+data.shape[0], :, :] = data
    ind += data.shape[0]

supercube = newcube[sortvect, :, :]

# Create a pyspeckit cube so we can then fit a gaussian to each spectrum
pxarr = pyspeckit.units.SpectroscopicAxis(sortspaxis.value, units='km/s')
pcube = pyspeckit.Cube(cube=supercube, xarr=pxarr)

# more cores = more faster
pcube.fiteach(fittype='gaussian', guesses=[1/np.sqrt(np.pi), 10, 50.0],
              errmap=np.ones(supercube.shape[1:])/10., multicore=40)

pcube.write_fit('velocity_fits_125.fits', overwrite=True)
```

7.23 Fitting using a Template

Pyspeckit allows you to use a spectral template as the model to fit to your spectrum. See `pyspeckit.spectrum.models.template`.

If your model spectrum only requires a shift and a scale, it's easy to use:

```
from pyspeckit.spectrum.models.template import template_fitter

template = pyspeckit.Spectrum('template_spectrum.fits')
dataspec = pyspeckit.Spectrum("DataSpectrum.fits")

# Create the fitter from the template spectrum and "Register" it
template_fitter = template_fitter(template,xshift_units='angstroms')
dataspec.Registry.add_fitter('template',template_fitter,2)

# The fitted parameters are amplitude & xshift
# perform the fit:
dataspec.specfit(fittype='template',guesses=[1,0])

# print the results
print dataspec.specfit.parinfo
```

7.24 Monte Carlo examples

There are (at least) two packages implementing Monte Carlo sampling available in python: pymc and emcee. pyspeckit includes interfaces to both. With the pymc interface, it is possible to define priors that strictly limit the parameter space. So far that is not possible with emcee.

The examples below use a custom plotting package from agpy. It is a relatively simple but convenient wrapper around numpy's histogram2d. pymc_plotting takes care of indexing, percentile determination, and coloring.

The example below shows the results of a gaussian fit to noisy data (S/N ~ 6). The parameter space is then explored with pymc and emcee in order to examine the correlation between width and amplitude.

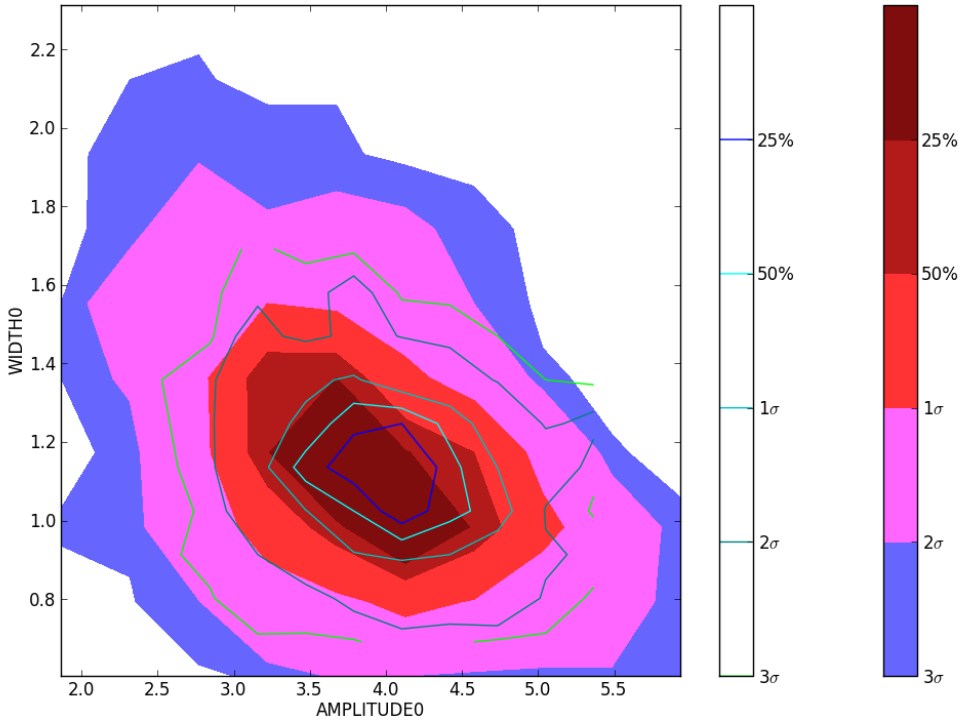
```
import pyspeckit
import numpy as np

# Create our own gaussian centered at 0 with width 1, amplitude 5, and
# gaussian noise with amplitude 1
x = pyspeckit.units.SpectroscopicAxis(np.linspace(-10,10,50), unit='km/s')
e = np.random.randn(50)
d = np.exp(-np.asarray(x)**2/2.)*5 + e

# create the spectrum object
sp = pyspeckit.Spectrum(data=d, xarr=x, error=np.ones(50)*e.std())
# fit it
sp.specfit(fittype='gaussian', guesses=[1,0,1])
# then get the pymc values
MCuninformed = sp.specfit.get_pymc()
MCwithpriors = sp.specfit.get_pymc(use_fitted_values=True)
MCuninformed.sample(101000,burn=1000,tune_interval=250)
MCwithpriors.sample(101000,burn=1000,tune_interval=250)

# MC vs least squares:
print(sp.specfit.parinfo)
# Param #0   AMPLITUDE0 =         4.51708 +/-        0.697514
# Param #1       SHIFT0 =       0.0730243 +/-        0.147537
# Param #2       WIDTH0 =        0.846578 +/-        0.147537   Range:   [0,inf)


print(MCuninformed.stats()['AMPLITUDE0'],MCuninformed.stats()['WIDTH0'])
#  {'95% HPD interval': array([ 2.9593463 ,  5.65258618]),
#   'mc error': 0.0069093803546614969,
#   'mean': 4.2742994714387068,
#   'n': 100000,
#   'quantiles': {2.5: 2.9772782318342288,
#    25: 3.8023115438555615,
#    50: 4.2534542126311479,
#    75: 4.7307441549353229,
#    97.5: 5.6795448148793293},
#   'standard deviation': 0.68803712503362213},
#  {'95% HPD interval': array([ 0.55673242,  1.13494423]),
#   'mc error': 0.0015457954546501554,
#   'mean': 0.83858499779600593,
#   'n': 100000,
#   'quantiles': {2.5: 0.58307734425381375,
```

(continues on next page)

(continued from previous page)

```
#    25: 0.735072721596429,
#    50: 0.824695077252244,
#    75: 0.92485225882530664,
#    97.5: 1.1737067304111048},
#   'standard deviation': 0.14960171537498618}


print(MCwithpriors.stats()['AMPLITUDE0'],MCwithpriors.stats()['WIDTH0'])
#  {'95% HPD interval': array([ 3.45622857,  5.28802497]),
#   'mc error': 0.0034676818027776788,
#   'mean': 4.3735547007147595,
#   'n': 100000,
#   'quantiles': {2.5: 3.4620369729291913,
#    25: 4.0562790782065052,
#    50: 4.3706408236777481,
#    75: 4.6842793868186332,
#    97.5: 5.2975444315549947},
#   'standard deviation': 0.46870135068815683},
#  {'95% HPD interval': array([ 0.63259418,  1.00028015]),
#   'mc error': 0.00077504289680683364,
#   'mean': 0.81025043433745358,
#   'n': 100000,
#   'quantiles': {2.5: 0.63457050661326331,
#    25: 0.7465422649464849,
#    50: 0.80661741451336577,
#    75: 0.87067288601310233,
#    97.5: 1.0040591994661381},
#   'standard deviation': 0.093979950317277294}



# optional
from mpl_plot_templates import pymc_plotting
import pylab
pymc_plotting.hist2d(MCuninformed,'AMPLITUDE0','WIDTH0',clear=True,bins=[25,25])
pymc_plotting.hist2d(MCwithpriors,'AMPLITUDE0','WIDTH0',contourcmd=pylab.contour,colors=[(0,
↪1,0,1),(0,0.5,0.5,1),(0,0.75,0.75,1),(0,1,1,1),(0,0,1,1)],clear=False,bins=[25,25])
pylab.plot([5],[1],'k+',markersize=25, zorder=1000)
pylab.savefig("pymc_example_Gaussian_amplitude_vs_width.pdf")

# Now do the same with emcee
emcee_ensemble = sp.specfit.get_emcee()
p0 = emcee_ensemble.p0 * (np.random.randn(*emcee_ensemble.p0.shape) / 10. + 1.0)
pos,logprob,state = emcee_ensemble.run_mcmc(p0,100000)

plotdict = {'AMPLITUDE0':emcee_ensemble.chain[:,:,0].ravel(),
            'WIDTH0':emcee_ensemble.chain[:,:,2].ravel()}
pymc_plotting.hist2d(plotdict,'AMPLITUDE0','WIDTH0',fignum=2,bins=[25,25],clear=True)
pylab.plot([5],[1],'k+',markersize=25)
pylab.savefig("emcee_example_Gaussian_amplitude_vs_width.pdf")
```

The Amplitude-Width parameter space sampled by pymc with (lines) and without (solid) priors. There is moderate anticorrelation between the line width and the peak amplitude. The + symbol indicates the input parameters; the model



does a somewhat poor job of recovering the true values (in case you're curious, there is no intrinsic bias - if you repeat the above fitting procedure a few hundred times, the mean fitted amplitude is 5.0).

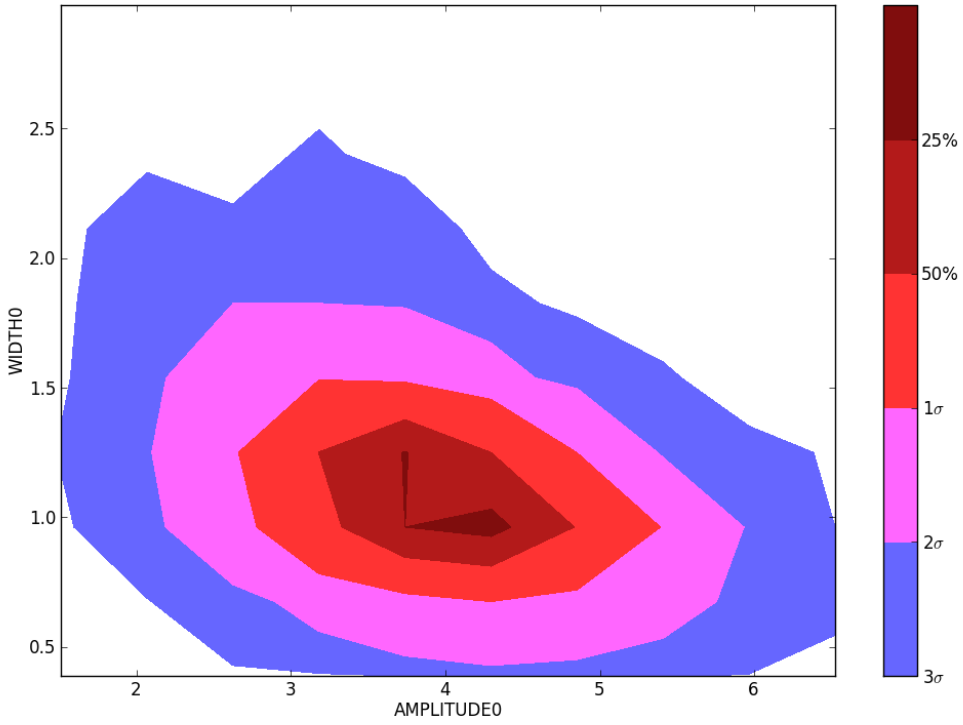
The parameter space sampled with `emcee` and binned onto a 25x25 grid. Note that `emcee` has 6x as many points (and takes about 6x as long to run) because there are 6 "walkers" for the 3 parameters being fit.

7.25 Ammonia Monte Carlo examples

This example shows the same process as in example_pymc, but for the ammonia model. Here we emphasize the degeneracy between the excitation temperature and the total column density.

```
import numpy as np
from astropy import units as u
import itertools
from operator import itemgetter
import pyspeckit
import scipy.stats

from pyspeckit.spectrum.models import ammonia, ammonia_constants

import pylab as pl
```

(continues on next page)



(continued from previous page)

```
pl.close('all')
pl.figure(1).clf()

oneonefreq = ammonia_constants.freq_dict['oneone']
twotwofreq = ammonia_constants.freq_dict['twotwo']
# create an axis that covers the 1-1 and 2-2 inversion lines
xaxis = pyspeckit.units.SpectroscopicAxis(np.linspace(oneonefreq*(1-50/3e5),
                                                      twotwofreq*(1+50/3e5),
                                                      1000.),
                                          unit=u.Hz)
sigma = 2.
center = 0.
trot = 15.
ntot = 14.7
# Adopting an NLTE model: T_ex < T_rot (this case is better for the default fitter)
# pyradex.Radex(species='p-nh3', column=5e14, collider_densities={'h2':5000},␣
↪temperature=15)()[8:10]
tex = {'oneone':8.5, 'twotwo':5.5}
synth_data = ammonia.ammonia(xaxis, trot=trot, tex=tex, width=sigma,
                             xoff_v=center, ntot=ntot)

# Add noise
stddev = 0.1
noise = np.random.randn(xaxis.size)*stddev
```

(continues on next page)

(continued from previous page)

```
error = stddev*np.ones_like(synth_data)
data = u.Quantity(noise+synth_data, u.K)

# create the spectrum object
sp = pyspeckit.Spectrum(data=data,
                        xarr=xaxis,
                        error=np.ones_like(synth_data)*stddev)
# fit it
# Setting limits is important for running the MCMC chains below (they set the
# priors)
sp.specfit(fittype='ammonia',
           # trot, tex, ntot, width, center, fortho
           guesses=[35,10,15,5,5,0],
           fixed=[False,False,False,False,False,True],
           minpars=[3,3,10,0,-10,0],
           maxpars=[80,80,20,10,10,1],
           limitedmin=[True]*6,
           limitedmax=[True]*6,
          )
# then get the pymc values
MCuniformpriors = sp.specfit.get_pymc()
MCuniformpriors.sample(10100,burn=100,tune_interval=250)

# MC vs least squares:
print(sp.specfit.parinfo)

print(MCuniformpriors.stats()['tex0'],
      MCuniformpriors.stats()['ntot0'])



# optional plotting
from mpl_plot_templates import pymc_plotting
import pylab
pylab.figure(1).clf()
pymc_plotting.hist2d(MCuniformpriors, 'tex0', 'ntot0',
                     bins=[25,25])
pylab.plot([tex['oneone']],[ntot],'k+',markersize=25)
pylab.axis([6,10.5,14.6,14.9])
pylab.savefig("tex_vs_ntot_pymc_example.pdf")


# Now do the same with emcee
emcee_ensemble = sp.specfit.get_emcee()
p0 = emcee_ensemble.p0 * (np.random.randn(*emcee_ensemble.p0.shape) / 50. + 1.0)
pos,logprob,state = emcee_ensemble.run_mcmc(p0,10000)

plotdict = {'tex0':emcee_ensemble.chain[:,1000:,1].ravel(),
            'ntot0':emcee_ensemble.chain[:,1000:,2].ravel()}
# one of the samplers went off the hook and got locked at high ntot
OK = (plotdict['ntot0'] < 20)
plotdict['tex0'] = plotdict['tex0'][OK]
```

(continues on next page)

(continued from previous page)

```
plotdict['ntot0'] = plotdict['ntot0'][OK]

pylab.figure(2).clf()
pymc_plotting.hist2d(plotdict, 'tex0', 'ntot0', fignum=2, bins=[25,25],
                     clear=True)
pylab.plot([tex['oneone']],[ntot],'k+',markersize=25)
pylab.axis([6,10.5,14.6,14.9])
pylab.savefig("tex_vs_ntot_emcee_example.pdf")
```

7.26 Guide for GILDAS-CLASS users

7.26.1 Reading Files

PySpecKit can read many file types including CLASS .cls files.

For example, a typical single-dish observing file will consist of a large number of individual spectra. These may be on-the-fly spectra or multiple integrations on a single source; either way the pointing and spectral information for each individual data point should be incorporated in the data file.

The spectra are recorded from the data file into an `pyspeckit.classes.ObsBlock` object. An ObsBlock (Observation Block) is simply a container for many spectra; in most regards it behaves the same way as a `pyspeckit.Spectrum` object, but it has a few extra attributes built-in (e.g., spectral averaging).

A file can be loaded as a named object. The data is filtered as it is read in. The following line will ensure that the spectra in the `n2hp` ObsBlock contain only data from the F1M spectrometer with lines labeled 'N2HP(3-2)' or 'N2H+(3-2)':

```
from pyspeckit.spectrum.readers.read_class import class_to_obsblocks
n2hp = class_to_obsblocks(filename,
    telescope=['SMT-F1M-HU','SMT-F1M-VU'],
    line=['N2HP(3-2)','N2H+(3-2)'])
```

7.26.2 Working with the data

You can treat an ObsBlock like any other `pyspeckit.Spectrum` object, but there are a few unique features.

The 'smooth' function will smooth each individual spectrum in the ObsBlock, which can be useful if you want to smooth before averaging.

The 'average' function averages spectra. Weights for the averaging can be specified. The error is also computed either by taking the RMS of the averaged spectra or by averaging the error spectra.

You can plot each spectrum in an individual window with the `ploteach` function, or all overlaid simultaneously with the 'plotter' function. If you want to loop through each spectrum, waiting for user input after displaying, you can do something like:

```
ax = pylab.gca()
for sp in n2hp:
    sp.plotter(axis=ax)
    input("Waiting for input...")
```

You can also fit a line profile to *each* spectrum in the observation block using the `fiteach` command.

7.26.3 Selecting Spectra

There are 3 keywords that can be used to select spectra when reading in a file. The `line` and `telescope` keywords are required in order to make an observation block, otherwise the spectral axes will not be common to all spectra:

```
n2hp = class_to_obsblocks(filename,
    telescope=['SMT-F1M-HU','SMT-F1M-VU'],
    line=['N2HP(3-2)','N2H+(3-2)'])
```

If you want to read in all of the data and don't care about the line or telescope, you can instead use:

```
all_data = class_to_spectra(filename)
```

You can select data after they are read in by matching header keywords:

```
g10 = pyspeckit.ObsBlock([sp for sp in all_data if sp.specname == 'g10'])
```

7.26.4 Fitting Data

In CLASS, you would specify the data range with individual commands on the command line, e.g.:

```
[this isn't quite right]
xrange 5 25
```

In pyspeckit, you can specify the range in multiple ways, but the default is to use the plotted window. For example:

```
sp.plotter(xmin=5, xmax=25)
sp.baseline()
sp.specfit()
```

will perform a baseline fit and spectrum fit over the range 5-25 km/s (the units of xmin, xmax are the plotted units). More intricate specifications are possible:

```
sp.plotter()
# Fit a baseline over the region 5-25 km/s, excluding 7-10 km/s and 15-18 km/s
sp.baseline.selectregion(xmin=5,xmax=25,exclude=[7,10,15,18])
sp.baseline()
sp.specfit()
```

7.27 Guide for IRAF users

PySpecKit is similar in intent and implementation to IRAF's *splot* routine. IRAF users will probably want to do most of their data reduction (i.e., turning an image/cube into a 1D wavelength-calibrated spectrum) in IRAF, but will be comfortable fitting lines and making publication-quality plots using PySpecKit.

7.27.1 Loading a Spectrum

If you have an IRAF spectrum, it is straightforward to load into PySpecKit:

```
sp = pyspeckit.Spectrum('iraf_spectrum.ms.fits')
sp.plotter()
```

7.27.2 Fitting Line Profiles

Note: See *A guide to interactive fitting* for a comprehensive graphical demonstration of these instructions.

In IRAF, a line profile is fitted using *k* to start the fitter, then *k*, *l*, or *v* to perform the fit.

In PySpecKit, the continuum (*baseline*) and line profile are determined separately.

Instead of using a key twice to specify the continuum level, a continuum must be fitted from the data. This is done by pressing *b* to turn on the baseline fitter. Click or press *1* to select baseline regions - they will be highlighted in green. Press *3* to fit the baseline and display it as an orange line.

In PySpecKit, the interactive fitter is started by pressing *f* in the plot window. After pressing *f*, instructions will be provided in the terminal window telling you which line profiles are implemented. Select one of these, or use a gaussian by default.

Select the line fitting region by pressing *1* on either side of the line. Select the peak and full-width-half-maximum of the line by pressing *2* at each of these locations in turn. You may repeat this cycle indefinitely to fit multiple profiles (comparable to IRAF*s *deblend* capability). Then, press *3* to perform the fit.

The fitted parameters can be accessed (as variables, or printed) through the `Spectrum.specfit.parinfo` parameter. They will also be displayed in the plot legend.

7.28 PySpecKit Projects

A few projects appropriate for a Google Summer of Code or similar are outlined below.

7.28.1 Refactor and Expand the pyspeckit modeling tools

Since the development of pyspeckit, there has been substantial progress on a more general class of modeling tools from astropy. Pyspeckit already has a wide variety of data fitting and modeling tools that can readily be modified to use the astropy modeling formalism.

Details of this project need to be worked out, but will include:

- refactoring pyspeckit.models to use astropy.models
- building a graphical interface to astropy.models

7.28.2 Refactor and Expand the Unit Test suite

Pyspeckit is a complicated code suite, which has led to many bugs, particularly in the UI. An improved unit test suite would help prevent or remove these bugs. Such a project would start by breaking down the existing tests, which are really end-to-end tests, into their component units.

The refactor would be to use astropy's wrappers around pytest to make the testing smoother and easier to extend. The current model has tests in a different repository because the data got too big, but we can and should have the *data* in a different repository, but the tests all in one place.

7.28.3 Incorporate astropy.units into pyspeckit.units

This project was done by an ESO-hosted summer student, Dinos Kousidis.

This project is at the core of both `pyspeckit` and specutils.

The most important base functionality of a spectroscopic toolkit is to be able to convert between different spectroscopic systems, e.g. wavelength<->frequency<->velocity. This can be achieved using astropy's unit equivalencies.

The X-axis unit changes will be straightforward project that should require about 2 weeks to complete. The more complicated and interesting project is creating Y-axis units (i.e., flux units) that appropriately adjust with changes to the X-axis. These would make use of other astropy unit equivalencies, e.g. the spectral density equivalency.

The end goal will be to have a `Spectrum` object that will live in specutils and be inherited by `pyspeckit`, which will provide the interface to modeling and graphical tools.

Additionally, here are some real-world examples. They are not guaranteed to work, but they show the use of pyspeckit in pipelines and paper-producing code.

- Fit each spectrum in a VLA NH3 6-6 cube (Goddi, Ginsburg, Zhang 2016)
- Fit each spectrum in an Arecibo H2CO 1-1 cube (Ginsburg et al, 2015)
- Crop an ALMA CH3CN cube, then fit each spectrum
- Fit each spectrum in an APEX H2CO 3-2 cube with 1 and 2 components (Ginsburg et al, 2016)
- Compute moments, then fit each spectrum in a MUSE cube (McCleod et al, 2015)
- Fit H2 and BrG in a TripleSpec NIR cube (Youngblood et al, 2016)
- Fit NH3 1-1 and 2-2 spectra for the GBT Ammonia Survey (Pineda, Friesen et al)

CHAPTER

EIGHT

FREQUENTLY ASKED QUESTIONS

8.1 I see "UnitConversionError: 'km / s' (speed) and 'GHz' (frequency) are not convertible" errors when using the ammonia fitter

For versions of pyspeckit 0.16 (May 2015) and later, pyspeckit uses astropy's units for the spectroscopic axis. It therefore requires an `equivalency` to be defined.

To create a `SpectroscopicAxis` with the appropriate equivalency defined, the axis must have a reference frequency (`refX`) and a `velocity_convention`.:

```
>>> from astropy import units as u
>>> from pyspeckit.spectrum.units import SpectroscopicAxis
>>> import numpy as np
>>> xarr = SpectroscopicAxis(np.linspace(22,24)*u.GHz,
                             refX=23*u.GHz,
                             velocity_convention='radio')
```

If you have loaded a spectrum from file and it doesn't contain the appropriate metadata (usually a `CTYPE` in the header), you can set the `refX` and `velocity_convention` manually. The options for `velocity_convention` are `radio`, `optical`, and `relativisitic`.

For details on the meaning of the various velocity conventions, see Frank Ghigo's site or FITS Paper III, especially table 4. For details on the accepted FITS `CTYPE` keywords, see FITS Paper III. In particular, their table 1 specifies all valid spectral coordinate type codes.

8.2 No plot appears when I run plot commands

Plots appearing and not appearing depends on your global matplotlib configuration.

If you are already in a python session, you can enable interactive plotting by doing:

The `.ion()` call turns interactive mode on. If it wasn't on when you made the plot, you need to call `.show()` next. If you call `.ion()` before calling any pyspeckit plot command, though, the `.show()` is not required.

If you want to make sure this is enabled automatically on startup, you can start your python session with

(I tend to use `ipython --pylab`, but it's deprecated)

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

N

O

P

R

S

T

U

V

W

X