pyspeckit Documentation

Release 0.1.13

Adam Ginsburg and Jordan Mirocha

September 10, 2013

An extensible spectroscopic analysis toolkit for astronomy.

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

Download the January 2012 version, the latest commit, see *Installation and Requirements*, or see our pypi entry.

Supported file types and their formats:

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

CHAPTER

ONE

GUIDES / GETTING STARTED

- **guide_class** A simple getting started guide aimed at Gildas-CLASS users
- **guide_iraf** Intended for users of IRAF's `splot` interactive fitting routine.

CHAPTER

TWO

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 Spectrums. 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.

CHAPTER

THREE

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

3.1 Installation and Requirements

Hint: You can *easy_install* or *pip_install* pyspeckit:

```
pip install pyspeckit
easy_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

- astrop >=0.1
- pyfits >=2.4
- atpy (which depends on asciitable [github link])
- hdf5

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

```
pip install pyspeckit
```

Or the most recent version:

```
pip install https://bitbucket.org/pyspeckit/pyspeckit.bitbucket.org/get/tip.tar.gz
```

You can acquire the code with this clone command:

```
hg clone https://bitbucket.org/pyspeckit/pyspeckit.bitbucket.org pyspeckit
cd pyspeckit
python setup.py install
```

or if you use git:

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

Or you can Download the latest tarball version, then extract and install using the standard python method:

```
wget https://bitbucket.org/pyspeckit/pyspeckit.bitbucket.org/get/tip.tar.gz
tar -xzf pyspeckit-pyspeckit.bitbucket.org-tip.tar.gz
cd pyspeckit-pyspeckit.bitbucket.org-tip
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`.

3.2 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'),
        multisingle=multisingle,
        )
```

Then you can register these fitters.

3.2.1 API Documentation for Models

class pyspeckit.spectrum.models.model.**SpectralModel**(*modelfunc*, *npars*, *shortvarnames=('A', '\Delta x', '\sigma')*, *multisingle='multi'*, *fitunits=None*, *centroid_par=None*, *fwhm_func=None*, *fwhm_pars=None*, *integral_func=None*, *use_lmfit=False*, ***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

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... hmm (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

fitunits : 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

multisingle : list (optional)

Are there multiple peaks (no background will be fit) or just a single peak (a background may/will be fit)

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,
>>> 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

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=
>>> 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

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

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

Ammonia inversion transition TKIN fitter translated from Erik Rosolowsky's http://svn.ok.ubc.ca/svn/signals/nh3fit/

Module API

pyspeckit.spectrum.models.ammonia.**ammonia**(*xarr*, *tkin=20*, *tex=None*, *ntot=100000000000000.0*, *width=1*, *xoff_v=0.0*, *fortho=0.0*, *tau=None*, *fillingfraction=None*, *return_tau=False*, *thin=False*, *verbose=False*, *return_components=False*, *debug=False*)

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

ntot can be specified as a column density (e.g., 10^15) or a log-column-density (e.g., 15)

tex can be specified or can be assumed LTE if unspecified, if tex>tkin, or if "thin" is specified

"thin" uses a different parametetrization and requires only the optical depth, width, offset, and tkin to be specified. In the 'thin' approximation, tex is not used in computation of the partition function - LTE is implicitly assumed

If tau is specified, ntot is NOT fit but is set to a fixed value fillingfraction is an arbitrary scaling factor to apply to the model fortho is the ortho/(ortho+para) fraction. The default is to assume all ortho. xoff_v is the velocity offset in km/s

tau refers to the optical depth of the 1-1 line. The optical depths of the other lines are fixed relative to tau_oneone

(not implemented) if tau is specified, ntot is ignored

Adds a variable height (background) component to any model 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

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

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*, *grid_vwidth=1.0*, *grid_vwidth_scale=False*, *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

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

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.

pyspeckit.spectrum.models.inherited_gaussfitter.**gaussian**(*x*, *A*, *dx*, *w*, *return_components=False*, *normalized=False*)

Returns a 1-dimensional gaussian of form A*numpy.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 **x** : np.ndarray

array of x values

A : 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

w : float

Width of the gaussian (sigma)

return_components : bool

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

normalized : bool

Return a normalized Gaussian?

pyspeckit.spectrum.models.inherited_gaussfitter.**gaussian_fitter**(*multisingle='multi'*)
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**(*multisingle='multi'*)
Generator for Gaussian fitter class

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

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

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

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

pyspeckit.spectrum.models.hill5infall.**hill5_model**(*xarr*, *tau*, *v_lsr*, *v_infall*, *sigma*, *tpeak*, *TBG=2.73*)
The rest of this needs to be translated from C

pyspeckit.spectrum.models.hill5infall.**jfunc**(*t*, *nu*)
t- kelvin nu - Hz?

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

line_names is a LIST of the line names to be used as indices for the dictionaries

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

freq_dict - frequencies of the indvidual transitions

line_strength_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.

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_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_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

The simplest and most useful model.

Until 12/23/2011, lorentzian fitting used the complicated and somewhat bloated gaussfitter.py code. Now, this is a great example of how to make your own model!

pyspeckit.spectrum.models.inherited_lorentzian.**lorentzian**(*x*, *A*, *dx*, *w*, *return_components=False*)

Returns a 1-dimensional lorentzian of form A*2*pi*w/((x-dx)**2 + ((w/2)**2))

[amplitude,center,width]

return_components does nothing but is required by all fitters

pyspeckit.spectrum.models.inherited_lorentzian.**lorentzian_fitter**(*multisingle='multi'*)
Generator for lorentzian fitter class

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

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 at 0x1021b45f0>*)
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

N2H+ fitter

Reference for line params: Daniel, F., Dubernet, M.-L., Meuwly, M., Cernicharo, J., Pagani, L. 2005, MNRAS 363, 1083

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

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

Does not yet implement: http://adsabs.harvard.edu/abs/2010ApJ...716.1315K

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

pyspeckit.spectrum.models.n2hp.**relative_strength_total_degeneracy** = **{'121-011': 9.0, '121-010': 9.0, '1**
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.inherited_voigtfitter.**voigt**(*xarr*, *amp*, *xcen*, *sigma*, *gamma*, *normalized=False*)

Normalized 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

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 :

pyspeckit.spectrum.models.inherited_voigtfitter.**voigt_fitter**(*multisingle='multi'*)

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

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 [HummerStorey1987]. They are also computed in [Hummer1994] and tabulated at a [wiki]. I had to OCR and pull out by hand some of the coefficients.

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.**rrl**(*n*, *dn=1*, *amu=1.007825*)
compute Radio Recomb Line freqs in GHz from Brown, Lockman & Knapp ARAA 1978 16 445

3.2.2 Model Documentation Table of Contents

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', multisingle='multi', 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, anda 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', multisingle='multi', 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 Hα-[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)
```

3.3 Features

3.3.1 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.

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)

__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)

__init__(*Spectrum*)

__module__ = 'pyspeckit.spectrum.baseline'

annotate(*loc='upper left'*)

button2action(*event=None*, *debug=False*, *subtract=True*, *powerlaw=None*, *fit_original=False*, *baseline_fit_color='orange'*, ***kwargs*)
Do the baseline fitting and save and plot the results.

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*)

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

plot_baseline (*annotate=True*, *baseline_fit_color=(1, 0.65, 0, 0.75)*, *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 ()
Set the baseline spectrum based on the fitted parameters

set_spectofit (*fit_original=True*)
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

3.3.2 Model Fitting

class `pyspeckit.spectrum.fitters.`**Specfit** (*Spectrum*, *Registry=None*)
Bases: `pyspeckit.spectrum.interactive.Interactive`

EQW (*plot=False*, *plotcolor='g'*, *fitted=True*, *continuum=None*, *components=False*, *annotate=False*, *alpha=0.5*, *loc='lower left'*, *xmin=None*, *xmax=None*)
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.

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

Returns Equivalent Width, or widths if components=True :

add_sliders(*parlimitdict=None*, ***kwargs*)
Add a Sliders window in a new figure appropriately titled

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.

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*, *frameon=False*, *chi2=None*, ***kwargs*)
Add a legend to the plot showing the fitted parameters

_clearlegend() will remove the legend

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

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

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

event_manager (*event*, *debug=False*)
Decide what to do given input (click, keypress, etc.)

firstclick_guess ()
Initialize self.guesses

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.

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*, *units='pixels'*)
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 :

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 lines; if the true FWHM is known (i.e., the line is well-represented by a single gauss/voigt/lorentz profile), use that instead!

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 :

moments(***kwargs*)

Return the moments

see the `moments` module

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

Fit multiple gaussians (or other profiles)

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

renormalize - 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

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*, *nsigcut_moments=None*, *plot=True*, ***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 singlefitters 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*, ***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

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

selectregion_interactive(*event*, *mark_include=True*, *debug=False*, ***kwargs*)

select regions for baseline fitting

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.

3.3.3 Measurements

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

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.

Currently will only work with Gaussians. To generalize: 1. make sure we manipulate modelpars correctly, i.e. read in entries corresponding to wavelength/frequency/whatever correctly.

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.

`Measurements.`**`bisection`**(*f*, *x_guess*)
Find root of function using bisection method. Absolute tolerance of 1e-4 is being used.

`Measurements.`**`bracket_root`**(*f*, *x_guess*, *atol=0.0001*)
Bracket root by finding points where function goes from positive to negative.

`Measurements.`**`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.

`Measurements.`**`compute_flux`**(*pars*)
Calculate integrated flux of emission line. Works for multi-component fits too. Unnormalized.

`Measurements.`**`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.

`Measurements.`**`compute_luminosity`**(*pars*)
Determine luminosity of line (need distance and flux units).

`Measurements.`**`derive`**()
Calculate luminosity and FWHM for all spectral lines.

`Measurements.`**`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

Measurements.**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.

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

Measurements.**to_tex**()
Write out fit results to tex format.

3.3.4 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**
Bases: numpy.ndarray

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

as_unit(*unit*, *frame=None*, *quiet=True*, *center_frequency=None*, *center_frequency_units=None*, *debug=False*, *xtype_check='check'*, ***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.

***unit* [string]** What unit do you want to 'view' the array as? None returns the x-axis unchanged (NOT a copy!)

***frame* [None]** NOT IMPLEMENTED. When it is, it will allow you to convert between LSR, topocentric, heliocentric, rest, redshifted, and whatever other frames we can come up with. Right now the main holdup is finding a nice python interface to an LSR velocity calculator... and motivation.

center_frequency [None | float] *center_frequency_units* [None | string]

If converting between velocity and any other spectroscopic type, need to specify the central frequency around which that velocity is calculated. I think this can also accept wavelengths....

***xtype_check* ['check' | 'fix']** Check whether the xtype matches the units. If 'fix', will set the xtype to match the units.

cdelt(*tolerance=1e-08*, *approx=False*)
Return the channel spacing if channels are linear

Parameters **tolerance** : float

Tolerance in the difference between pixels that determines how near to linear the xarr must be

approx : bool

Return the mean DX even if it is inaccurate

change_frame (*frame*)
Change velocity frame

convert_to_unit (*unit*, ***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.units = 'km/s' xarr.refX = 5.0 xarr.refX_units = GHz xarr.coord_to_x(6000,'GHz') == 5.1 # GHz

in_frame (*frame*)
Return a shifted xaxis

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.

coordinate_location **['left', 'center', 'right']** Does the coordinate mark the left, center, or right edge of the pixel? If 'center' or 'left', the *last* pixel will have the same dx as the second to last pixel. If right, the *first* pixel will have the same dx as the second pixel.

umax (*units=None*)
Return the maximum value of the SpectroscopicAxis. If units specified, convert to those units first

umin (*units=None*)
Return the minimum value of the SpectroscopicAxis. If units specified, convert to those units first

x_in_frame (*xx*, *frame*)
Return the value 'x' shifted to the target frame

x_to_coord (*xval*, *xunit*, *verbose=False*)
Given a wavelength/frequency/velocity, return the value in the SpectroscopicAxis's units e.g.: xarr.units = 'km/s' xarr.refX = 5.0 xarr.refX_units = GHz xarr.x_to_coord(5.1,'GHz') == 6000 # km/s

x_to_pix (*xval*, *xval_units=None*)
Given an X coordinate in SpectroscopicAxis' units, return the corresponding pixel number

class pyspeckit.spectrum.units.**SpectroscopicAxes**
Bases: pyspeckit.spectrum.units.SpectroscopicAxis

Counterpart to Spectra: takes a list of SpectroscopicAxis's and concatenates them while checking for consistency and maintaining header parameters

3.3.5 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)
```

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*, *multisingle='single'*, *override=False*, *key=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?

3.4 Readers

3.4.1 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.

API

Routines for reading in ASCII format spectra. If atpy 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'*, *atpytype='ascii'*, ***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 atpy or a 'simple' reader. If you have an IPAC, CDS, or formally formatted table, you'll want to use atpy.

If you have a simply formatted file of the form, e.g. # name name # unit unit data data data data

kwargs are passed to atpy.Table

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.

3.4.2 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')
```

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 div>*, *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

3.4.3 hdf5

(work in progress)

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.

3.4.4 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.

An essentially undocumented API

GILDAS CLASS file reader

Read a CLASS file into an `pyspeckit.spectrum.ObsBlock`

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.**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*)
A hacked-together method to read a binary CLASS file. It is strongly dependent on the incomplete GILDAS CLASS file type Specification

3.4.5 GBTIDL FITS files

GBTIDL SDFITS sessions can be loaded as `pyspeckit.ObsBlock` objects using the GBTSession reader:

```
gbtsession = pyspeckit.readers.GBTSession('AGBTsession.fits')
```

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_blocks**(*blocks*, *verbose=False*, *average=True*, *fd0=1*, *fd1=2*)
Do a nodded on/off observation given a dict of observation blocks as produced by read_gbt_target

pyspeckit.spectrum.readers.gbt.**reduce_gbt_target**(*sdfitsfile*, *objectname*, *verbose=False*)
Wrapper - read an SDFITS file, get an object, reduce it (assuming nodded) and return it

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

3.5 Wrappers

These are wrappers to simplify some of the more complicated (and even some of the simpler) functions in PySpecKit

3.5.1 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*, *clobber=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

clobber: [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*, *units='km/*
do-
plot=True,
base-
line=True,
plotresid-
u-
als=False,
fig-
ure-
save-
name=Non
cro-
prange=No
save-
name=Non
***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

3.5.2 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)
```

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], vrangeunits='km/s', quiet=False, Tex=20, Tkin=15, column=1000000000000000.0, fortho=1.0, tau=None*)

pyspeckit.wrappers.fitnh3.**fitnh3tkin**(*input_dict*, *dobaseline=True*, *baselinekwargs={}*, *crop=False*, *guessline='twotwo'*, *tex=15*, *tkin=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*, *npeaks=1*, *guesses=None*, ***kwargs*)

Given a dictionary of filenames and lines, fit them together e.g. {'oneone':'G000.000+00.000_nh3_11.fits'}

pyspeckit.wrappers.fitnh3.**plot_nh3**(*spdict*, *spectra*, *fignum=1*, *show_components=False*, *residfignum=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()

3.5.3 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`

3.5.4 N2H+ extras

In place of the actual contents of N2H+ fitter, here are the modules used to make the wrapper

model.**SpectralModel**()

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.

static 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

3.6 Examples

Check out the flickr gallery.

Want your image or example included? E-mail us.

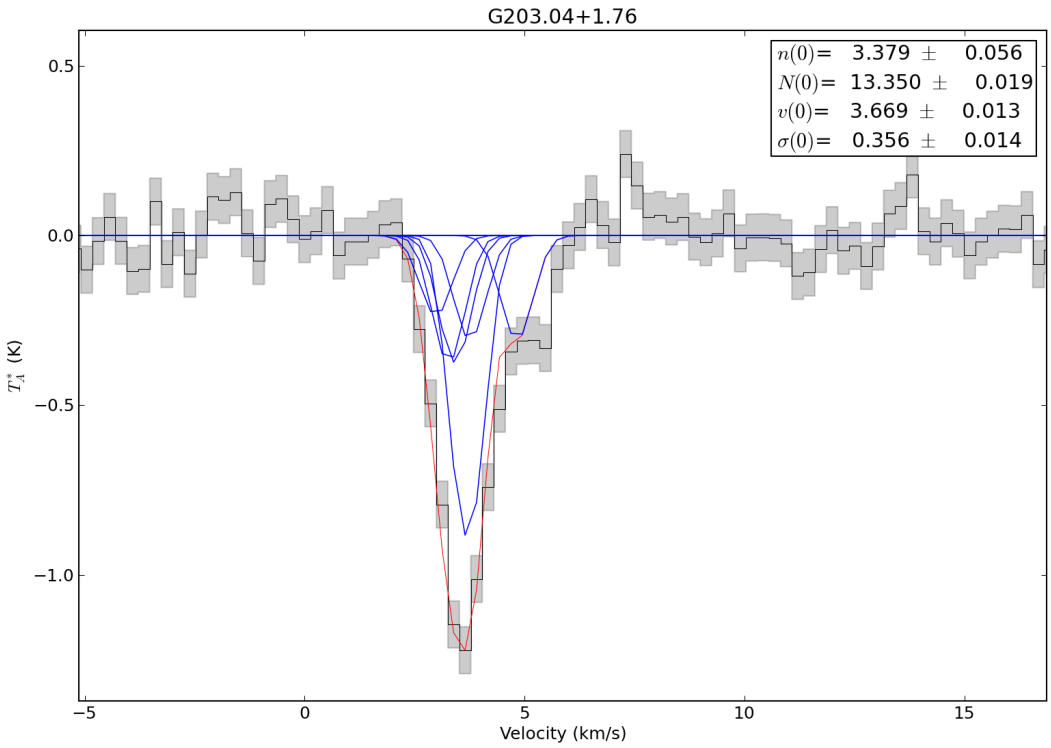
3.6.1 Radio Fitting: H_2CO RADEX example

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 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
texgrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_greenscale
taugrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_greenscale
texgrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_greenscale
taugrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_greenscale
hdr = pyfits.getheader('/Users/adam/work/h2co/radex/grid_greenscaled/1-1_2-2_T5to55_lvg_greenscaled_t

# 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
        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.f
taugrid1 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_sphere_tau1.f
texgrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_sphere_tex2.f
taugrid2 = pyfits.getdata('/Users/adam/work/h2co/radex/grid_aug2011_sphere/grid_aug2011_sphere_tau2.f
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],
        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,multisingle='multi')
sp.Registry.add_fitter('formaldehyde_radex_sphere',
        formaldehyde_radex_fitter_sphere,4,multisingle='multi')

sp.plotter()
sp.specfit(fittype='formaldehyde_radex',multifit=True,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')
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=True,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)
```



3.6.2 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',
```

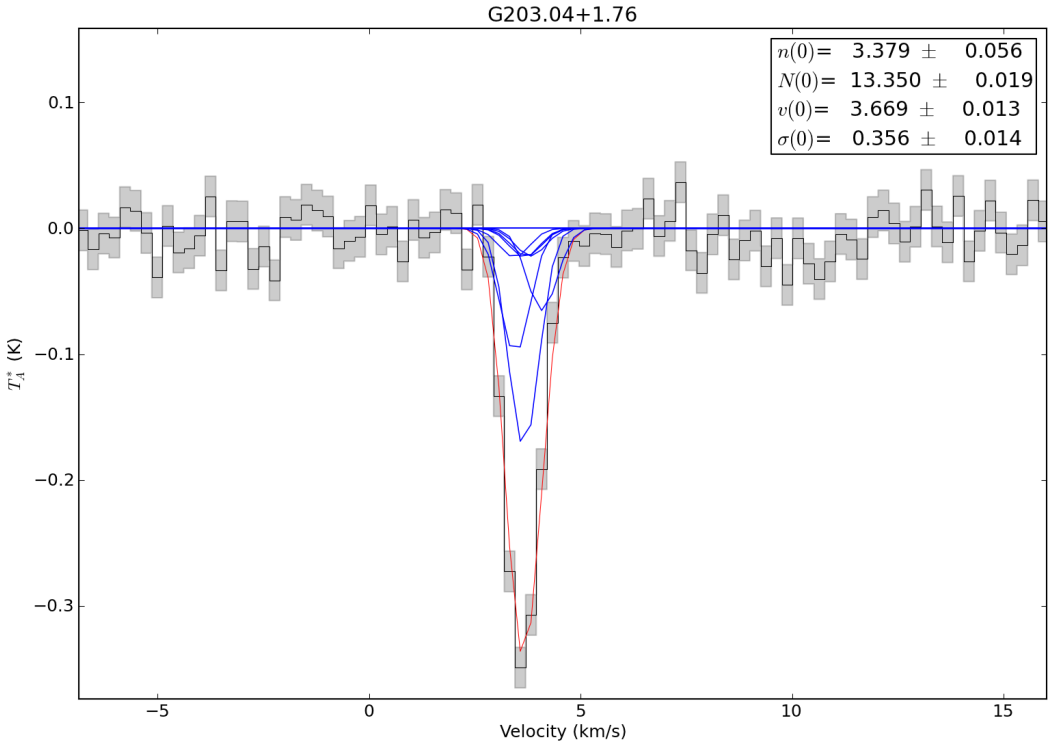


Figure 3.1: Both transitions are fit simultaneously using a RADEX model. The input (fitted) parameters are therefore density, column density, width, and velocity.

```
    '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,col
```

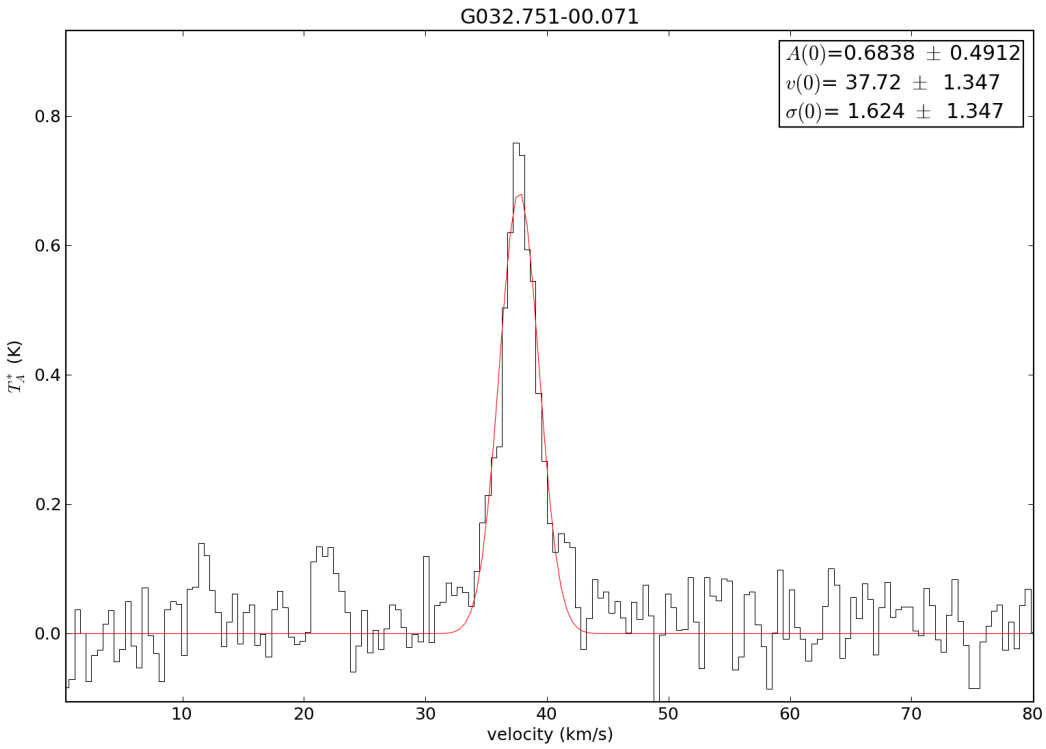


Figure 3.2: The 2-2 transition is used to guess the central velocity and width via gaussian fitting because its hyperfine lines are weaker

3.6.3 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
```

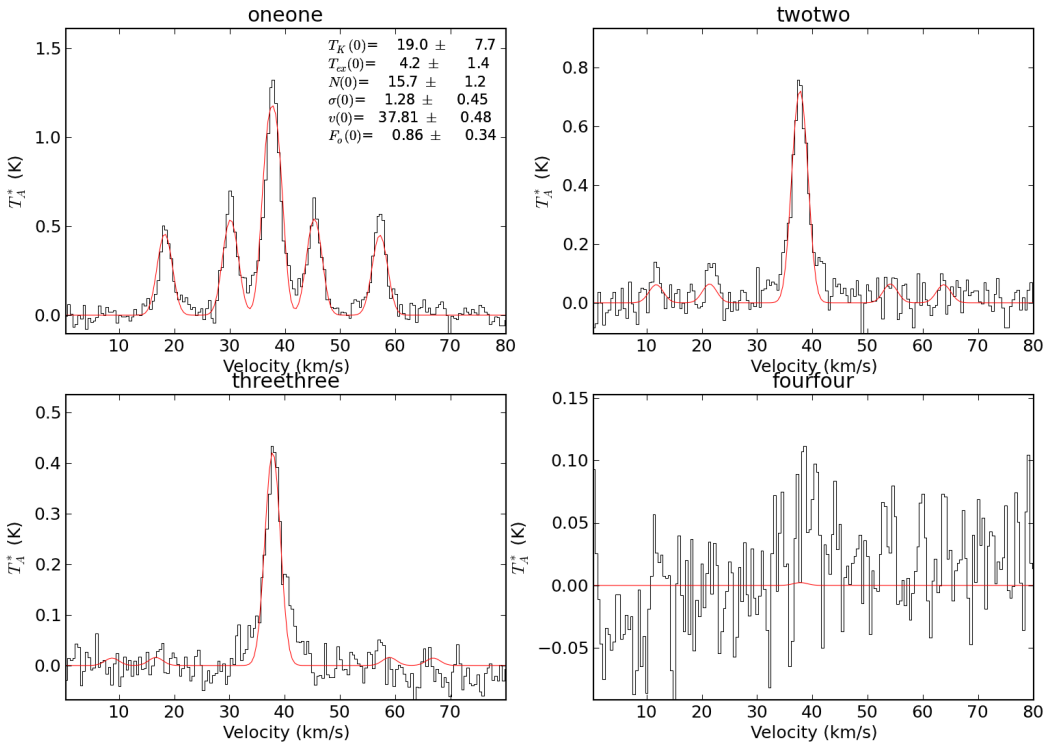


Figure 3.3: Then all 4 lines are simultaneously fit. Even upper limits on the 4-4 line can provide helpful contraints on the model

```
    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
try:
    import astropy.io.fits as pyfits
except ImportError:
    import pyfits
import numpy as np
import os
import agpy

# 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_rerun.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.units = "K"
cube22 = pyspeckit.Cube('hotclump_22.cube_r0.5_contsub.image.fits', maskmap=mask)
cube22.cube *= (13.6 * (300.0 /
        (pyspeckit.spectrum.models.ammonia.freq_dict['twotwo']/1e9))**2 *
        1./cube22.header.get('BMAJ')/3600. * 1./cube22.header.get('BMIN')/3600. )
cube22.units = "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.units = "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
errmap11 = (pyfits.getdata('hotclump_11.cube_r0.5_rerun.image.moment_linefree.fits').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.
        )
errmap11[errmap11 != errmap11] = agpy.smooth(errmap11, interp_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.units = "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)
    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
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)
    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=True, 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
    fitcubefile.header.update('PLANE1','TKIN')
    fitcubefile.header.update('PLANE2','TEX')
    fitcubefile.header.update('PLANE3','COLUMN')
    fitcubefile.header.update('PLANE4','SIGMA')
    fitcubefile.header.update('PLANE5','VELOCITY')
    fitcubefile.header.update('PLANE6','FORTHO')
    fitcubefile.header.update('PLANE7','eTKIN')
    fitcubefile.header.update('PLANE8','eTEX')
    fitcubefile.header.update('PLANE9','eCOLUMN')
    fitcubefile.header.update('PLANE10','eSIGMA')
    fitcubefile.header.update('PLANE11','eVELOCITY')
    fitcubefile.header.update('PLANE12','eFORTHO')
    fitcubefile.header.update('CDELT3',1)
    fitcubefile.header.update('CTYPE3','FITPAR')
    fitcubefile.header.update('CRVAL3',0)
    fitcubefile.header.update('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_units='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.
```

3.6.4 Radio Fitting: N_2H+ example

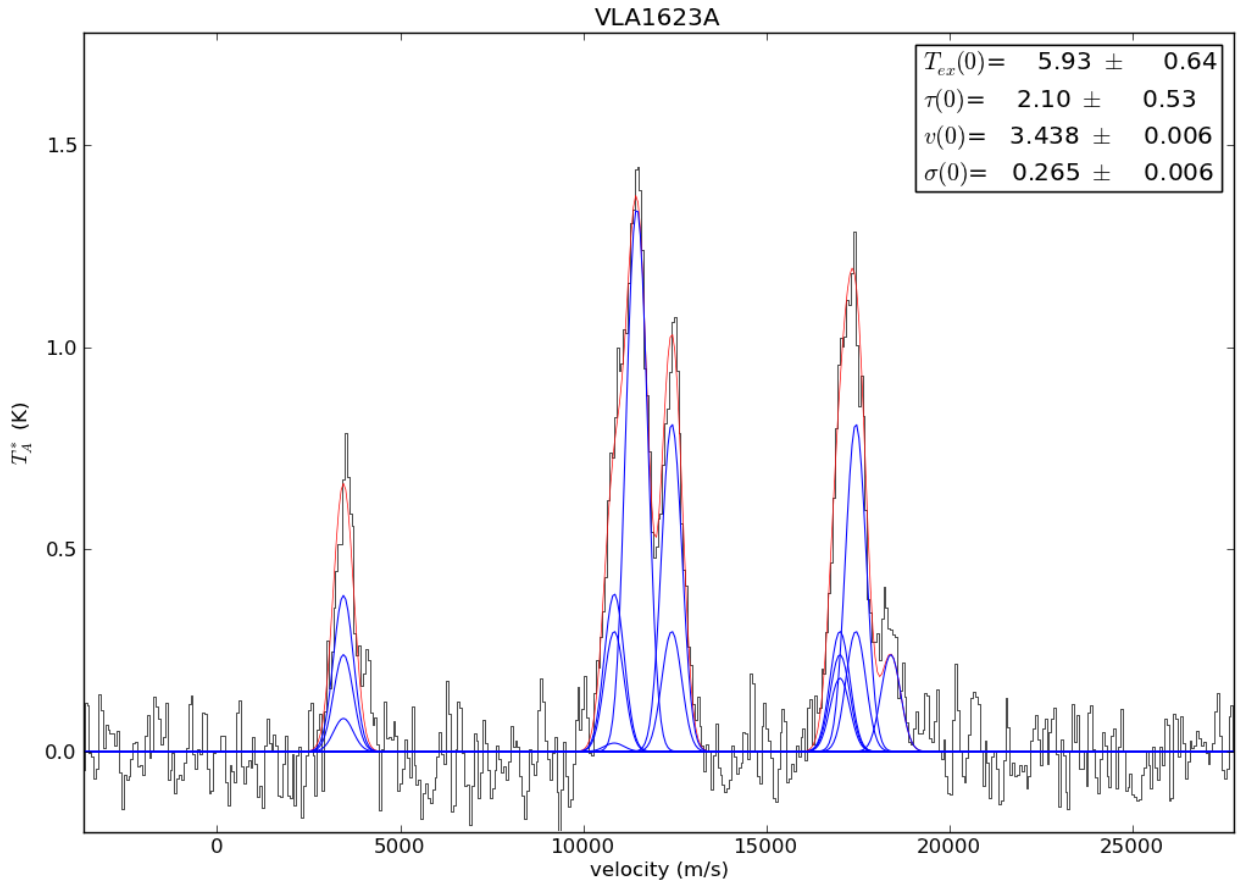
Example hyperfine line fitting for the N_2H+ 1-0 line.

```
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, multisingle='multi')

# Run the fitter
sp.specfit(fittype='n2hp_vtau',multifit=True,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=True,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')
```

3.6.5 Radio Fitting: N_2H+ cube example

Example hyperfine line fitting of a data cube for the N_2H+ 1-0 line.

```
import pyspeckit

# Load the spectral cube
spc = pyspeckit.Cube('n2hp_cube.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)
spc.Registry.add_fitter('n2hp_vtau',pyspeckit.models.n2hp.n2hp_vtau_fitter,4,multisingle='multi')

# Run the fitter
spc.fiteach(fittype='n2hp_vtau', multifit=True,
        guesses=[5,0.5,3,1], # Tex=5K, tau=0.5, v_center=12, width=1 km/s
        )
# 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.SpectralCu
#
# 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.
```


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)

```
# However, pyspeckit@gmail.com will support them!  They are being used
# in current and pending publications

# Show an integrated image
spc.mapplot()

# plot one of the fitted spectra
spc.plot_spectrum(0,0,plot_fit=True)

# 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.show()
```

3.6.6 Radio Fitting: HCN example with freely varying hyperfine amplitudes

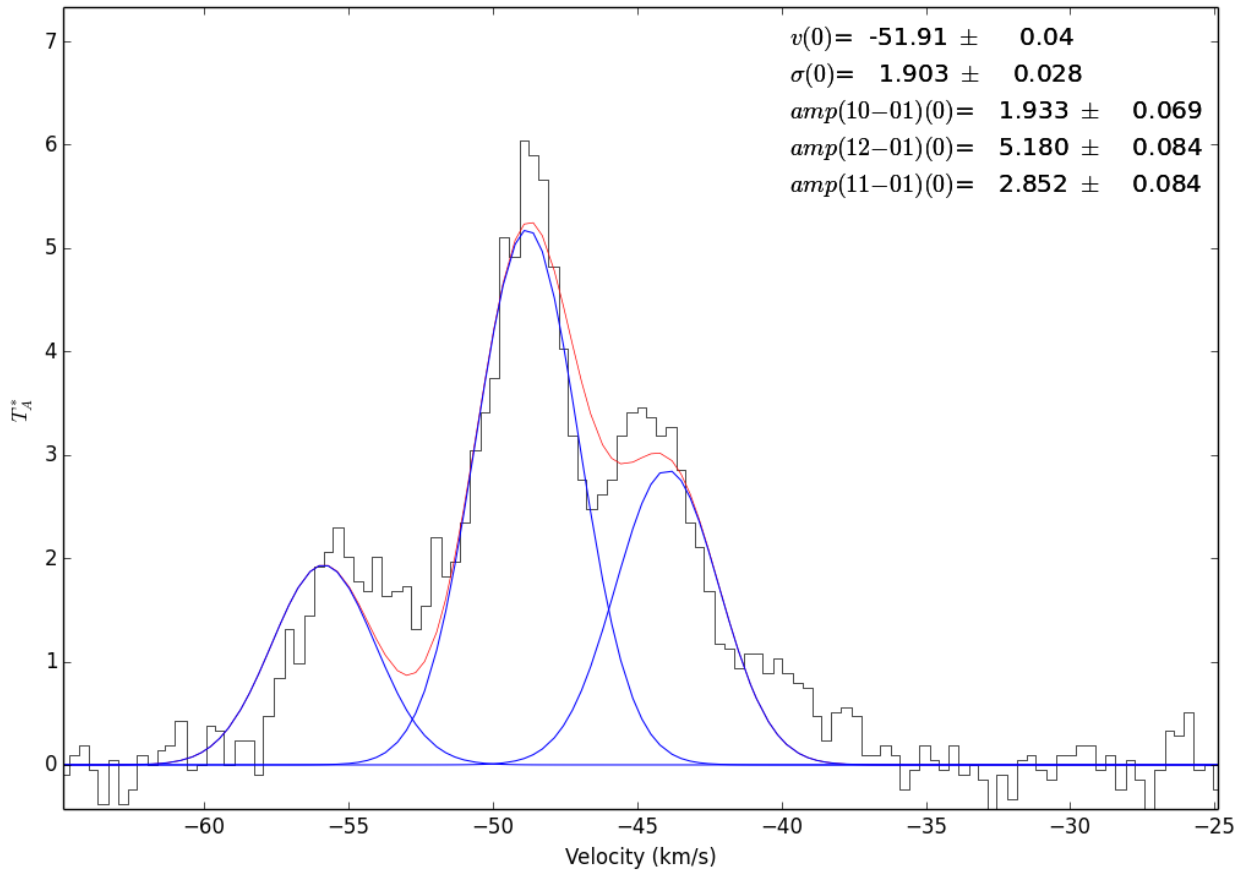
Example hyperfine line fitting for the HCN 1-0 line.

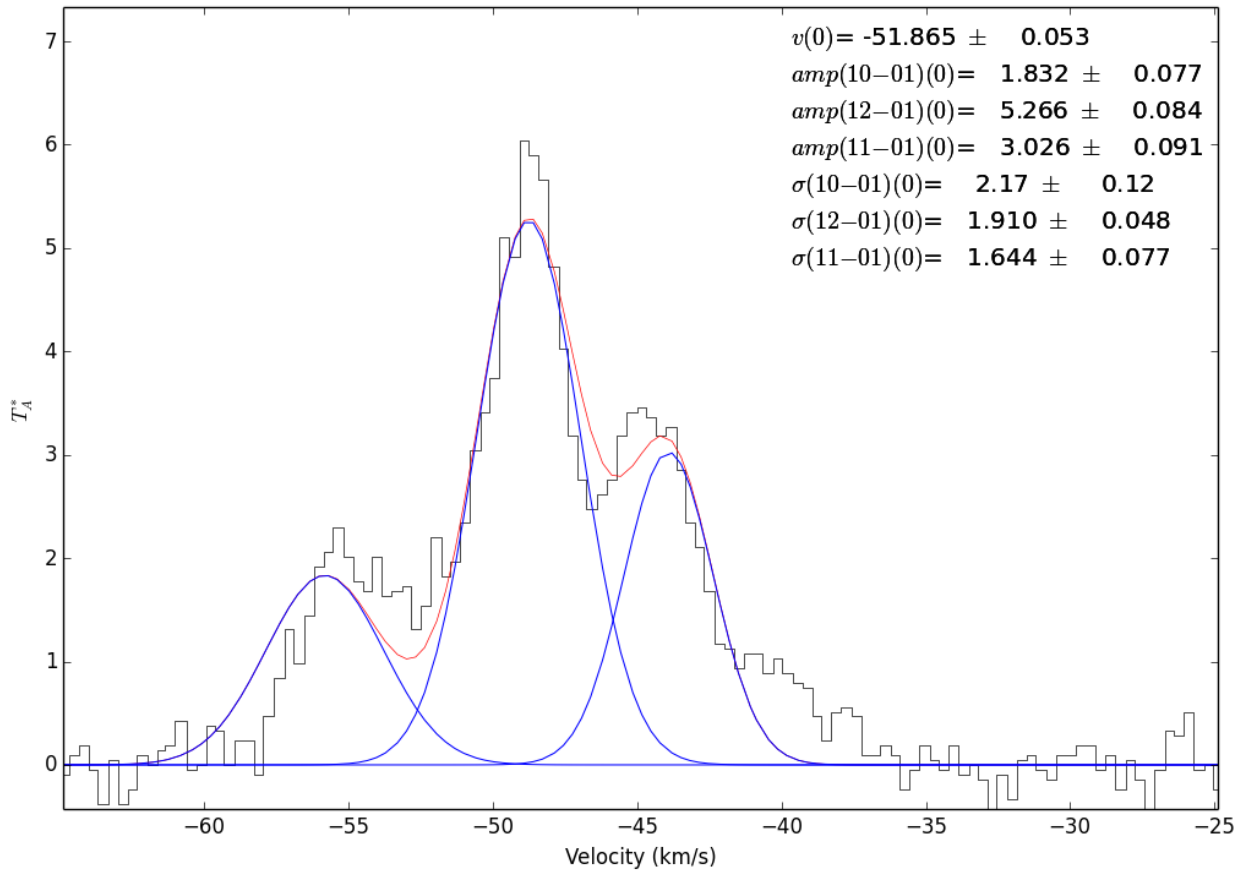
```
import pyspeckit

# Load the spectrum & properly identify the units
# The data is from http://adsabs.harvard.edu/abs/1999A%26A...348..600P
sp = pyspeckit.Spectrum('02232+6138.txt')
sp.xarr.units='km/s'
sp.xarr.refX = 88.63184666e9
sp.xarr.xtype='velocity'
sp.units='$T_A^*$'

# 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, multisingle='multi')

# Plot the results
sp.plotter()

# Run the fitter and show the individual fit components
sp.specfit(fittype='hcn_varyhf',
           multifit=True,
           guesses=[-53,1,0.2,0.6,0.3],
           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('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, multisingle='multi')

# Run the fitter
sp.specfit(fittype='hcn_varyhf_width',multifit=True,
           guesses=[-53,0.2,0.6,0.3,1,1,1],
           show_hyperfine_components=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)

# 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')
```

3.6.7 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.frequency_to_velocity()
# Default conversion is to m/s, but we traditionally work in km/s
spec.xarr.convert_to_unit('km/s')
# plot it up!
spec.plotter()
# Subtract a baseline (the data is only 'mostly' reduced)
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")
```


$v(0)$= -51.91 ± 0.04
$\sigma(0)$= 1.903 ± 0.028
$amp(10-01)(0)$= 1.933 ± 0.069
$amp(12-01)(0)$= 5.180 ± 0.084
$amp(11-01)(0)$= 2.852 ± 0.084
T_A^*
Velocity (km/s)


$v(0)$= -51.865 ± 0.053
$amp(10-01)(0)$= 1.832 ± 0.077
$amp(12-01)(0)$= 5.266 ± 0.084
$amp(11-01)(0)$= 3.026 ± 0.091
$\sigma(10-01)(0)$= 2.17 ± 0.12
$\sigma(12-01)(0)$= 1.910 ± 0.048
$\sigma(11-01)(0)$= 1.644 ± 0.077
T_A^*
Velocity (km/s)

```
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")
```

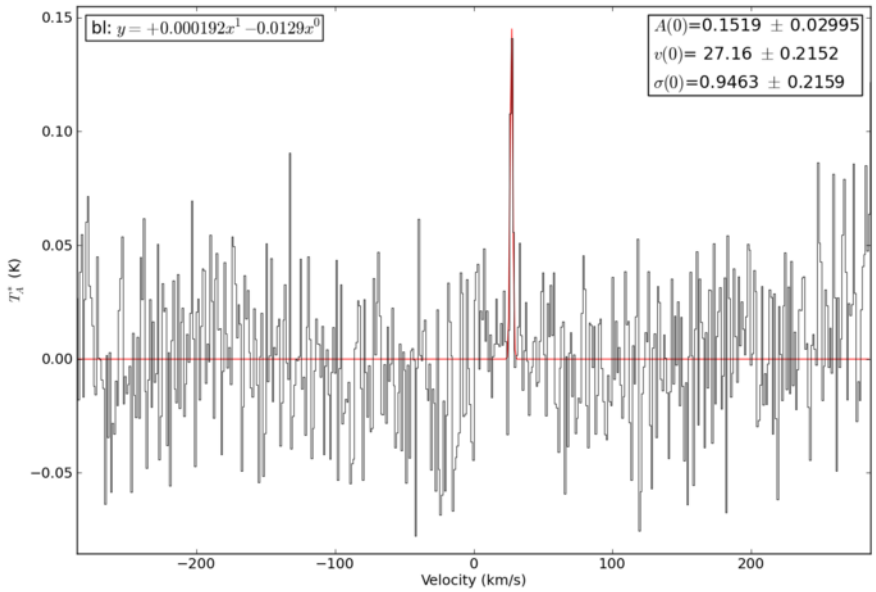


Figure 3.4: Sample HCO+ spectrum fitted with a gaussian

3.6.8 Optical fitting: The Hα-[NII] complex of a type-I Seyfert galaxy

```
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].
```

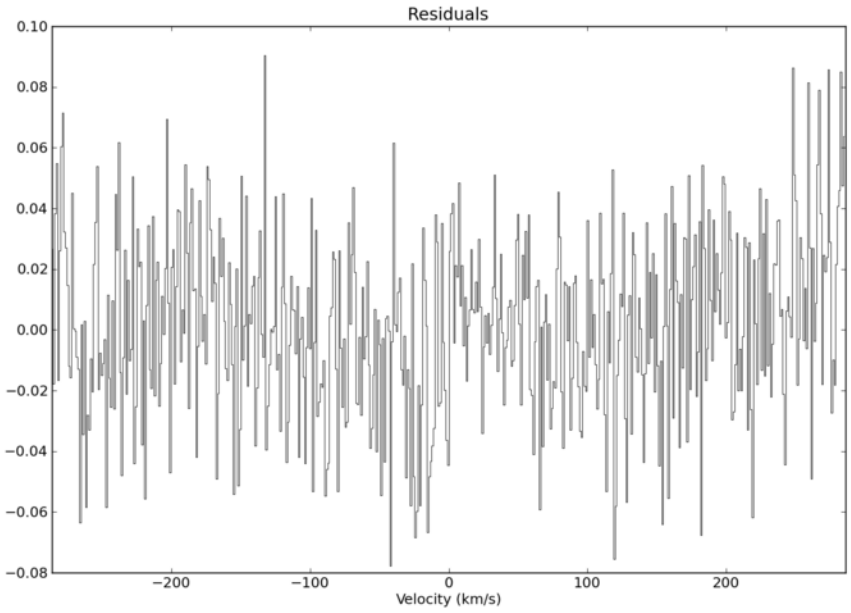


Figure 3.5: Residuals of the gaussian fit from the previous figure

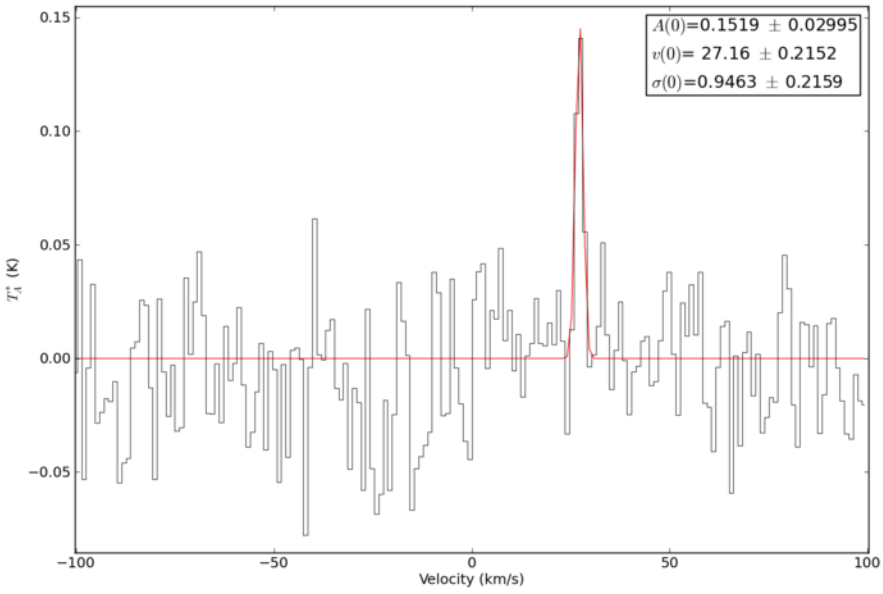
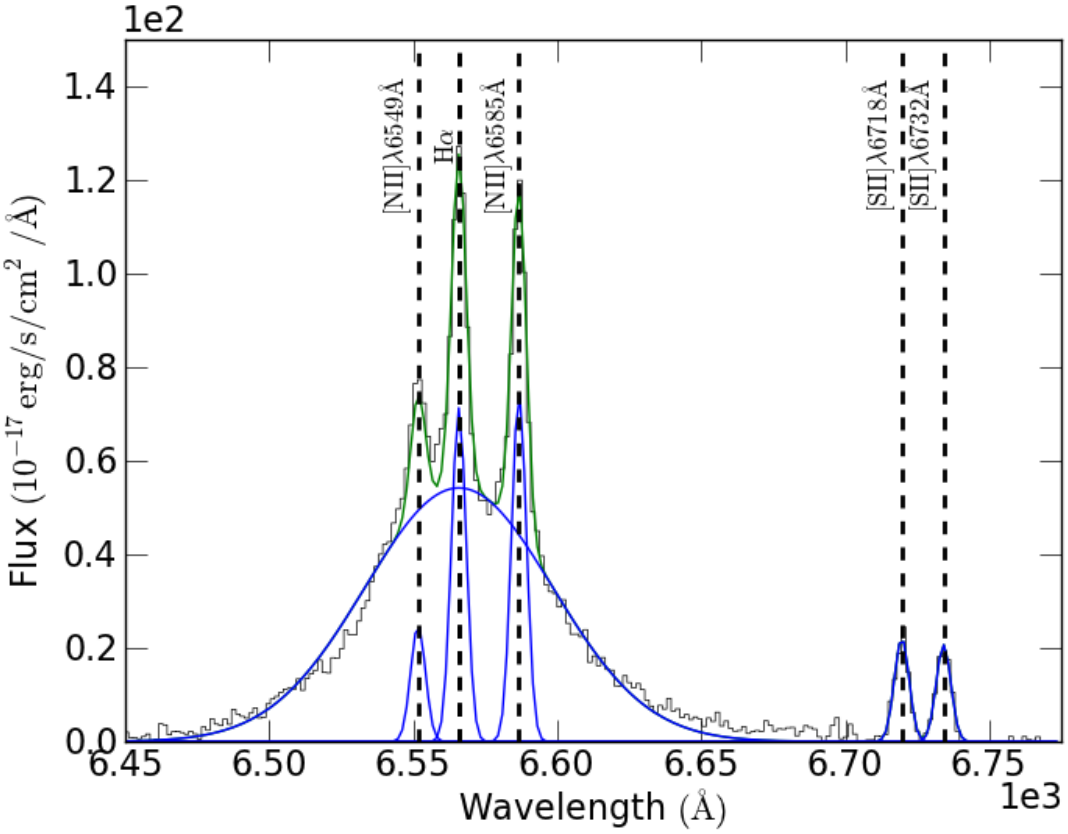


Figure 3.6: A zoomed-in, cropped version of the spectrum. With the 'crop' command, the excess data is discarded.

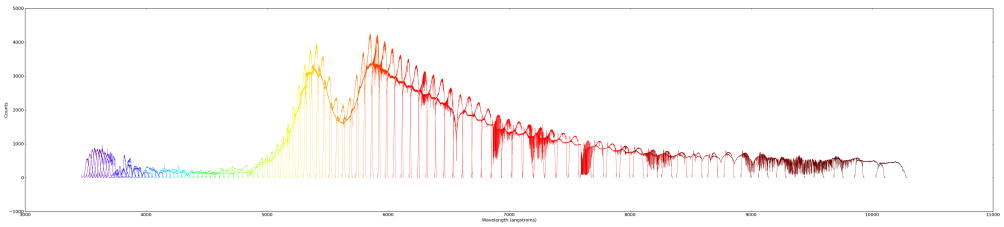
```
guesses = [50, NIIa, 5, 100, Halpha, 5, 50, Halpha, 50, 50, NIIb, 5, 20, SIIa, 5, 20, SIIb, 5]
tied = ['', '', 'p[17]', '', '', 'p[17]', '', 'p[4]', '', '3 * p[0]', '', 'p[17]', '', '', 'p[17]', '

# 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'.
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
    spec.plotter.axis.plot([x]*2, [spec.plotter.ymin, spec.plotter.ymax], ls = '--', color = 'k')
    spec.plotter.axis.annotate(spec.speclines.optical.lines[line][-1], (x, y), rotation = 90, ha = 'r

# 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.mea
        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")
```

3.6.9 Optical Plotting - Echelle spectrum of Vega (in color!)

```
import pyspeckit
from pylab import *
import wav2rgb

speclist = pyspeckit.wrappers.load_IRAF_multispec('evega.0039.rs.ec.dispcor.fits')
```


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)

```
for spec in speclist:
    spec.units="Counts"

SP = pyspeckit.Spectra(speclist)
SPa = pyspeckit.Spectra(speclist,xunits='angstroms',quiet=False)

SP.plotter(figure=figure(1))
SPa.plotter(figure=figure(2))

figure(3)
clf()
figure(4)
clf()

#clr = [list(clr) for clr in matplotlib.cm.brg(linspace(0,1,len(speclist)))]
clr = [wav2rgb.wav2RGB(c) + [1.0] for c in linspace(380,780,len(speclist))][::-1]
for ii,(color,spec) in enumerate(zip(clr,speclist)):
    spec.plotter(figure=figure(3), clear=False, reset=False, color=color, refresh=False)

    fig4=figure(4)
    fig4.subplots_adjust(hspace=0.35,top=0.97,bottom=0.03)
    spec.plotter(axis=subplot(10,1,ii%10+1), clear=False, reset=False, color=color, refresh=False)
    spec.plotter.axis.yaxis.set_major_locator( matplotlib.ticker.MaxNLocator(4) )

    if ii % 10 == 9:
        spec.plotter.refresh()
        spec.plotter.savefig('vega_subplots_%03i.png' % (ii/10+1))
        clf()

spec.plotter.refresh()
```

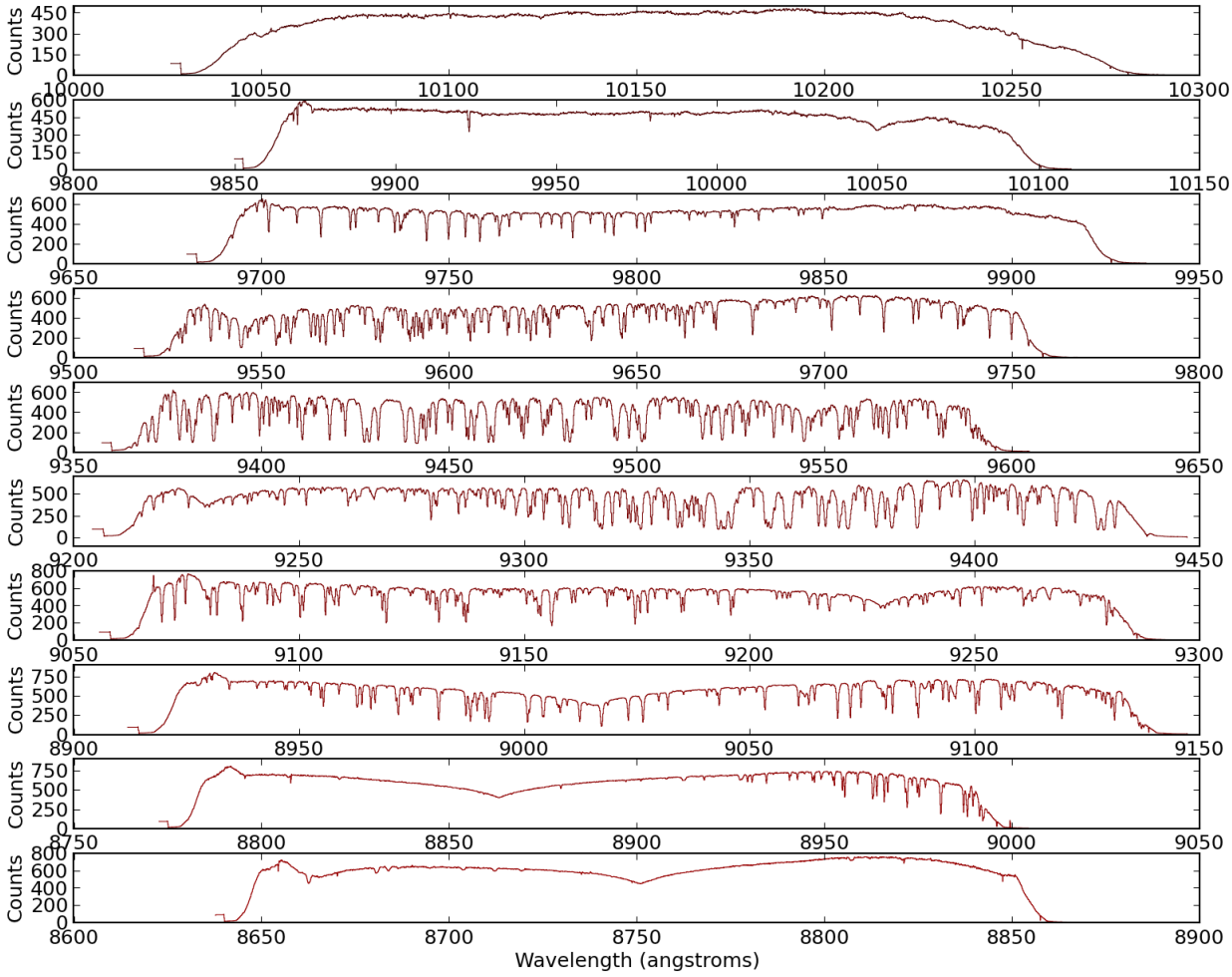


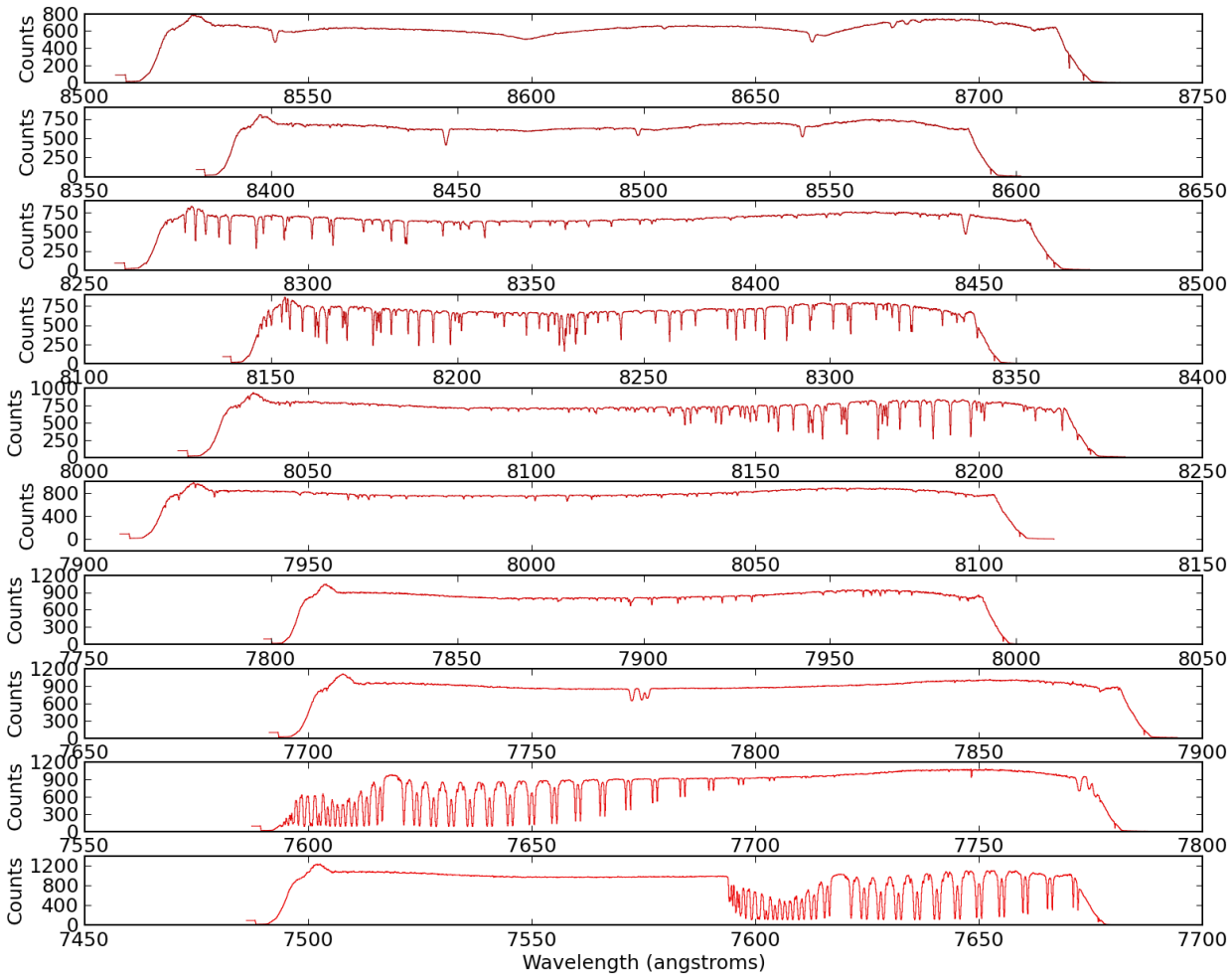
3.6.10 A guide to interactive fitting

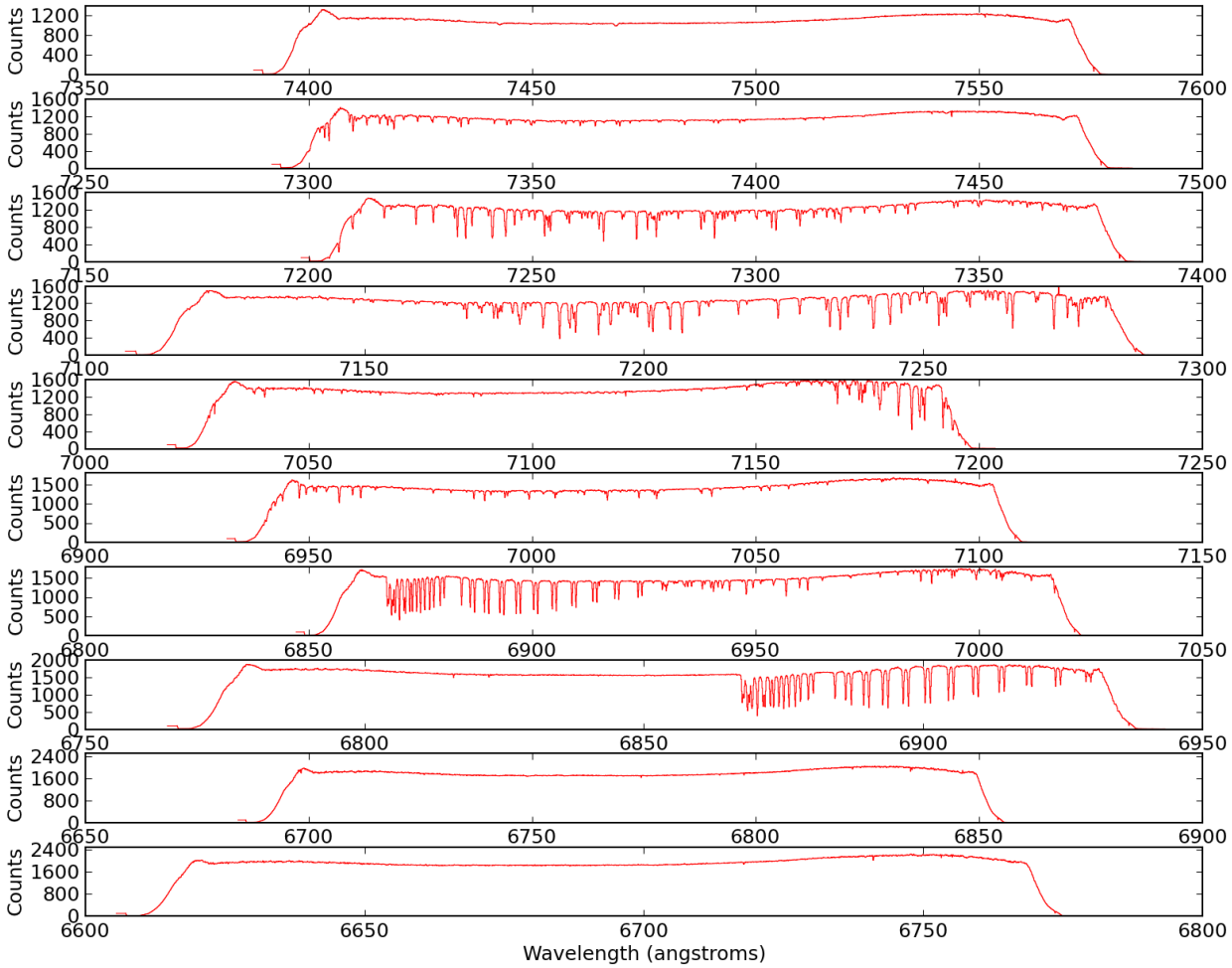
A step-by-step example of how to use the interactive fitter.

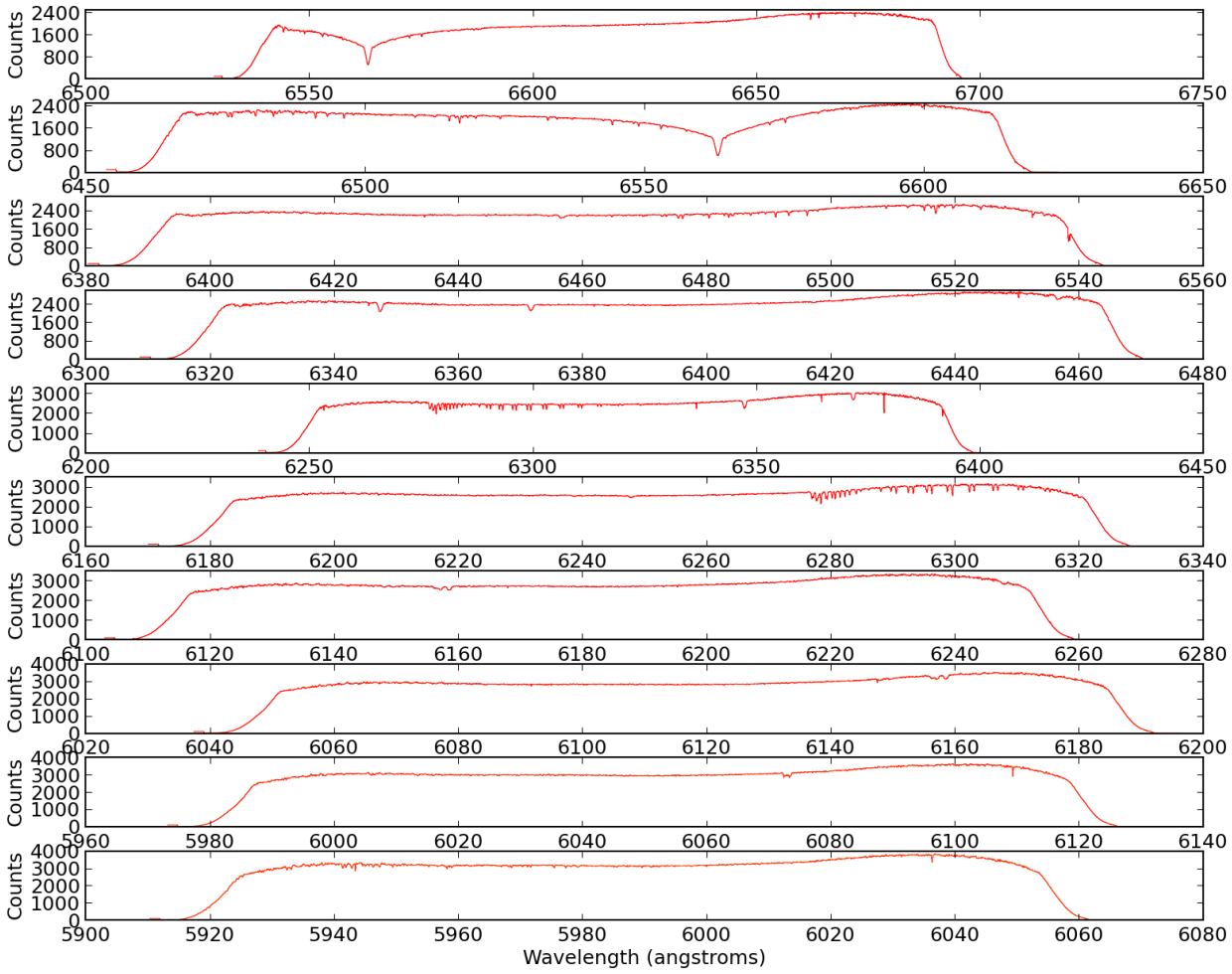
In short, we will do the following:

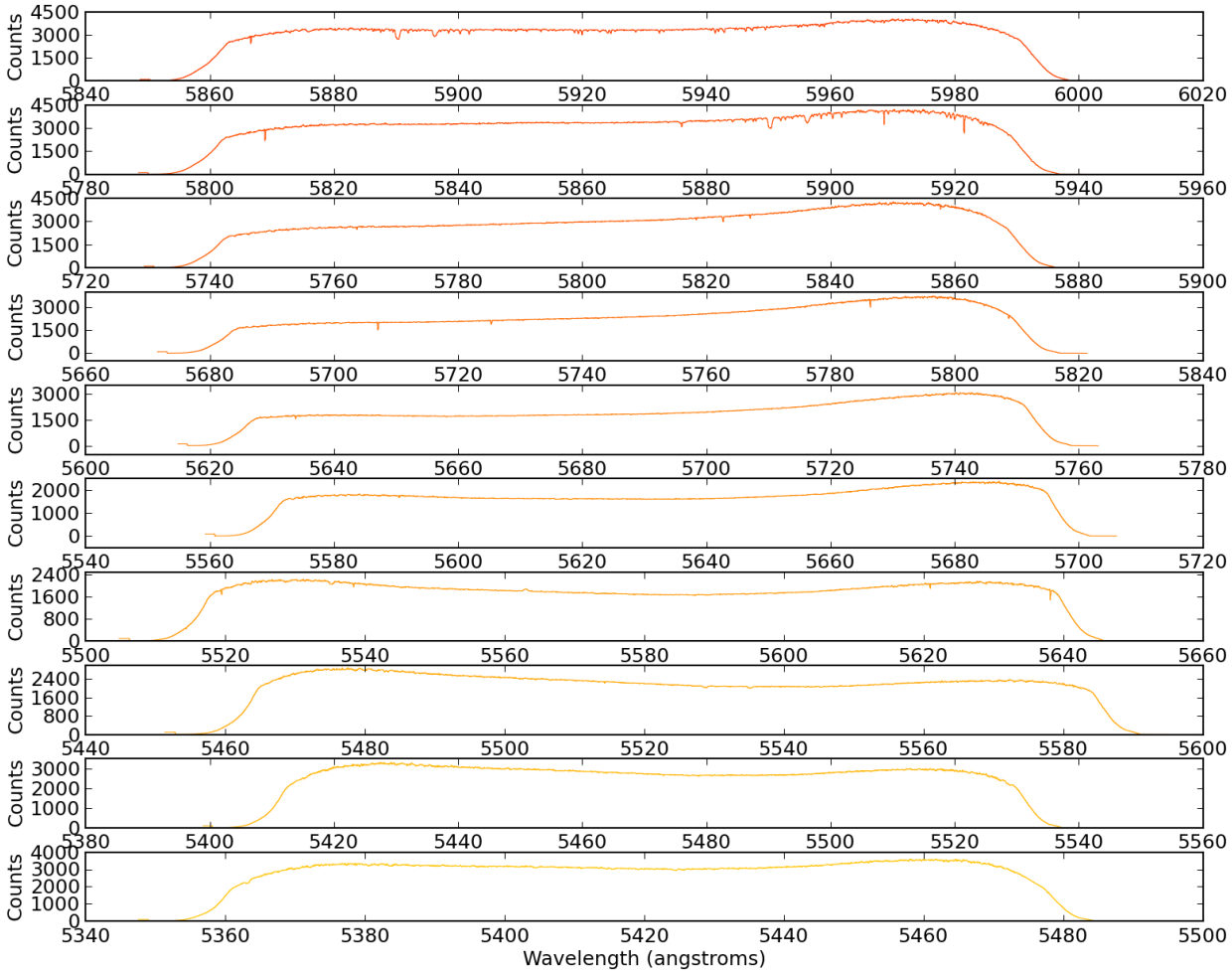
```
# 1. Load the spectrum
sp = pyspeckit.Spectrum('hr2421.fit')

# 2. Plot a particular spectral line
sp.plotter(xmin=4700,xmax=5000)

# 3. Need to fit the continuum first
sp.baseline(interactive=True, subtract=False)

# 4... (much work takes place interactively at this stage)
```

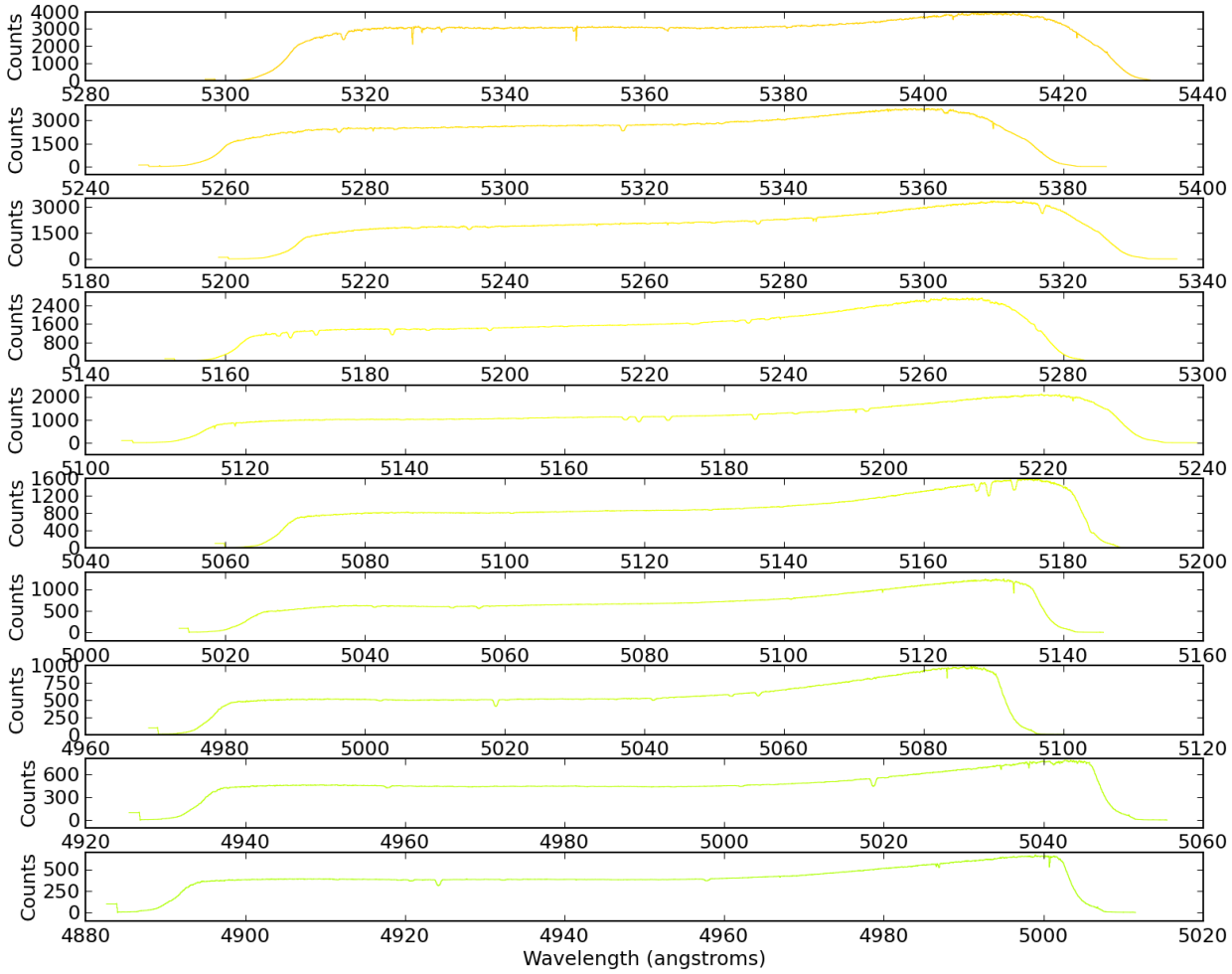

Counts
Wavelength (angstroms)

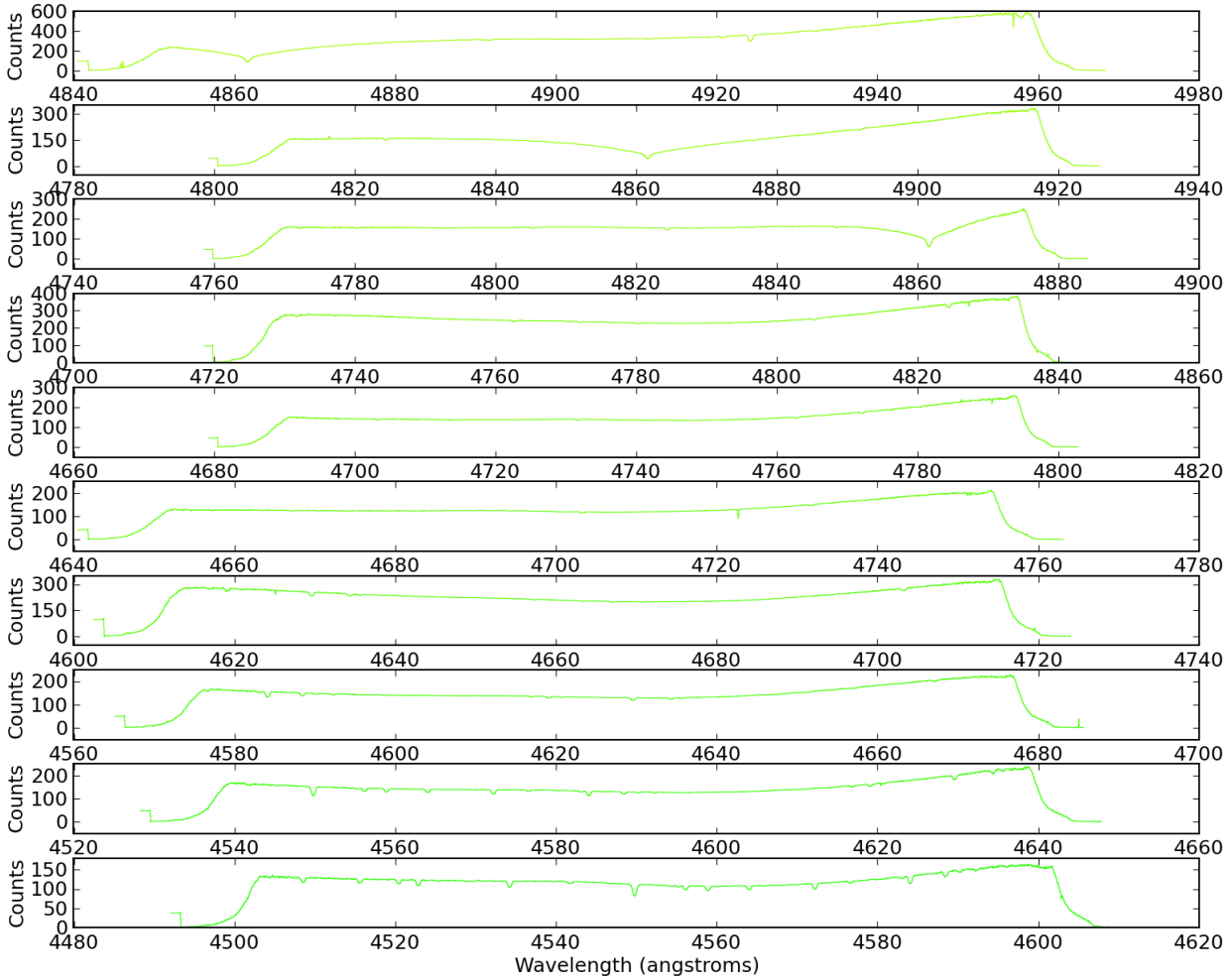

Counts
Wavelength (angstroms)

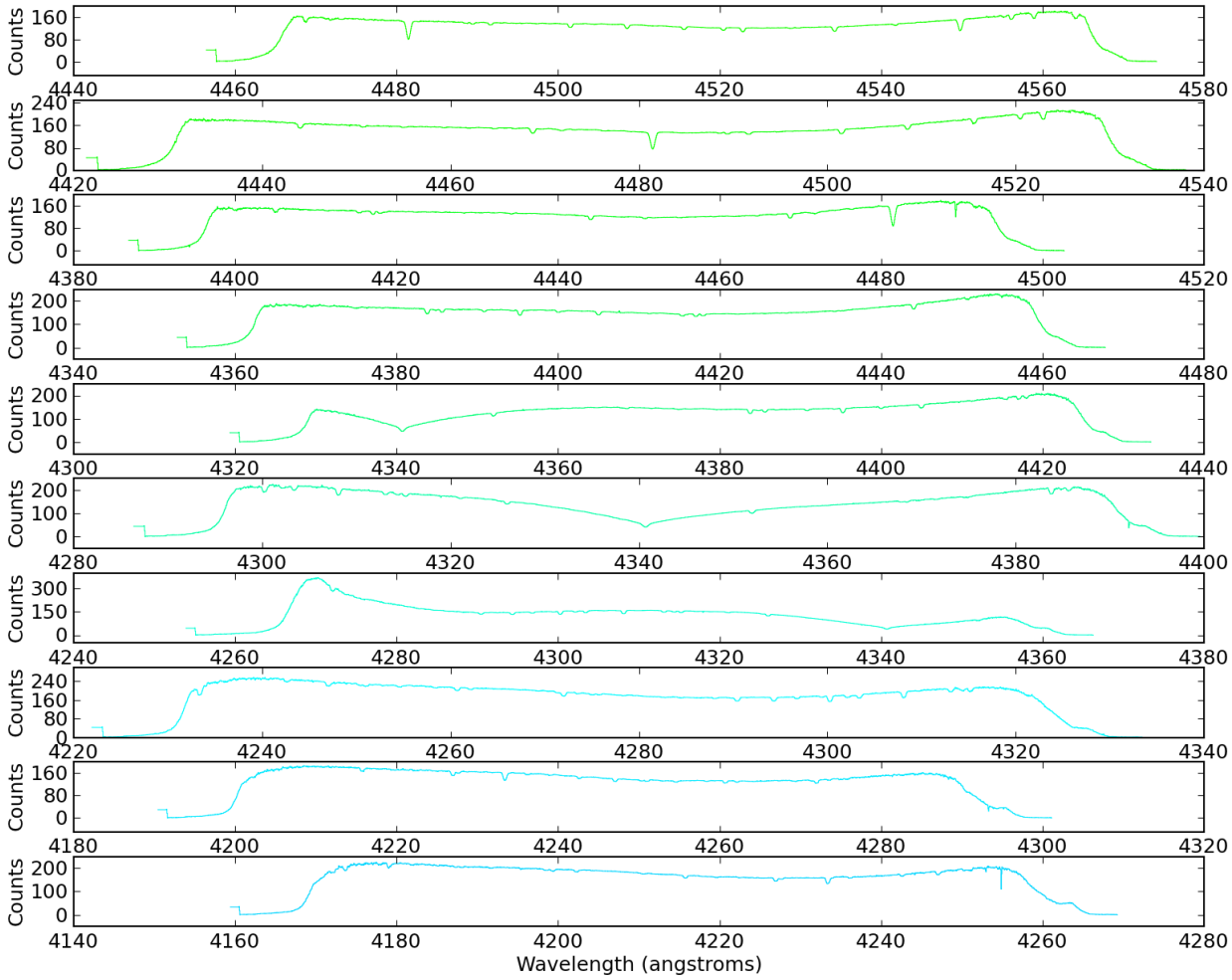

Counts
Wavelength (angstroms)

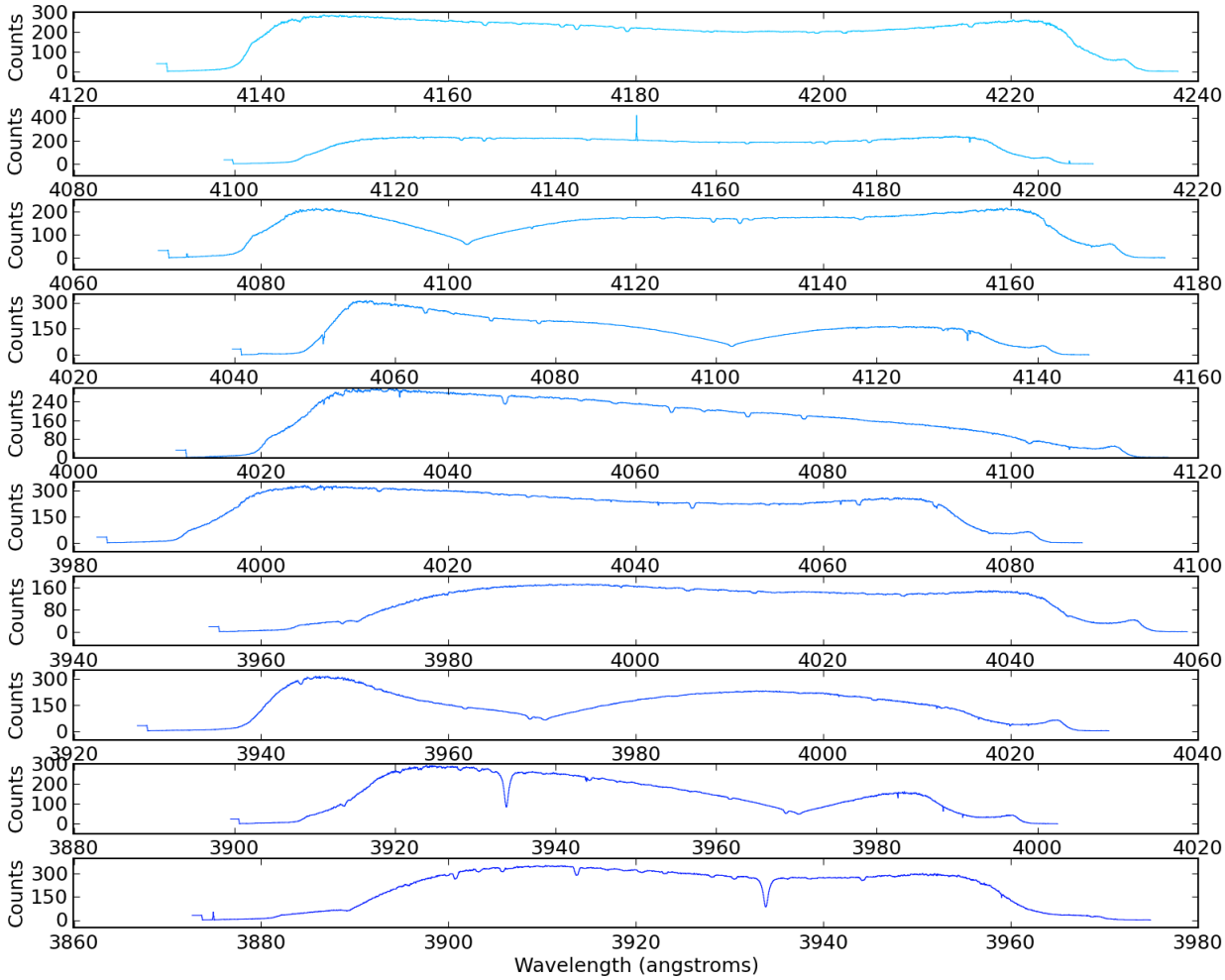

Counts
Wavelength (angstroms)

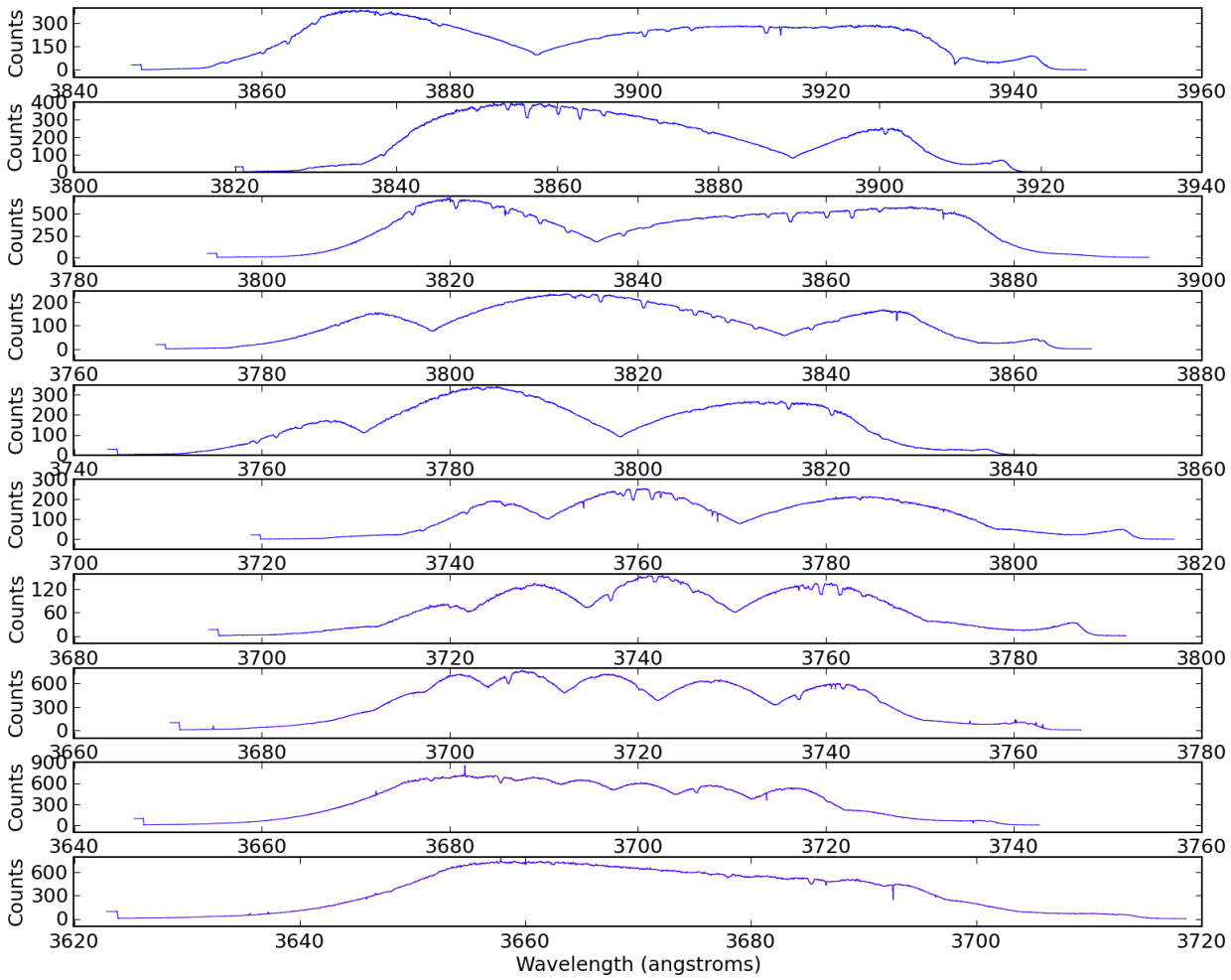

Counts
Wavelength (angstroms)


Counts
Wavelength (angstroms)


Counts
Wavelength (angstroms)

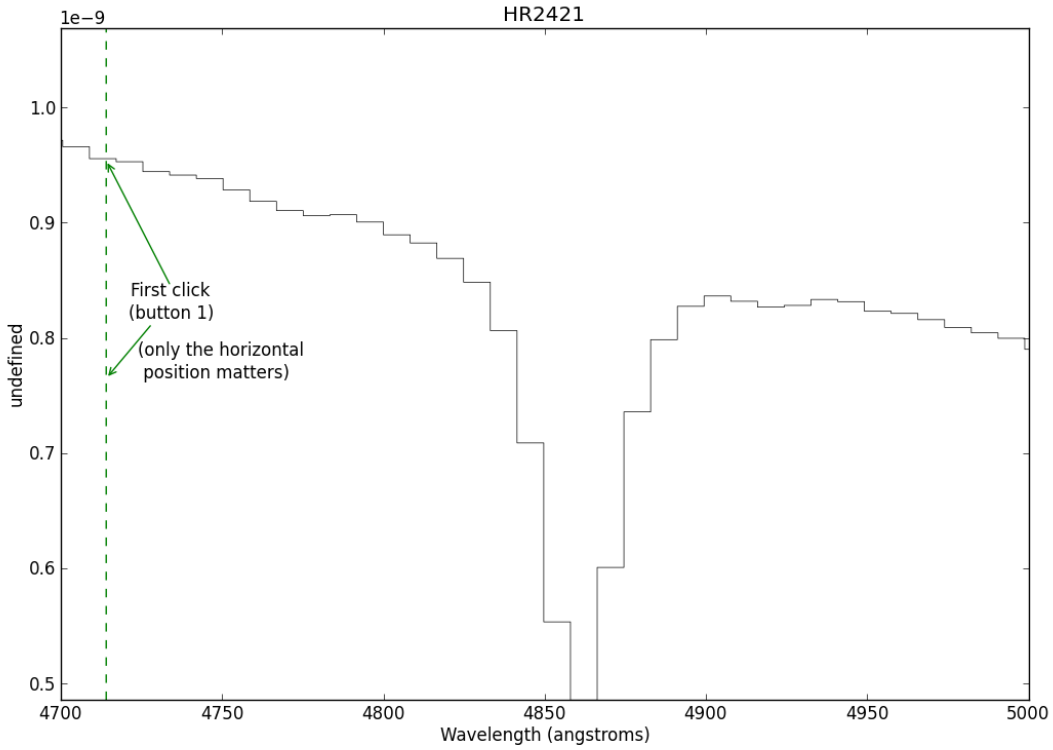

Counts
Wavelength (angstroms)


Counts
Wavelength (angstroms)


Counts
300
150
0
3840
3860
3880
3900
3920
3940
3960
Counts
400
300
200
100
0
3800
3820
3840
3860
3880
3900
3920
3940
Counts
500
250
0
3780
3800
3820
3840
3860
3880
3900
Counts
200
100
0
3760
3780
3800
3820
3840
3860
3880
Counts
300
200
100
0
3740
3760
3780
3800
3820
3840
3860
Counts
300
200
100
0
3700
3720
3740
3760
3780
3800
3820
Counts
120
60
0
3680
3700
3720
3740
3760
3780
3800
Counts
600
300
0
3660
3680
3700
3720
3740
3760
3780
Counts
900
600
300
0
3640
3660
3680
3700
3720
3740
3760
Counts
600
300
0
3620
3640
3660
3680
3700
3720
Wavelength (angstroms)

```
# 5. Start up an interactive line-fitting session
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`



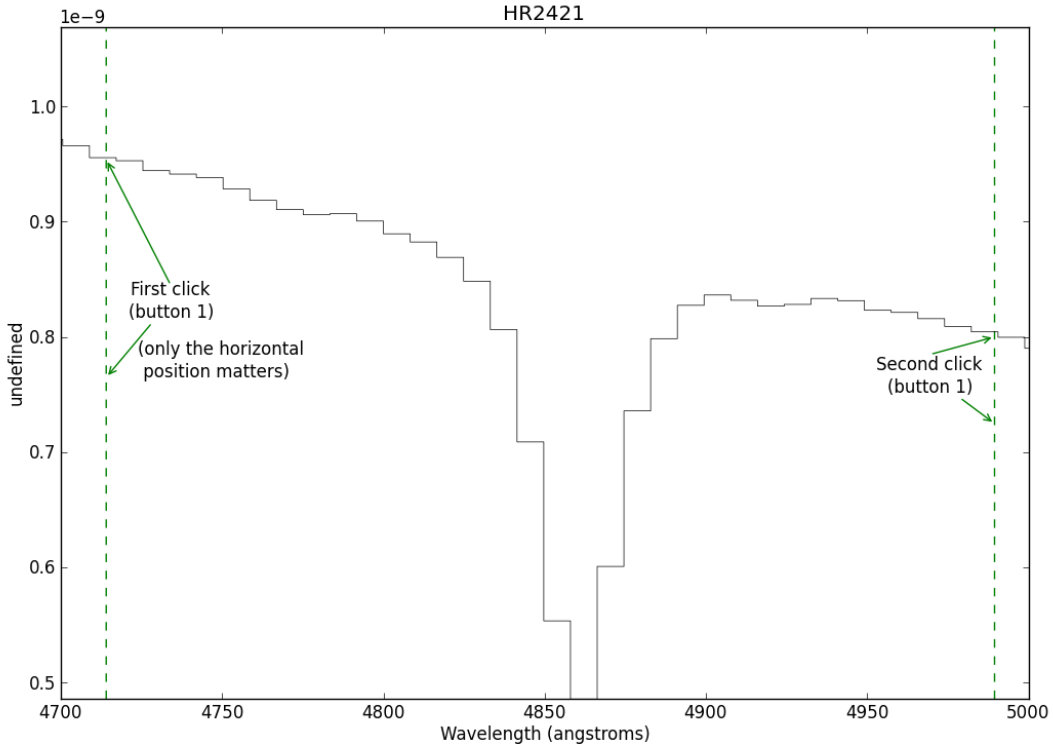
This is where you start the line-fitter:

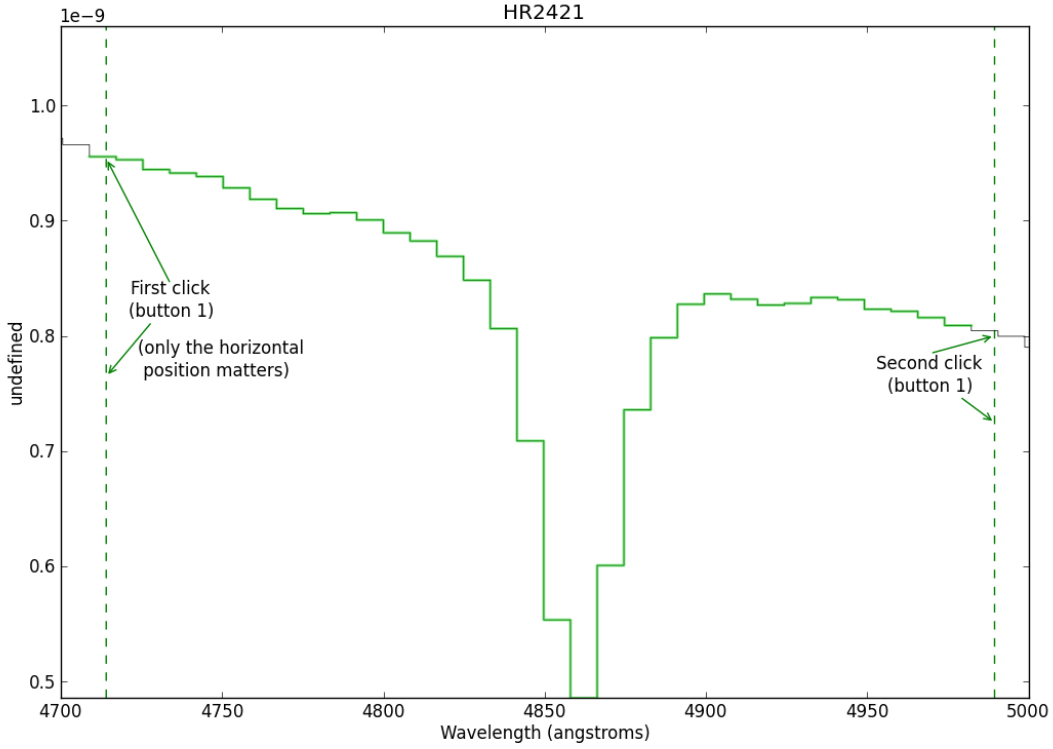
```
# Start up an interactive line-fitting session
sp.specfit(interactive=True)
```

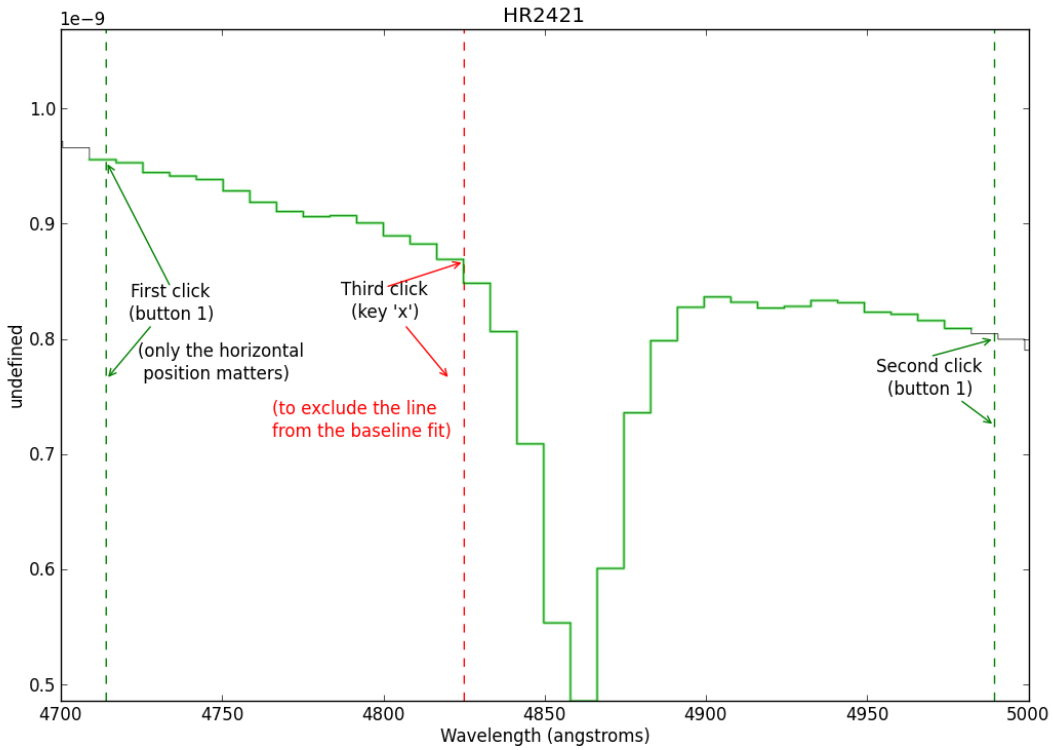
3.6.11 Complicated H-alpha Line Fitting

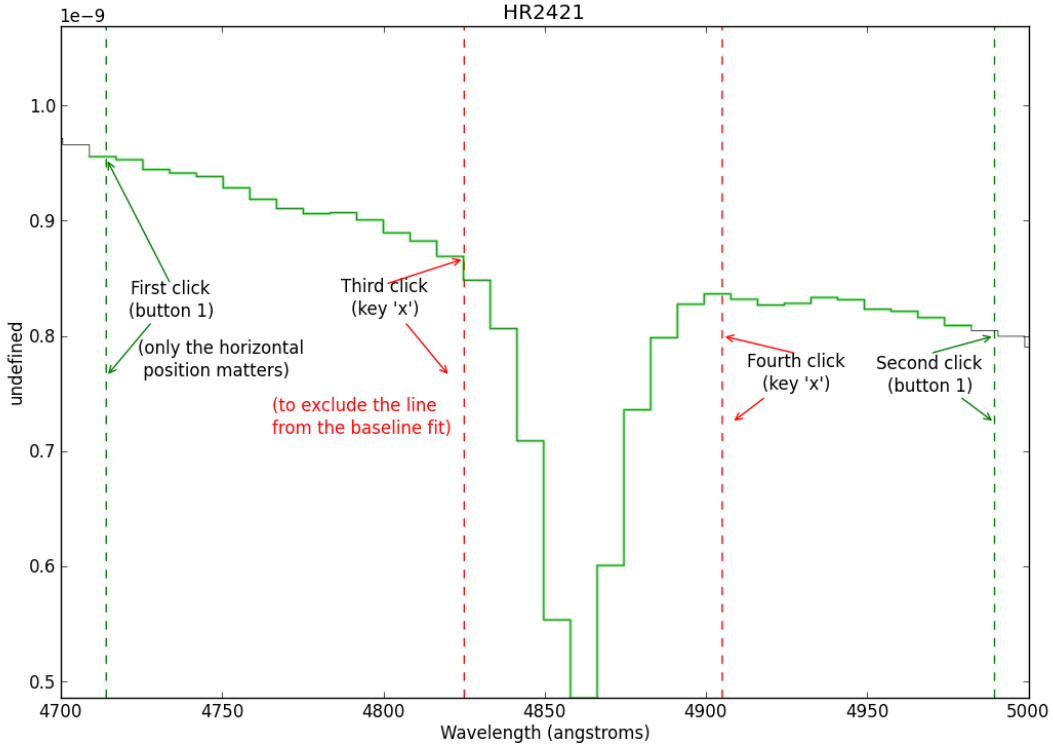
Source code for this example

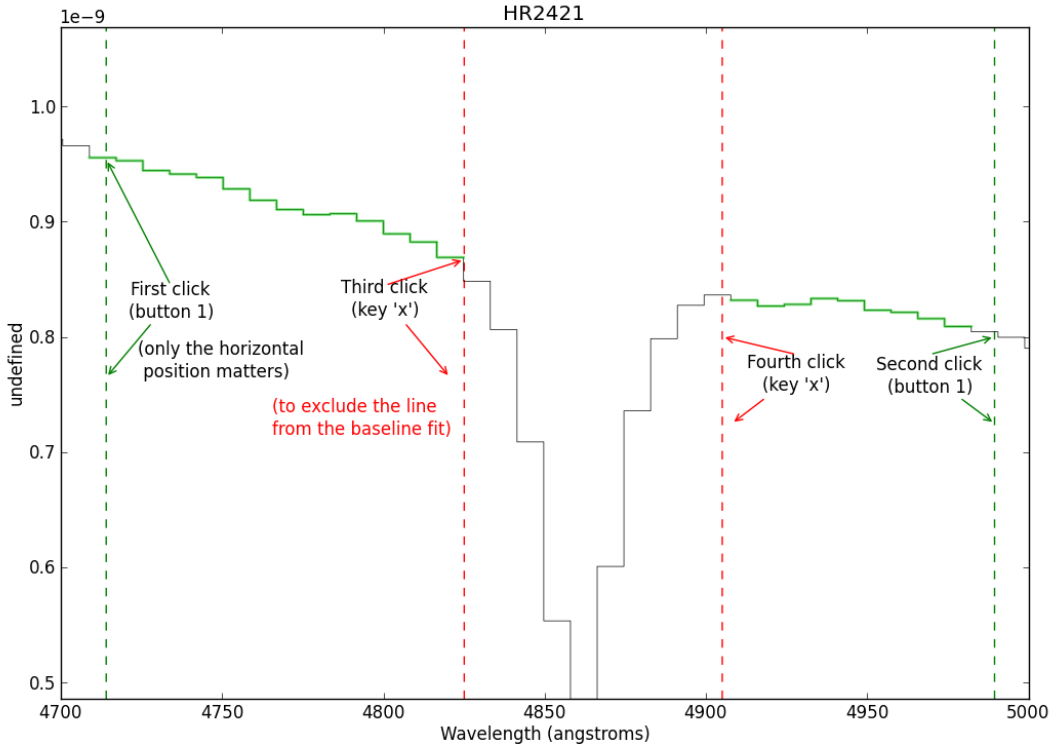
```
"""
Example demonstrating how to fit a complex H-alpha profile after subtracting off a satellite line
(in this case, He I 6678.151704)
"""
import pyspeckit

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)
```

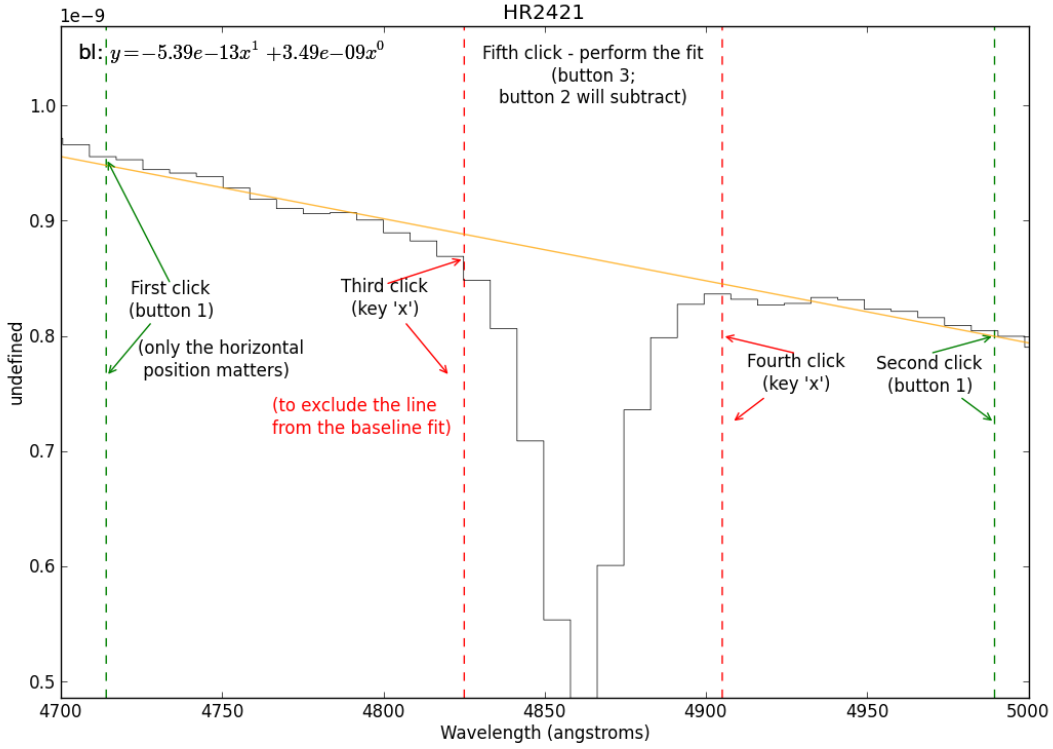

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

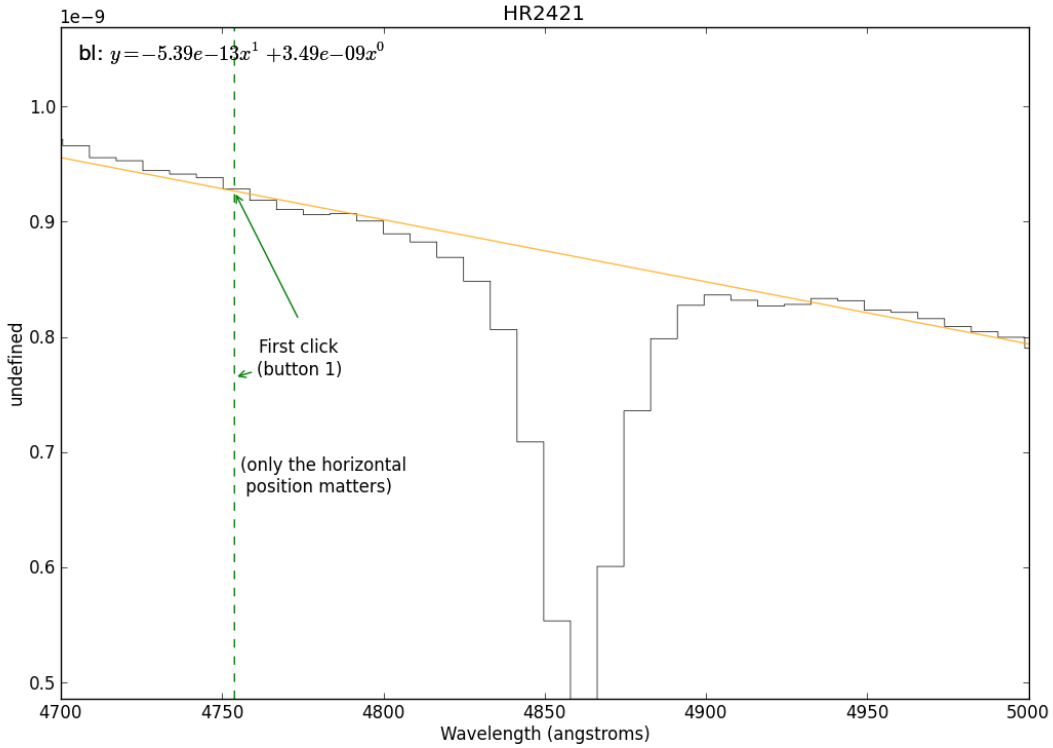

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

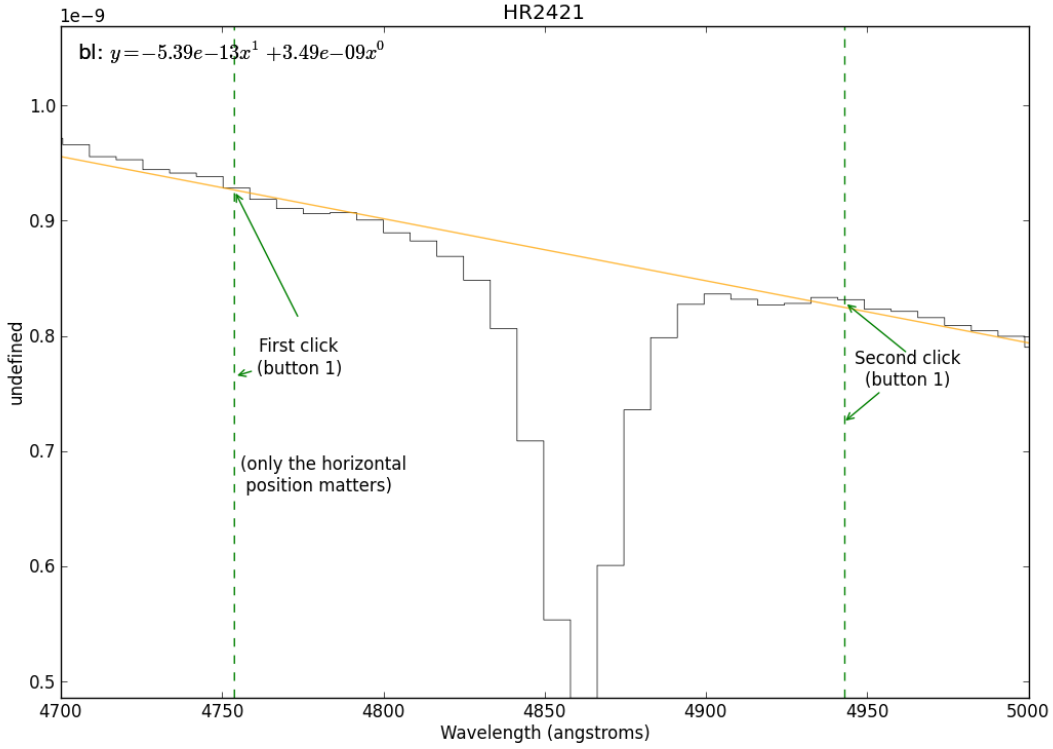

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

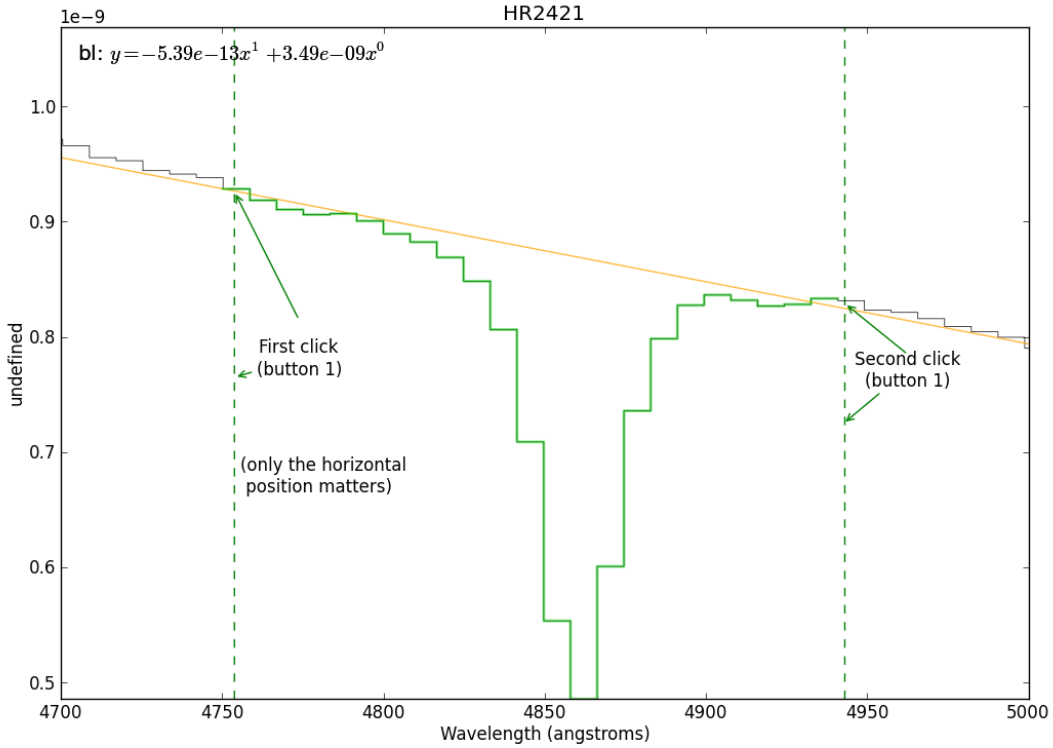

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

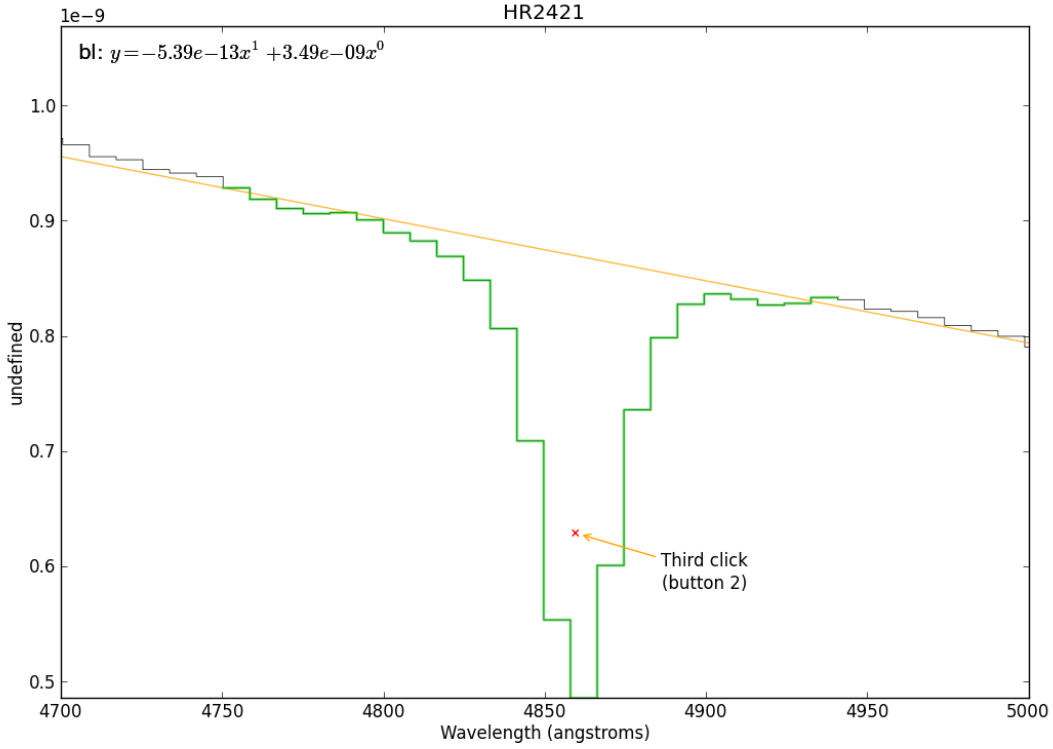

HR2421
1e−9
undefined
Wavelength (angstroms)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)

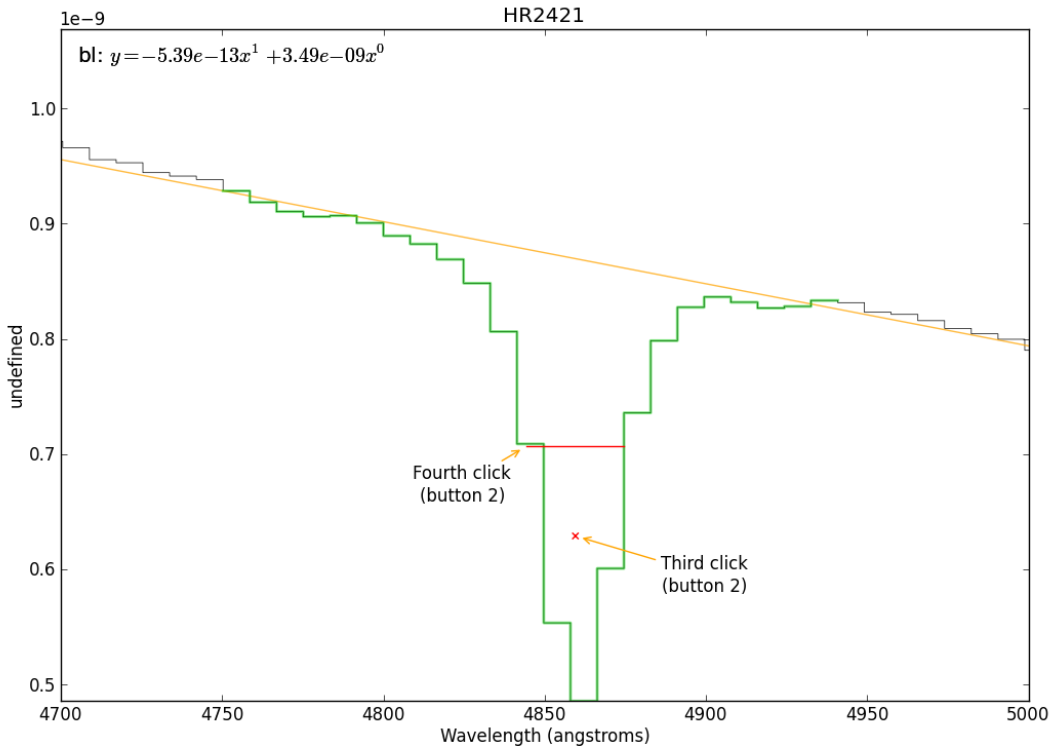

HR2421
1e−9
bl: $y = -5.39e-13x^{1} + 3.49e-09x^{0}$
Fifth click - perform the fit
(button 3;
button 2 will subtract)
undefined
Wavelength (angstroms)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)

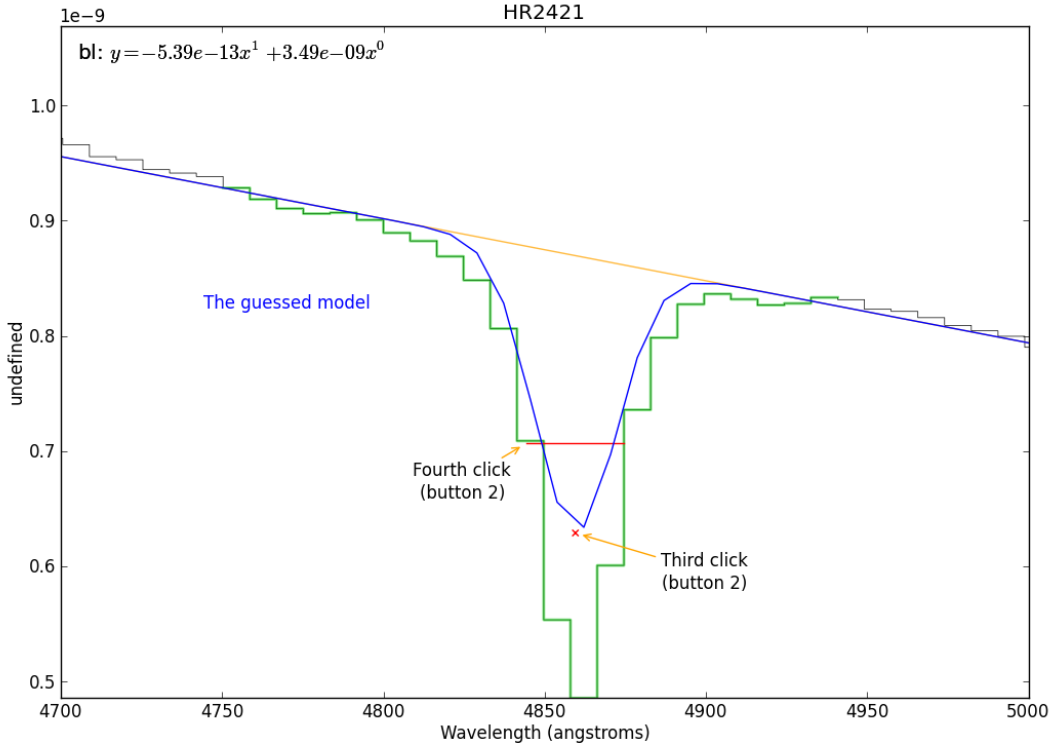

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
(only the horizontal
position matters)
undefined
Wavelength (angstroms)

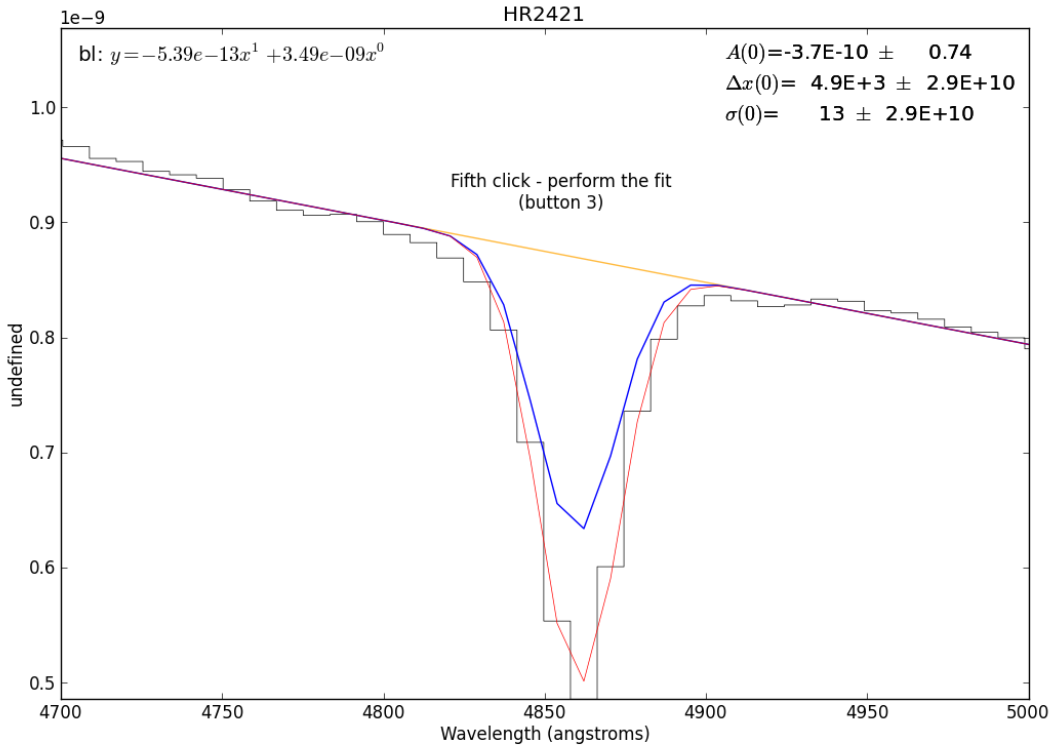

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
Second click
(button 1)
(only the horizontal
position matters)
undefined
Wavelength (angstroms)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
(only the horizontal
position matters)
Second click
(button 1)
undefined
Wavelength (angstroms)

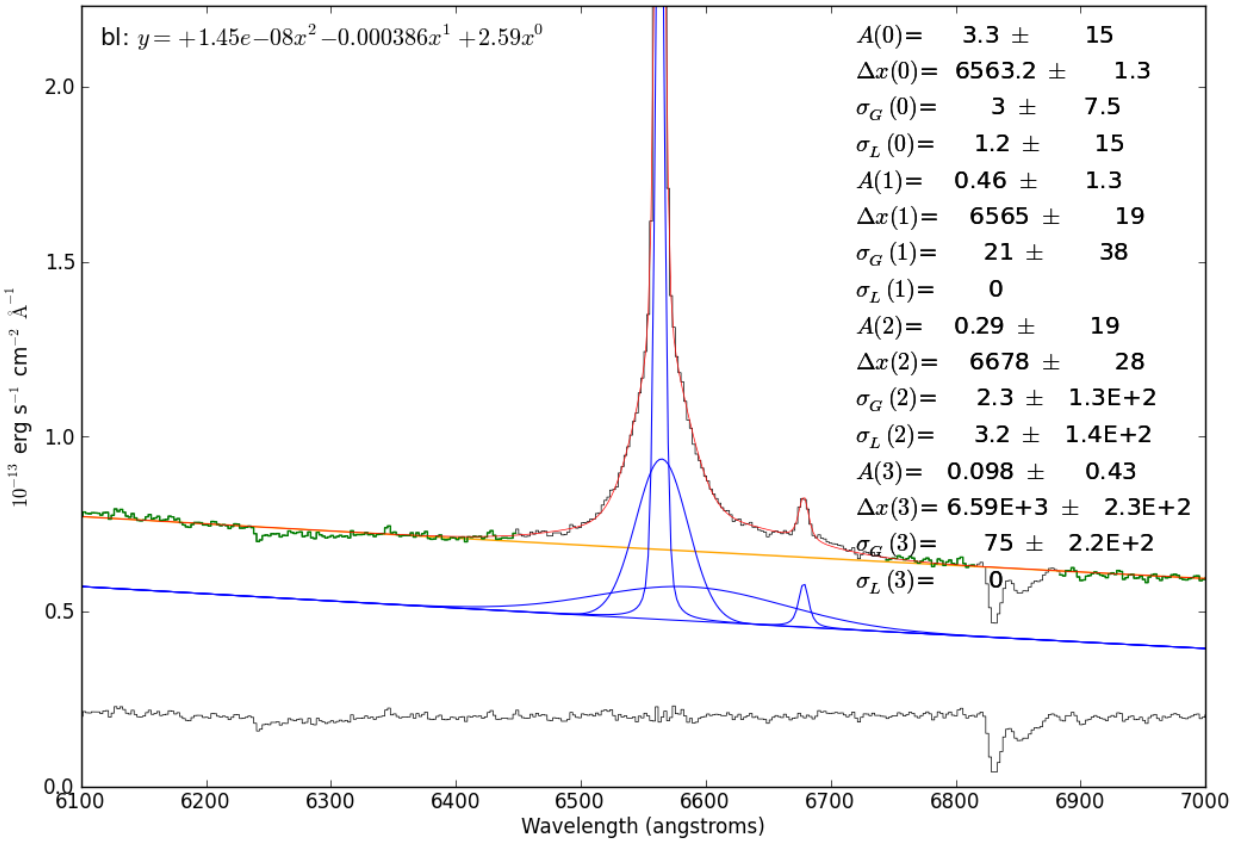

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
Third click
(button 2)
undefined
Wavelength (angstroms)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
undefined
Wavelength (angstroms)
Fourth click
(button 2)
Third click
(button 2)

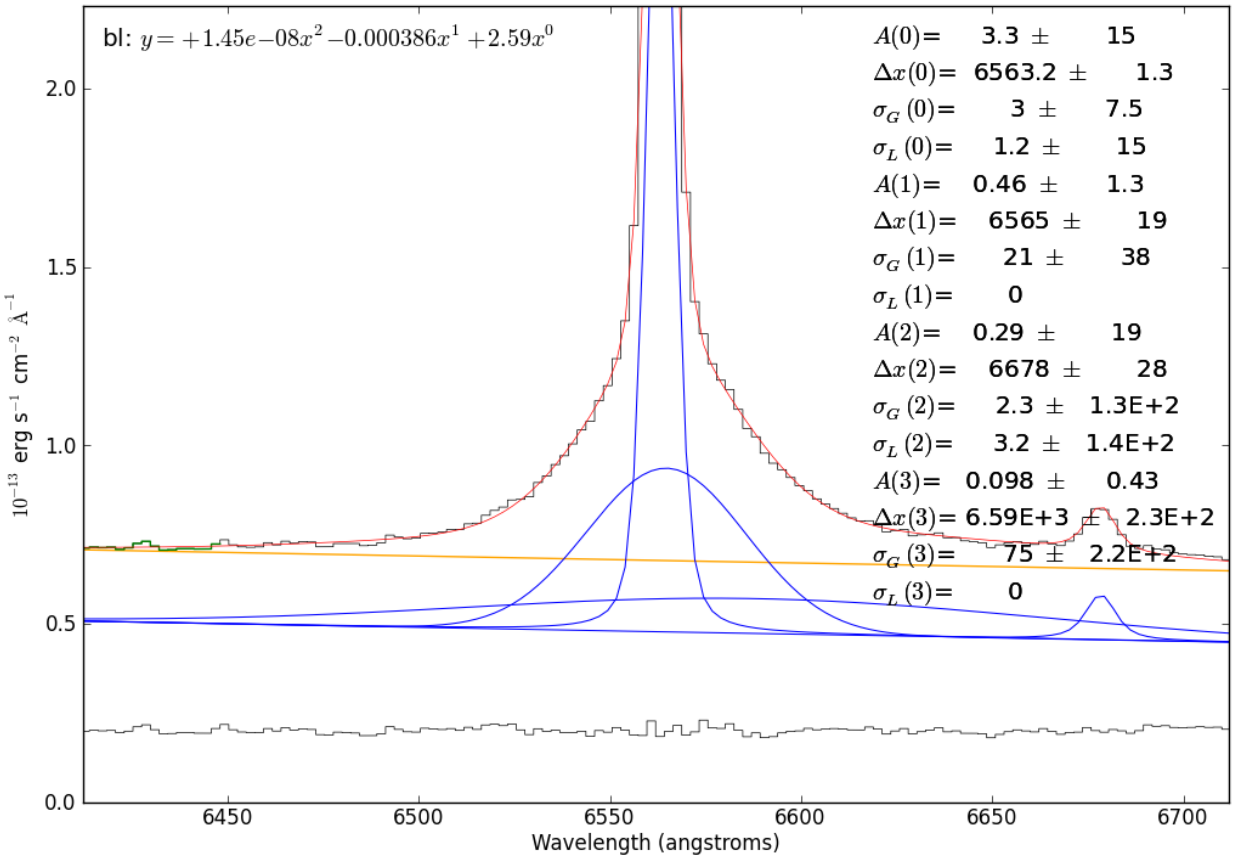

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
The guessed model
undefined
Wavelength (angstroms)
Fourth click
(button 2)
Third click
(button 2)



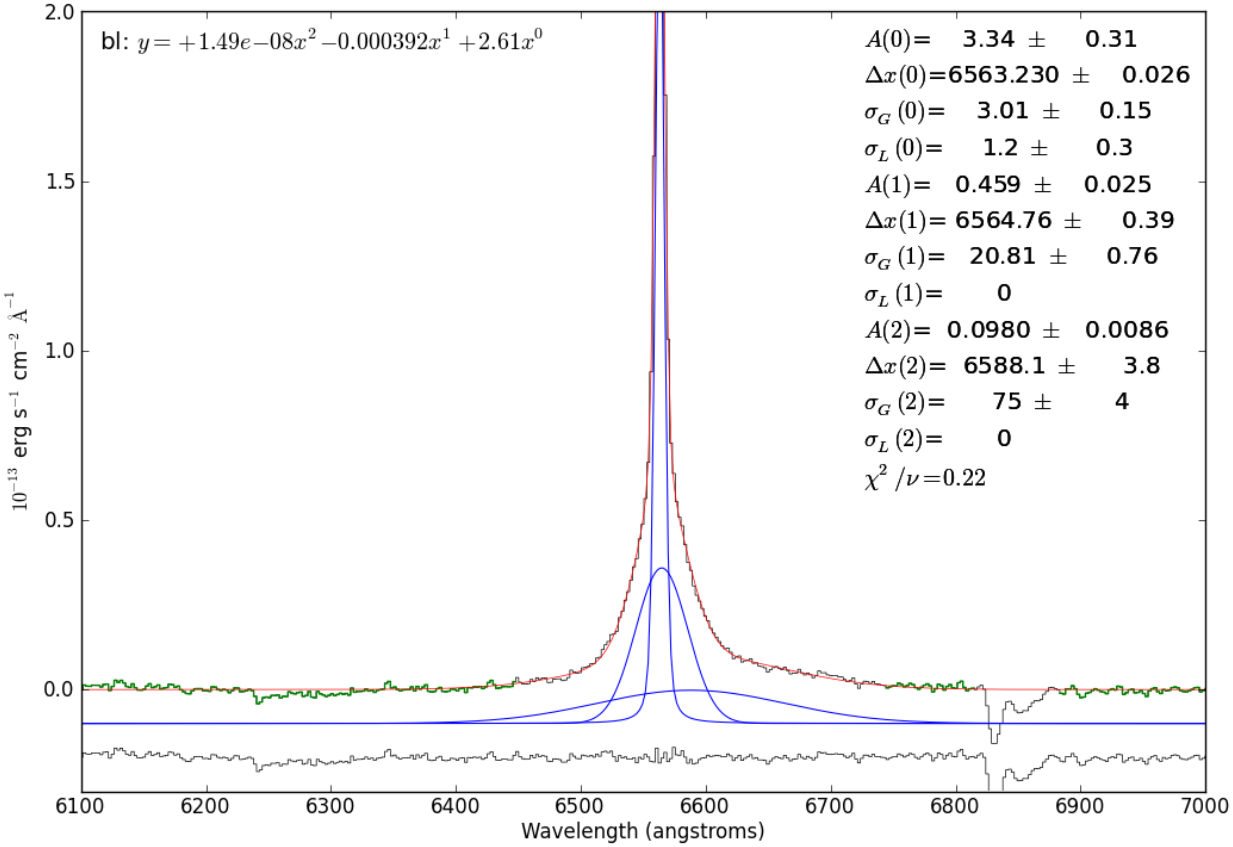

```
# the baseline (continuum) fit will be 2nd order, and excludes "bad"
# parts of the spectrum
# The exclusion zone was selected interatively (i.e., cursor hovering over the spectrum)
sp.baseline(xmin=6100, xmax=7000,
        exclude=[6450,6746,6815,6884,7003,7126,7506,7674,8142,8231],
        subtract=False, 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" %
print " ".join(["%15g %15g" % (par.value,par.error) for par in sp.specfit.parinfo])," ".join(["%15g"

# 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],
        subtract=True, reset_selection=True, highlight_fitregion=True, order=2)

# Do a 3-component fit now that the Helium line is gone
```

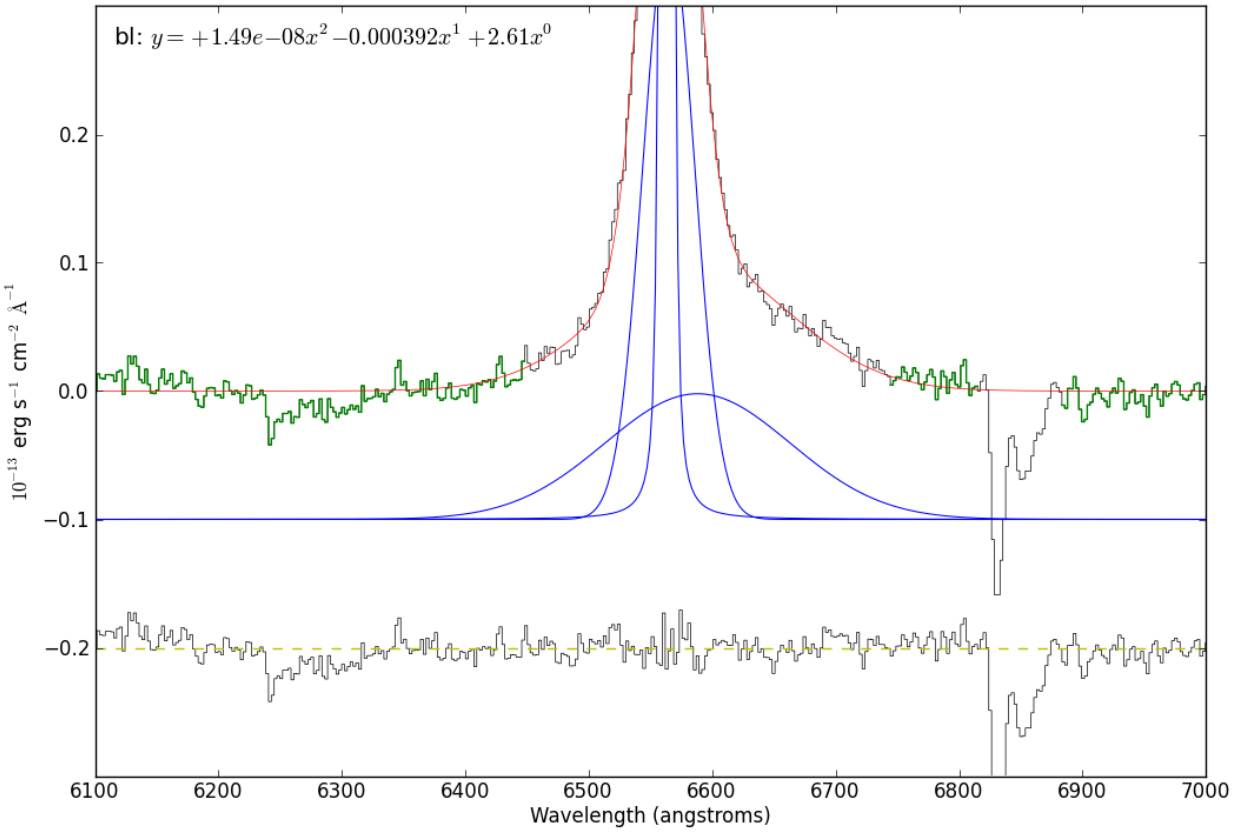

bl: $y = +1.45e{-}08x^2 - 0.000386x^1 + 2.59x^0$
$A(0)$= 3.3 ± 15
$\Delta x(0)$= 6563.2 ± 1.3
$\sigma_G\,(0)$= 3 ± 7.5
$\sigma_L\,(0)$= 1.2 ± 15
$A(1)$= 0.46 ± 1.3
$\Delta x(1)$= 6565 ± 19
$\sigma_G\,(1)$= 21 ± 38
$\sigma_L\,(1)$= 0
$A(2)$= 0.29 ± 19
$\Delta x(2)$= 6678 ± 28
$\sigma_G\,(2)$= 2.3 ± 1.3E+2
$\sigma_L\,(2)$= 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)

```
# 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")
```

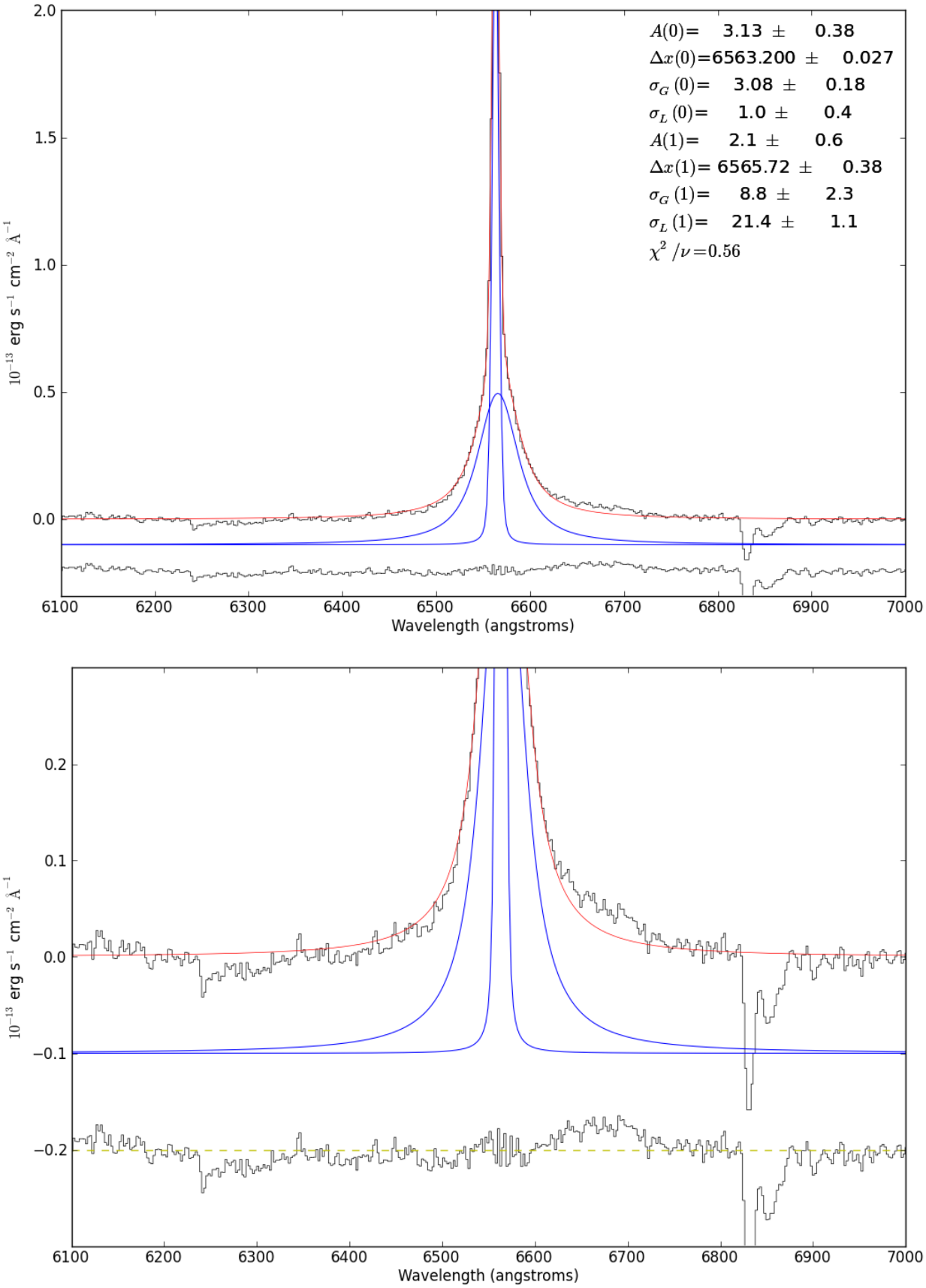



```
# A new zoom-in figure
import pylab

# 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,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")
```


$A(0)$= 3.13 ± 0.38
$\Delta x(0)$=6563.200 ± 0.027
$\sigma_G(0)$= 3.08 ± 0.18
$\sigma_L(0)$= 1.0 ± 0.4
$A(1)$= 2.1 ± 0.6
$\Delta x(1)$= 6565.72 ± 0.38
$\sigma_G(1)$= 8.8 ± 2.3
$\sigma_L(1)$= 21.4 ± 1.1
χ^2/ν=0.56
10^{-13} erg s^{-1} cm^{-2} Å^{-1}
Wavelength (angstroms)
10^{-13} erg s^{-1} cm^{-2} Å^{-1}
Wavelength (angstroms)

3.6.12 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,multisingle='multi')

# The fitted parameters are amplitude & xshift
# perform the fit:
dataspec.specfit(fittype='template',guesses=[1,0])

# print the results
print dataspec.specfit.parinfo
```

3.6.13 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.

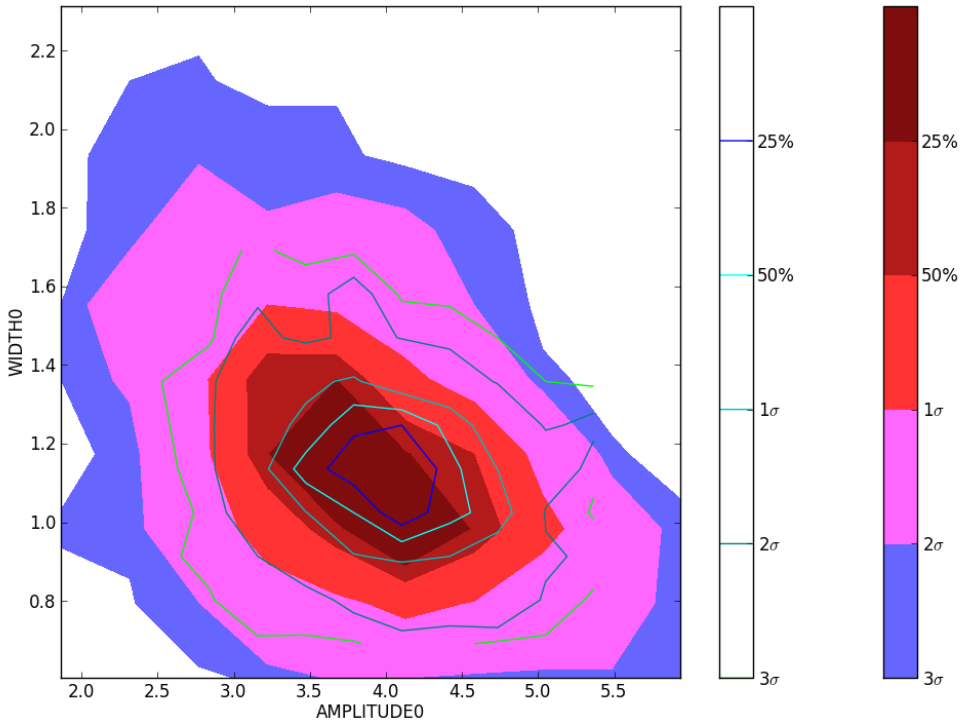
The examples below use a custom plotting package from agpy. It is a relatively simple but convenient wrapper around numpy's histogram2d. pymc_plotting takes care of indexing, percentile determination, and coloring.

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

# 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',multifit=True,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,
#    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
import agpy.pymc_plotting
import pylab
agpy.pymc_plotting.hist2d(MCuninformed,'AMPLITUDE0','WIDTH0',clear=True,bins=[25,25])
agpy.pymc_plotting.hist2d(MCwithpriors,'AMPLITUDE0','WIDTH0',contourcmd=pylab.contour,colors=[(0,1,0,
pylab.plot([5],[1],'k+',markersize=25)


# 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()}
agpy.pymc_plotting.hist2d(plotdict,'AMPLITUDE0','WIDTH0',fignum=2,bins=[25,25],clear=True)
pylab.plot([5],[1],'k+',markersize=25)
```



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).

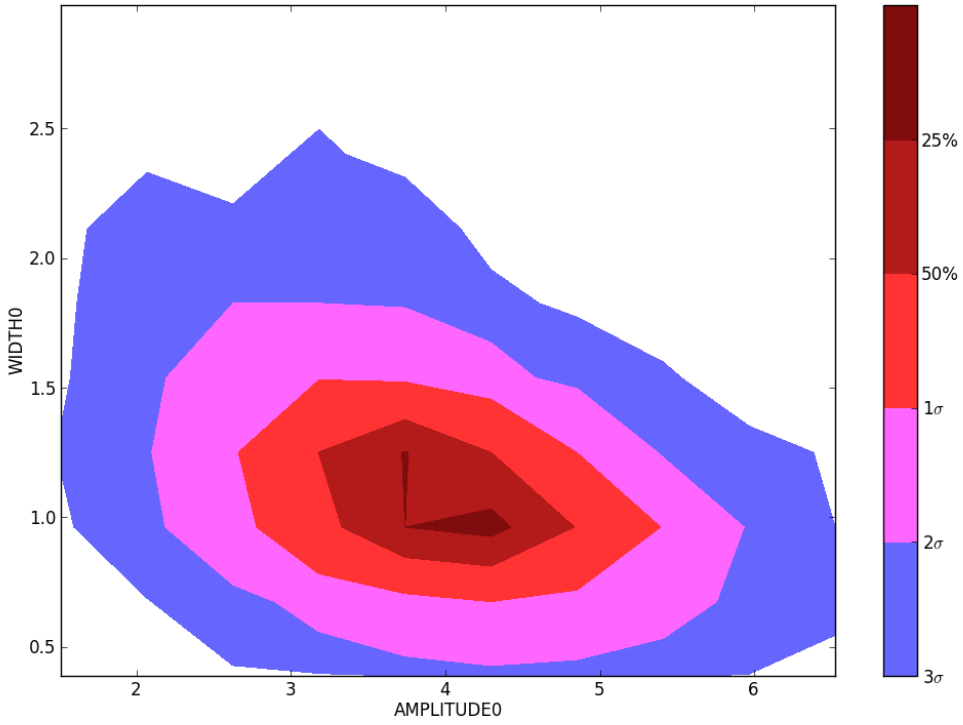
The parameter space sampled with `emcee` and binned onto a 25x25 grid. Note that `emcee` has 6x as many points (and takes about 6x as long to run) because there are 6 "walkers" for the 3 parameters being fit.

3.7 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
```

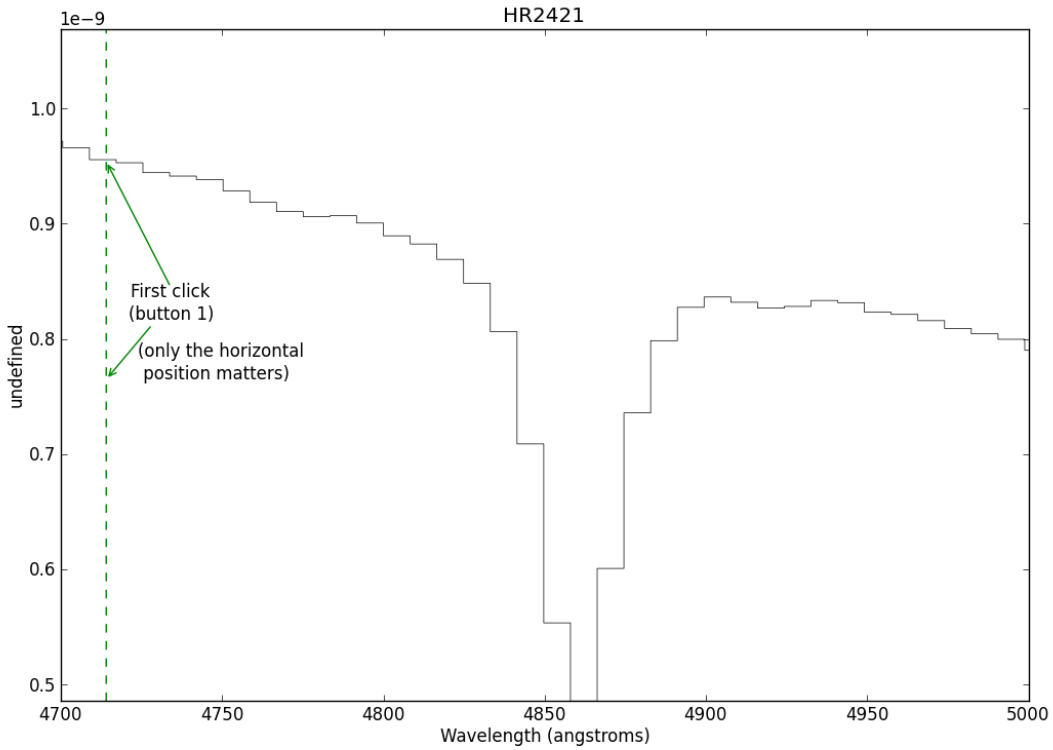


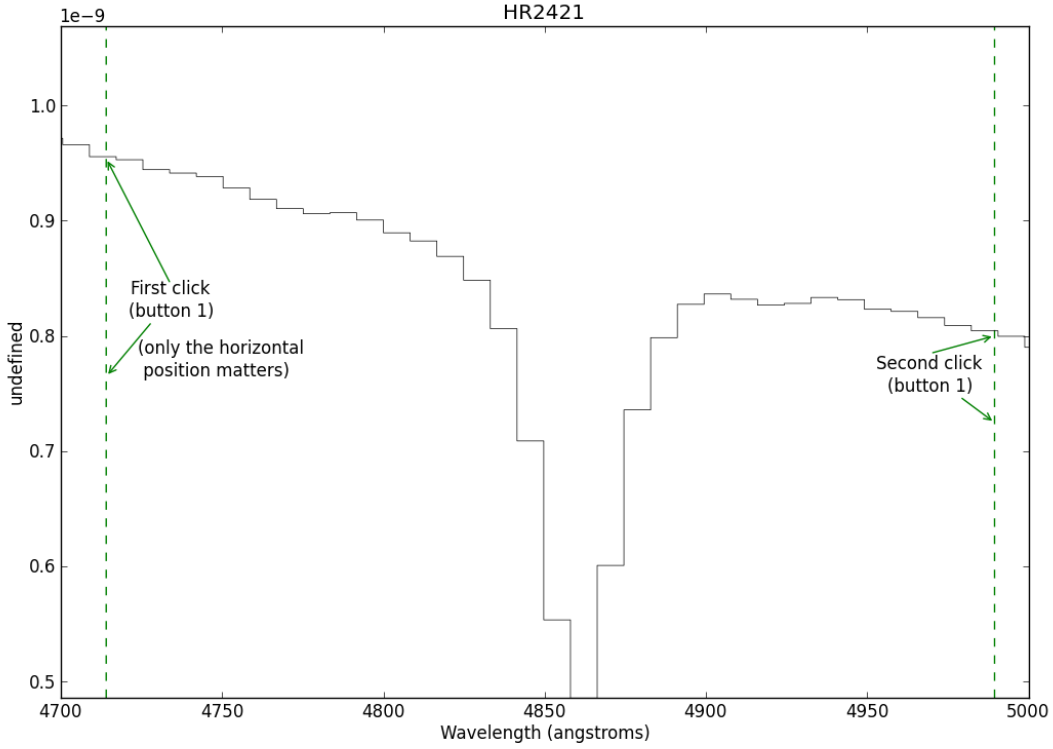

```
sp.plotter(xmin=4700,xmax=5000)

# 3. Need to fit the continuum first
sp.baseline(interactive=True, subtract=False)

# 4... (much work takes place interactively at this stage)

# 5. Start up an interactive line-fitting session
sp.specfit(interactive=True)
```

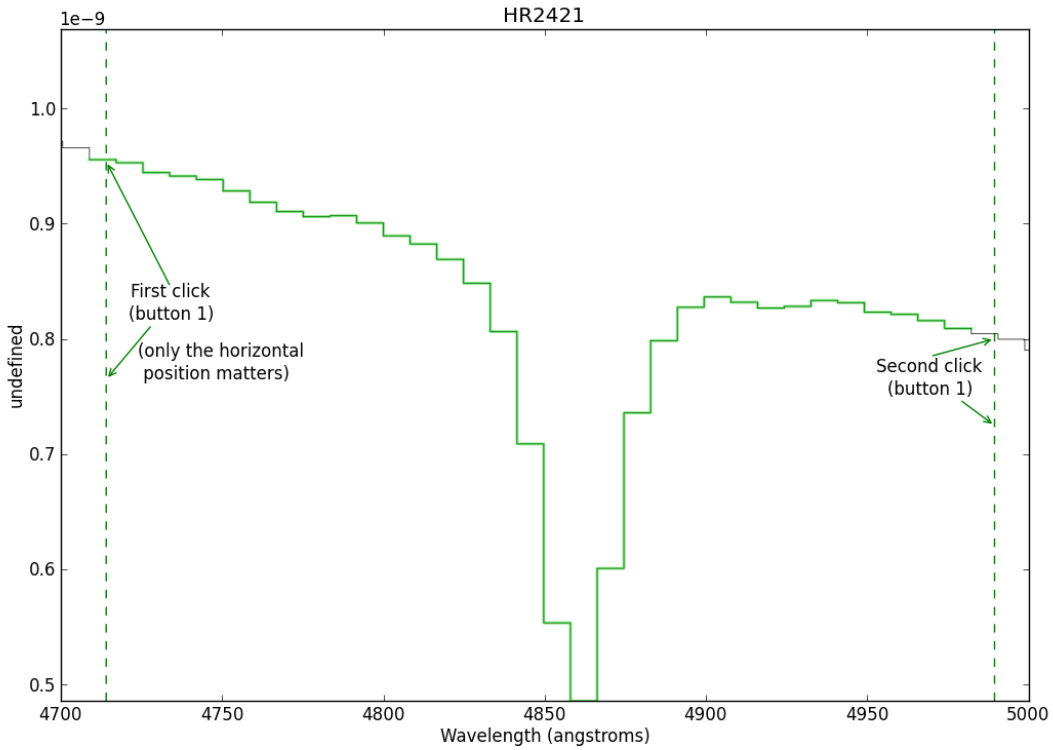
Note: If you don't see a plot window after step #2 above, make sure you're using matplotlib in interactive mode. This may require starting ipython as `ipython --pylab`

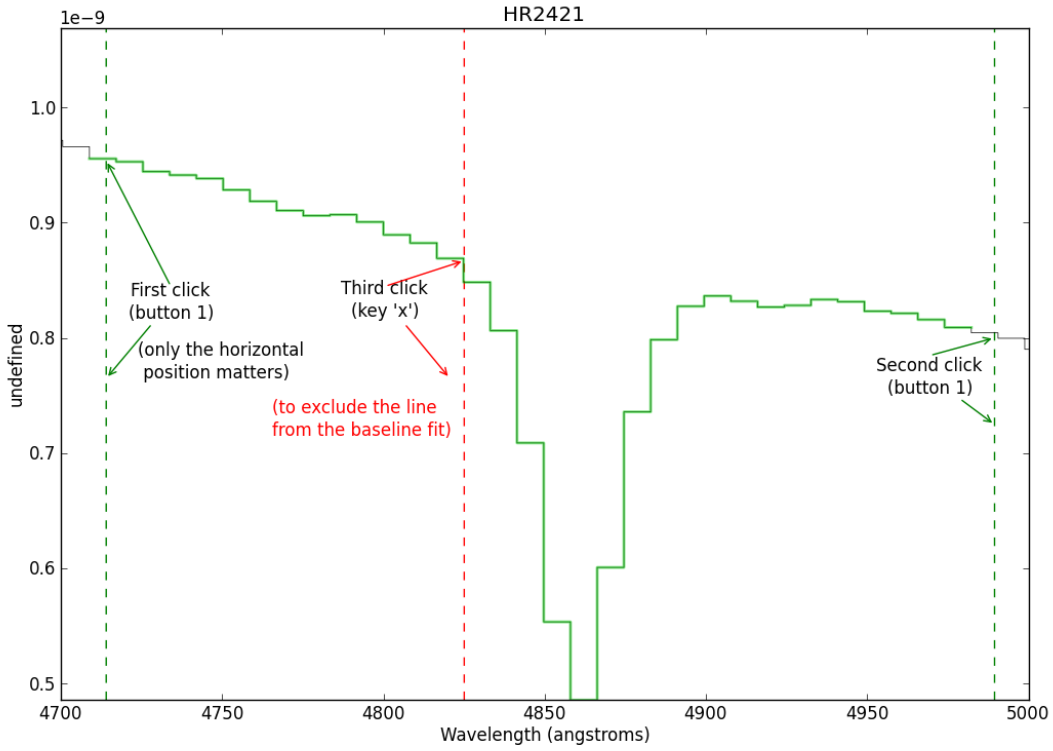
This is where you start the line-fitter:

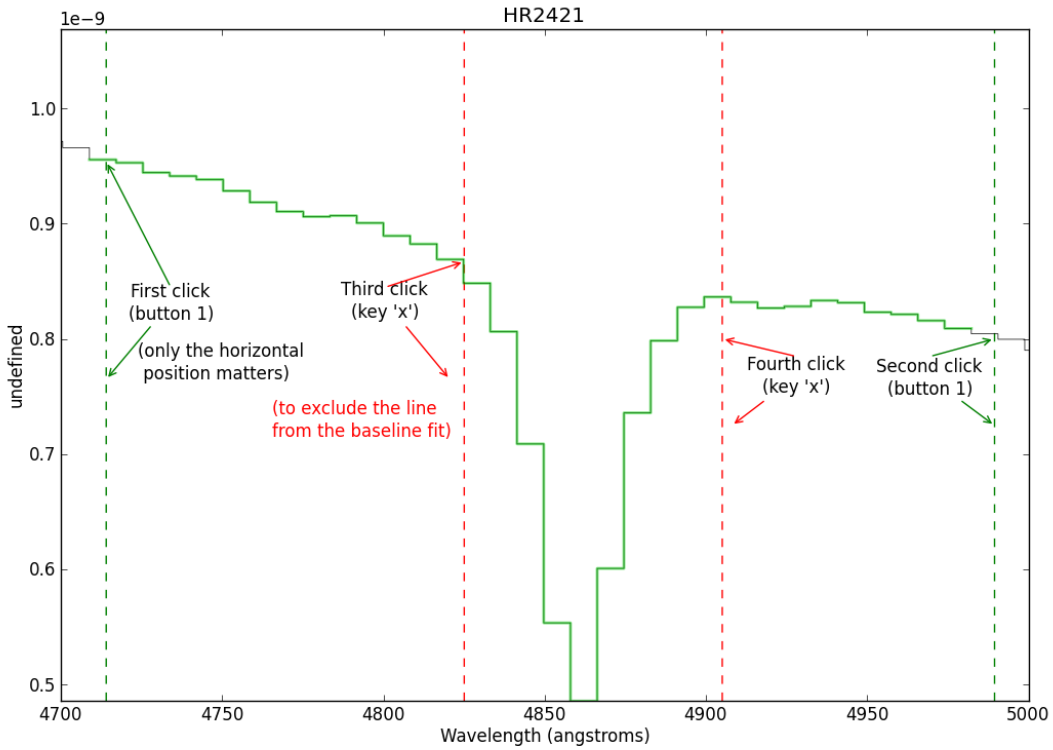
```
# Start up an interactive line-fitting session
sp.specfit(interactive=True)
```

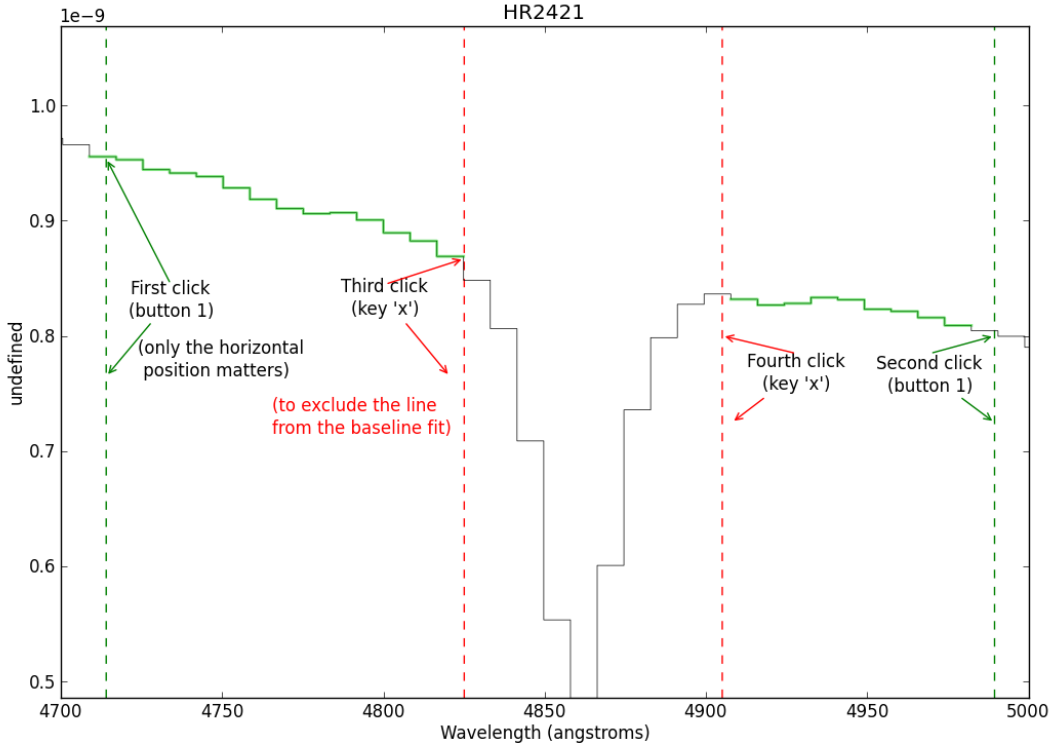

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

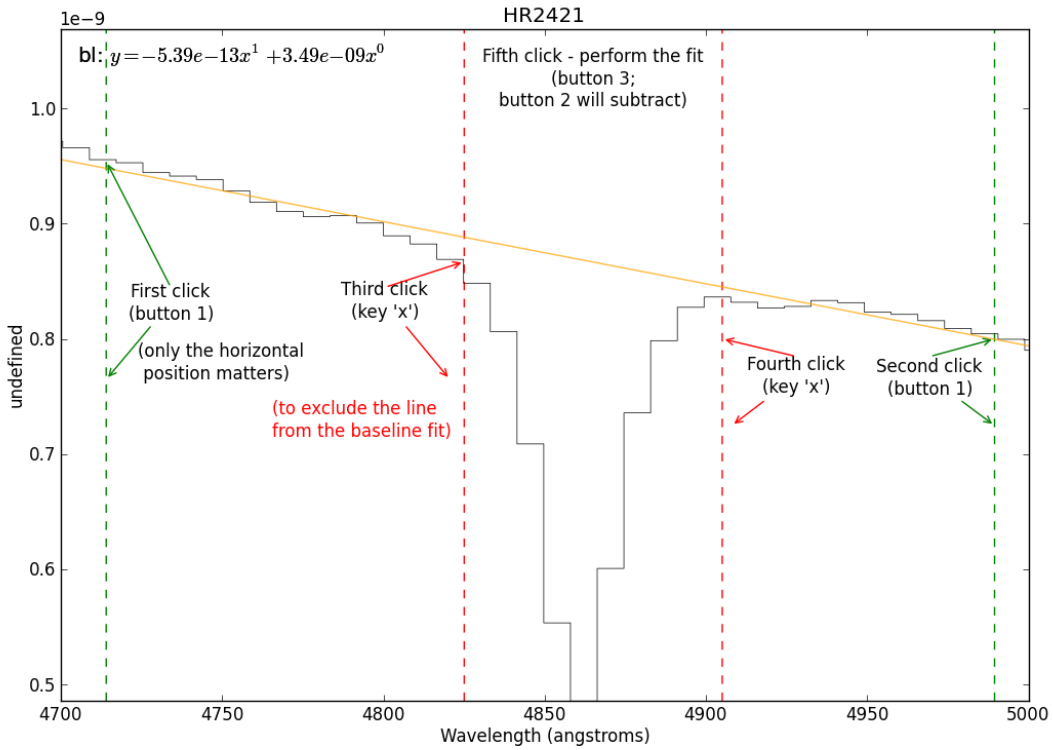

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

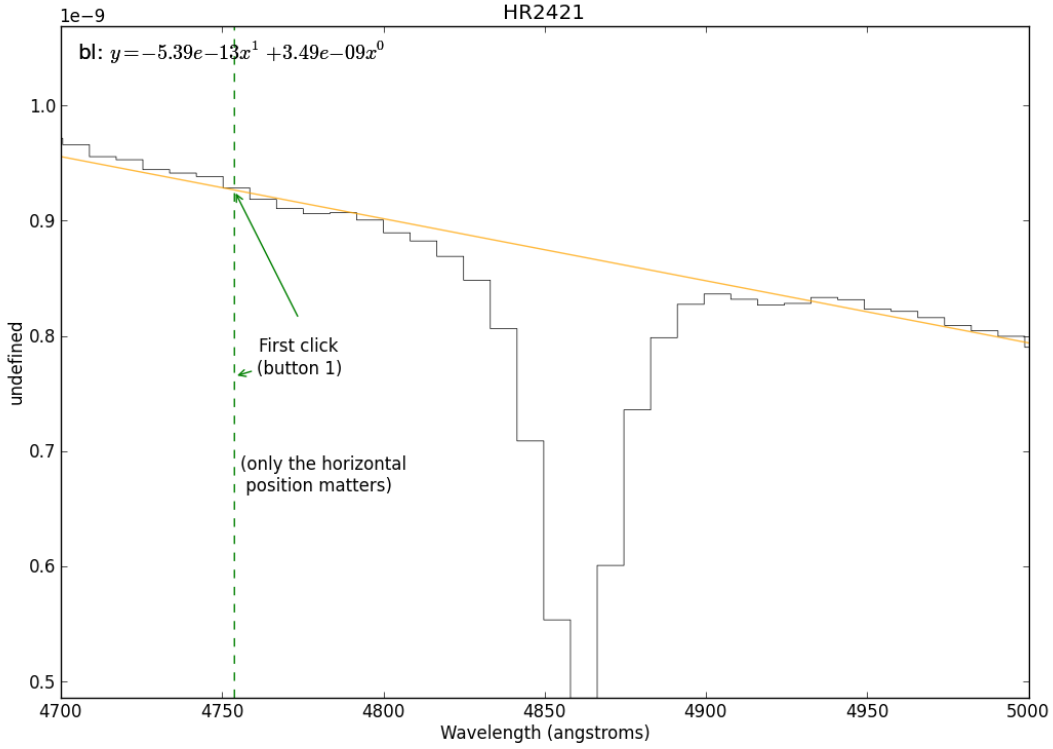

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

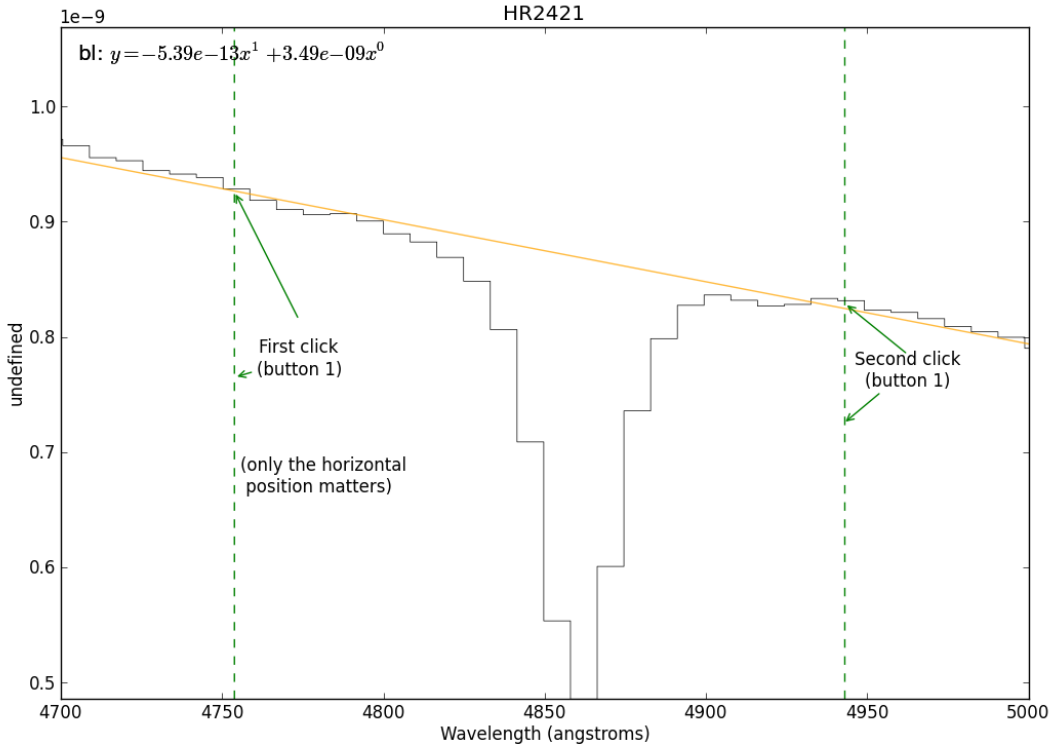

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Second click
(button 1)
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)

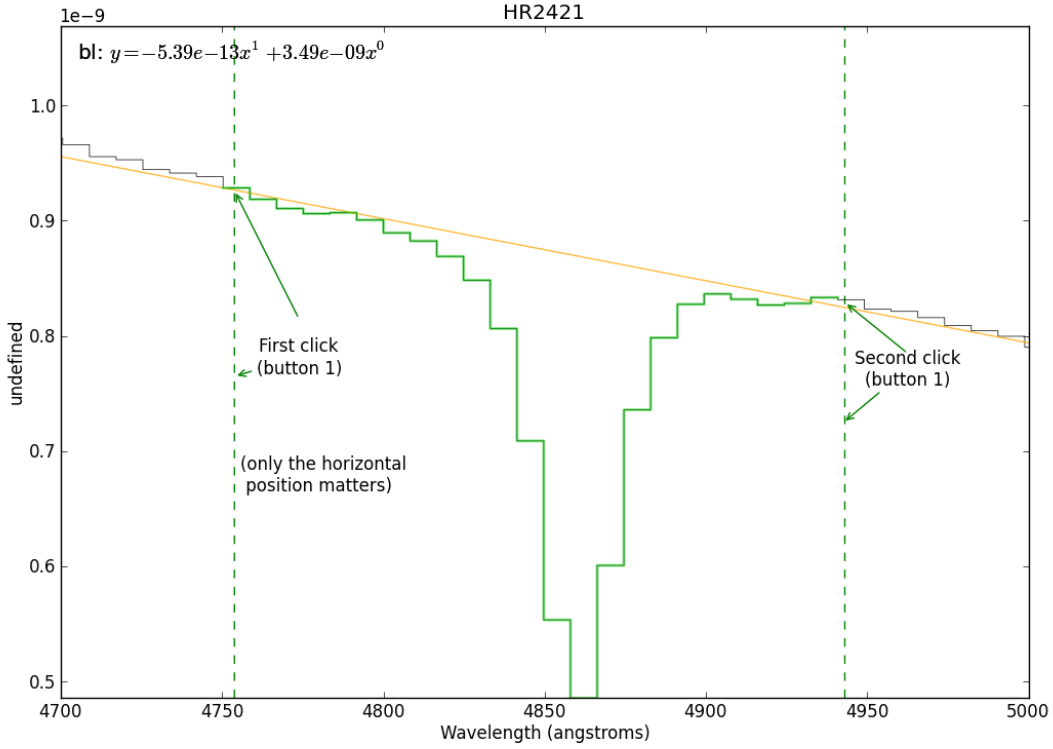

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)

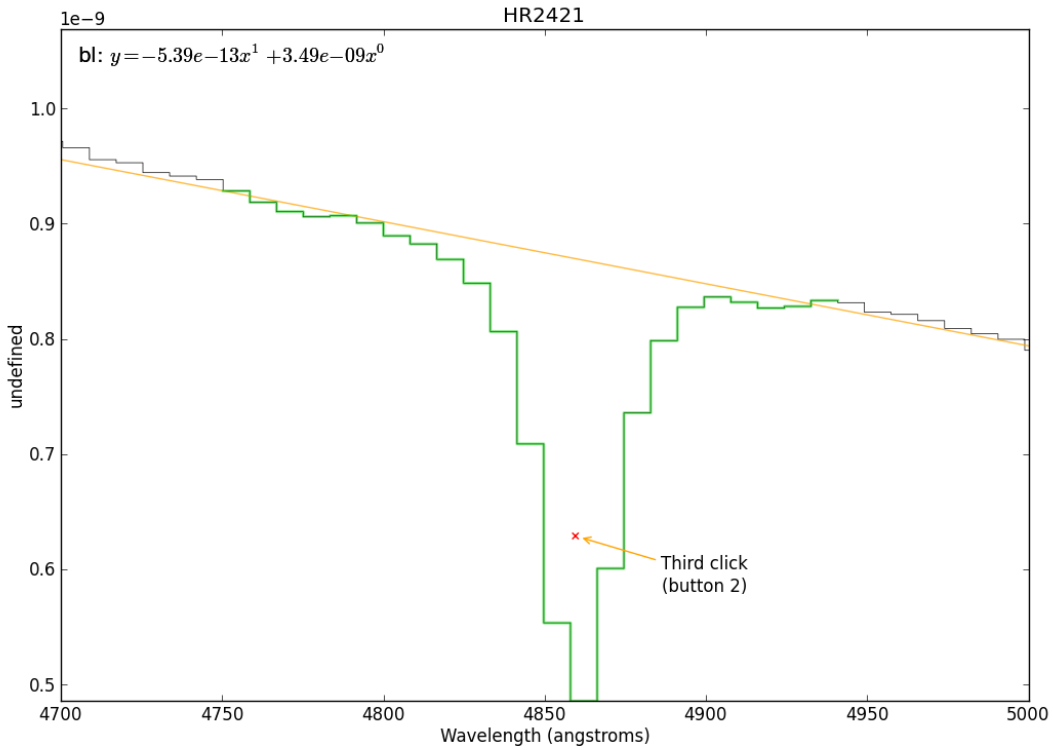

HR2421
1e−9
1.0
0.9
0.8
0.7
0.6
0.5
undefined
4700
4750
4800
4850
4900
4950
5000
Wavelength (angstroms)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)

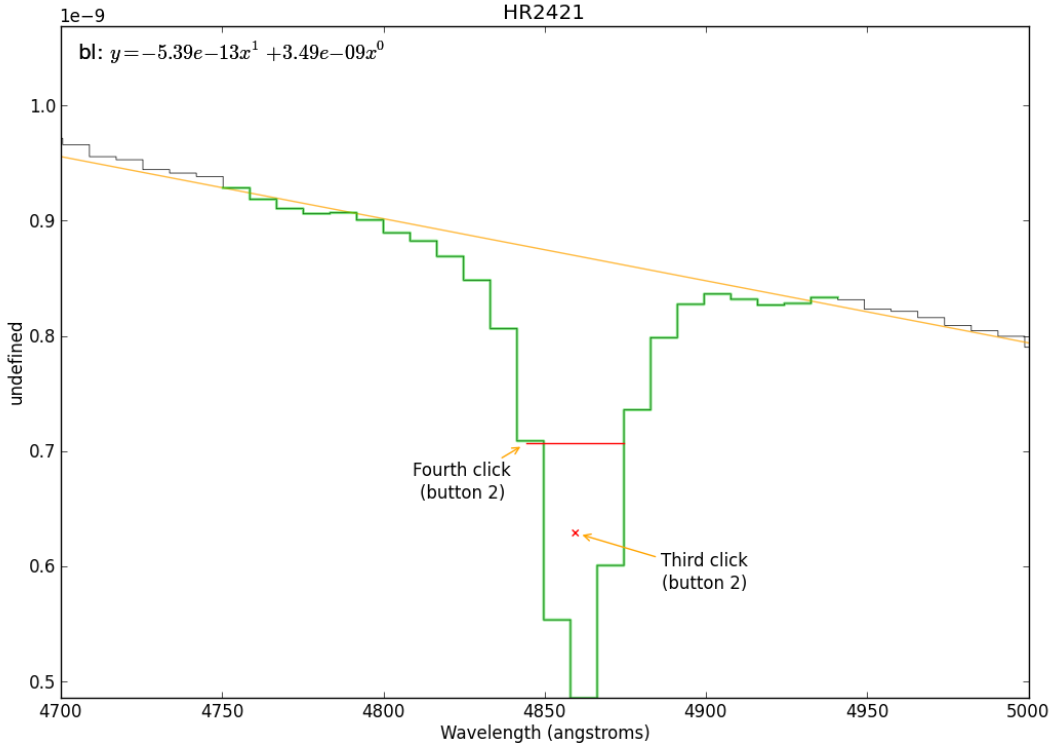

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
Fifth click - perform the fit
(button 3;
button 2 will subtract)
First click
(button 1)
(only the horizontal
position matters)
Third click
(key 'x')
(to exclude the line
from the baseline fit)
Fourth click
(key 'x')
Second click
(button 1)
undefined
Wavelength (angstroms)

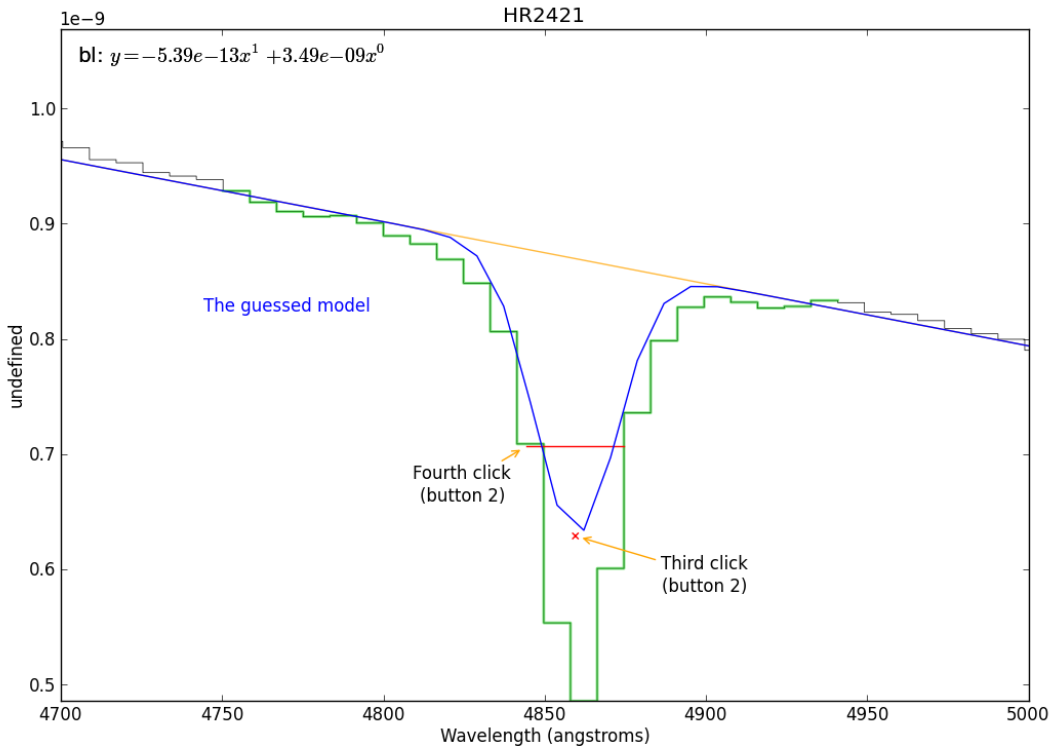

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
(only the horizontal
position matters)
undefined
Wavelength (angstroms)

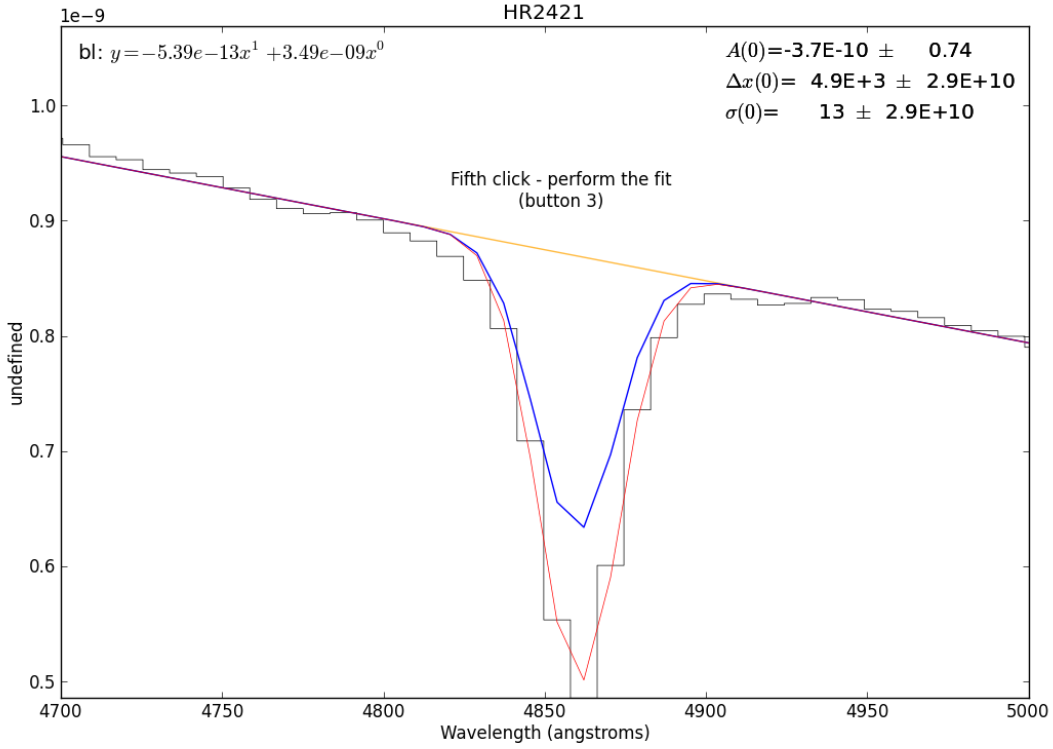

HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
First click
(button 1)
Second click
(button 1)
(only the horizontal
position matters)
undefined
Wavelength (angstroms)


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: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
undefined
Wavelength (angstroms)
Third click
(button 2)


HR2421
1e−9
bl: $y=-5.39e-13x^{1}+3.49e-09x^{0}$
undefined
Wavelength (angstroms)
Fourth click
(button 2)
Third click
(button 2)


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)

[HummerStorey1987] : Recombination-line intensities for hydrogenic ions. I - Case B calculations for H I and He II

[Hummer1994] : http://adsabs.harvard.edu/abs/1994MNRAS.268..109H

[wiki] : http://wiki.hmet.net/index.php/HII_Case_B_Recombination_Coefficients

p

Symbols

A

B

C

D

R

S

T

U

V

X