pyspeckit Documentation

Release 0.1.7

Adam Ginsburg and Jordan Mirocha

March 30, 2012

CONTENTS

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

CLASSES

- `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

TWO

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

2.1 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
- python2.7 or ordereddict

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

- pyfits
- 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
```

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 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

2.2 Models

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.

class pyspeckit.spectrum.models.model.**SpectralModel**(*modelfunc*, *npars*, *shortvarnames=('A', '\Delta x', '\sigma')*, *multisingle='multi'*, *fitunits=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.

Methods

modelfunc: the model function to be fitted. Should take an X-axis (spectroscopic axis) as an input, followed by input parameters. npars - number of parameters required by the model

parnames - a list or tuple of the parameter names

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

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

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

parerror - technically an output parameter... hmm (defaults to ZEROS)

partied - 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 - convert X-axis to these units before passing to model

parsteps - minimum step size for each paremeter (defaults to ZEROS)

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

shortvarnames - TeX names of the variables to use when annotating

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

Methods

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'])
```

components(*xarr*, *pars*, ***kwargs*)
Return a numpy ndarray of the independent components of the fits

fitter(*xax*, *data*, *err=None*, *quiet=True*, *veryverbose=False*, *debug=False*, *parinfo=None*, ***kwargs*)
Run the fitter. Must pass the x-axis and data. Can include error, parameter guesses, and a number of verbosity parameters.

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

veryverbose - print out a variety of mpfit output parameters

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

accepts *tied*, *limits*, *limited*, and *fixed* as keyword arguments. They must be lists of length len(params)

parinfo - You can override the class parinfo dict with this, though that largely defeats the point of having the wrapper class. This class does NO checking for whether the parinfo dict is valid.

kwargs are passed to mpfit after going through _make_parinfo to strip out things used by this class

integral(*modelpars*, ***kwargs*)
Extremely simple integrator: IGNORES modelpars; just sums self.model

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

mpfitfun(*x*, *y*, *err=None*)
Wrapper function to compute the fit residuals in an mpfit-friendly format

n_modelfunc(*pars*, ***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/

2.2.1 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*)
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

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*)

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

[height,amplitude,center,width]

return_components does nothing but is required by all fitters

pyspeckit.spectrum.models.inherited_gaussfitter.**gaussian_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': 2.4074999999999999e-05, '12-01': 2.4074999999999999e-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*)

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

Methods

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

Methods

hyperfine(*xarr*, *Tex=5.0*, *tau=0.10000000000000001*, *xoff_v=0.0*, *width=1.0*, *return_hyperfine_components=False*, *Tbackground=2.73*, *amp=None*)
Generate a model spectrum given an excitation temperature, optical depth, offset velocity, and velocity width.

hyperfine_amp(*xarr*, *amp=None*, *xoff_v=0.0*, *width=1.0*, *return_hyperfine_components=False*, *Tbackground=2.73*, *Tex=5.0*, *tau=0.10000000000000001*)
wrapper of self.hyperfine with order of arguments changed

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 0x10d974488>*)
Returns the spectral line that matches the given x-axis

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

pyspeckit.spectrum.models.modelgrid.**line_params_2D**(*gridval1*, *gridval2*, *griddim1*, *griddim2*, *valuegrid*)
Given a 2D grid of modeled line values - the amplitude, e.g. excitation temperature, and the optical depth, tau - return the model spectrum

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

2.2.2 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

pyspeckit.spectrum.models.n2hp.**line_strength_dict** = {'121-011': 0.41699999999999998, '121-010': 0.5560000

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.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.inherited_voigtfitter.**voigt**(*xarr*, *amp*, *xcen*, *Gfwhm*, *Lfwhm*)

voigt profile

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

Converted from http://mail.scipy.org/pipermail/scipy-user/2011-January/028327.html

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

Generator for voigt fitter class

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

2.3 Features

2.3.1 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)

Methods

__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

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

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*, *fit_original=False*, *powerlaw=False*, *baseline_fit_color='orange'*, *exclude=None*, ***kwargs*)

Do the baseline fitting and save and plot the results.

Can specify a region to exclude using velocity units or pixel units

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*, *powerlaw=False*, *fit_units='pixels'*)

plot_baseline(*annotate=True*, *baseline_fit_color='orange'*, *use_window_limits=None*, ***kwargs*)

Overplot the baseline fit

savefit()

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

2.3.2 Model Fitting

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

Methods

EQW (*plot=False*, *plotcolor='g'*, *annotate=False*, *alpha=0.5*, *loc='lower left'*)
Returns the equivalent width (integral of "baseline" or "continuum" minus the spectrum) over the selected range

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=0.10000000000000001*, *handlelength=0.10000000000000001*, *handletextpad=0.10000000000000001*, *frameon=False*, ***kwargs*)
Add a legend to the plot showing the fitted parameters

clearlegend() will remove the legend

kwargs passed to legend

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

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 gaussian was fit to a sub-region of the spectrum, expand it (with zeros) to fill the spectrum.

get_full_model(*debug=False*)
compute the model over the full axis

get_model(*xarr*, *debug=False*)
Compute the model over a given axis

guesspeakwidth(*event*, *debug=False*)
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='g'*, *clear_highlights=True*, ***kwargs*)
Re-highlight the fitted region

kwargs are passed to *matplotlib.plot*

integral(*direct=False*, *threshold='auto'*, *integration_limits=[]*, *return_error=False*, ***kwargs*)
Return the integral of the fitted spectrum

if direct=True, return the integral of the spectrum over a range defined by the threshold or integration limits if defined

note that integration_limits will operate directly on the DATA, which means that if you've baselined without subtract=True, the baseline will be included in the integral

if return_error is set, the error computed by sigma = sqrt(sum(sigma_i^2)) * dx will be returned as well

moments(***kwargs*)
Return the moments

see `moments`

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

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*, ***kwargs*)
Fit a single peak (plus a background)

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

annotate - Make a legend? vheight - Fit a (constant) background as well as a peak? height - initial guess for background negamp - If True, assumes amplitude is negative. If False, assumes positive. If

None, can be either.

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

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

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

plot_fit(*annotate=None*, *show_components=None*, *composite_fit_color='red'*, *component_fit_color='blue'*, *lw=0.5*, *composite_lw=0.75*, *component_lw=0.75*, *component_kwargs={}*, *use_window_limits=None*, *show_hyperfine_components=None*, ***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

plotresiduals(*fig=2*, *axis=None*, *clear=True*, *color='k'*, *linewidth=0.5*, *drawstyle='steps-mid'*, ***kwargs*)

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

kwargs are passed to matplotlib plot

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

Print the best-fit parameters to the command line

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

Register a model fitter

Register a fitter function.

Required Arguments:

***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?

Optional Keyword Arguments:

***multisingle*: ['multi' | 'single']** Is the function a single-function fitter (with a background), or does it allow N copies of the fitting function?

***override*: [True | False]** Whether to override any existing type if already present.

***key*: [char]** Key to select the fitter in interactive mode

savefit()

Save the fit parameters from a Gaussian fit to the FITS header ***THESE SHOULD BE WRITTEN FOR EACH TYPE OF MODEL TO BE FIT***

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

Pick a fitting region in either WCS units or pixel units

***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

Now things get a little complicated... *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 `pyspeckit.spectrum.interactive.use_window_limits`. Overwrites xmin,xmax set by plotter

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*, *print_message=True*, *clear_all_connections=True*, ***kwargs*)

2.3.3 Measurements

class `pyspeckit.spectrum.measurements.`**Measurements**(*Spectrum*, *z=None*, *d=None*, *xunits=None*, *fluxnorm=None*, *miscline=None*, *misctol=10*, *ignore=None*, *derive=True*, *debug=False*)

Bases: `object`

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

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

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.

misclines = dictionary miscline = {{'name': H_alpha', 'wavelength': 6565, 'etc': 0}, {}}

Methods

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**()
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.

2.3.4 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

Methods

as_unit(*unit*, *frame=None*, *quiet=True*, *center_frequency=None*, *center_frequency_units=None*, *debug=False*, ***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?

***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....

cdelt(*tolerance=1e-08*)

Return the channel spacing if channels are linear

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*)

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

Methods

2.3.5 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.

Required Arguments:

***filetype*: [string]** 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?

Optional Keyword Arguments:

pyspeckit.spectrum.__init__.**register_writer**(*filetype*, *function*, *suffix*, *default=False*)
Register a writer function.

Required Arguments:

***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?

Optional Keyword Arguments:

class pyspeckit.spectrum.fitters.**Registry**
This class is a simple wrapper to prevent fitter properties from being globals

Methods

add_fitter(*name*, *function*, *npars*, *multisingle='single'*, *override=False*, *key=None*)
Register a fitter function.

Required Arguments:

***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?

Optional Keyword Arguments:

***multisingle*: ['multi' | 'single']** Is the function a single-function fitter (with a background), or does it allow N copies of the fitting function?

***override*: [True | False]** Whether to override any existing type if already present.

***key*: [char]** Key to select the fitter in interactive mode

2.4 Readers

2.4.1 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.

2.4.2 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*, ***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*, ***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

2.4.3 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.

2.4.4 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' and 'line' names

`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*)
Create a `pyspeckit.spectrum.units.SpectroscopicAxis` from the CLASS "header"

`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

2.5 Wrappers

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

`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 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

Wrapper to fit ammonia spectra. Generates a reasonable guess at the position and velocity using a gaussian fit

pyspeckit.wrappers.fitnh3.**BigSpectrum_to_NH3dict**(*sp*, *vrange=None*)
A rather complicated way to make the spdicts above given a spectrum...

pyspeckit.wrappers.fitnh3.**fitnh3tkin**(*input_dict*, *dobaseline=True*, *baselinekwargs={}*, *crop=False*, *guessline='twotwo'*, *tex=15*, *tkin=20*, *column=15.0*, *fortho=0.66000000000000003*, *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

pyspeckit.wrappers.fitnh3.**plotter_override**(*sp*, *vrange=None*, ***kwargs*)
Do plot_nh3 with syntax similar to plotter()

2.5.1 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.

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

2.6 Examples

Check out the flickr gallery.

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

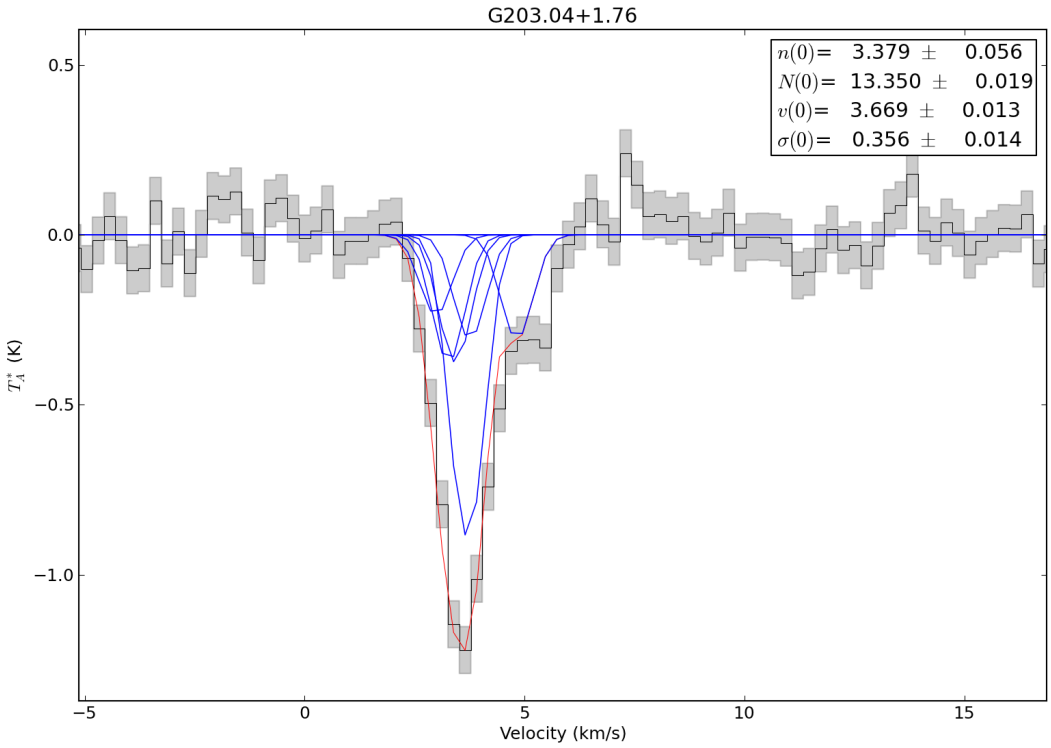
2.6.1 Radio Fitting: H_2CO RADEX example

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 <http://code.google.com/p/agpy/source/browse/trunk/radex>‘_. 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)
```



2.6.2 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',
    '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
```

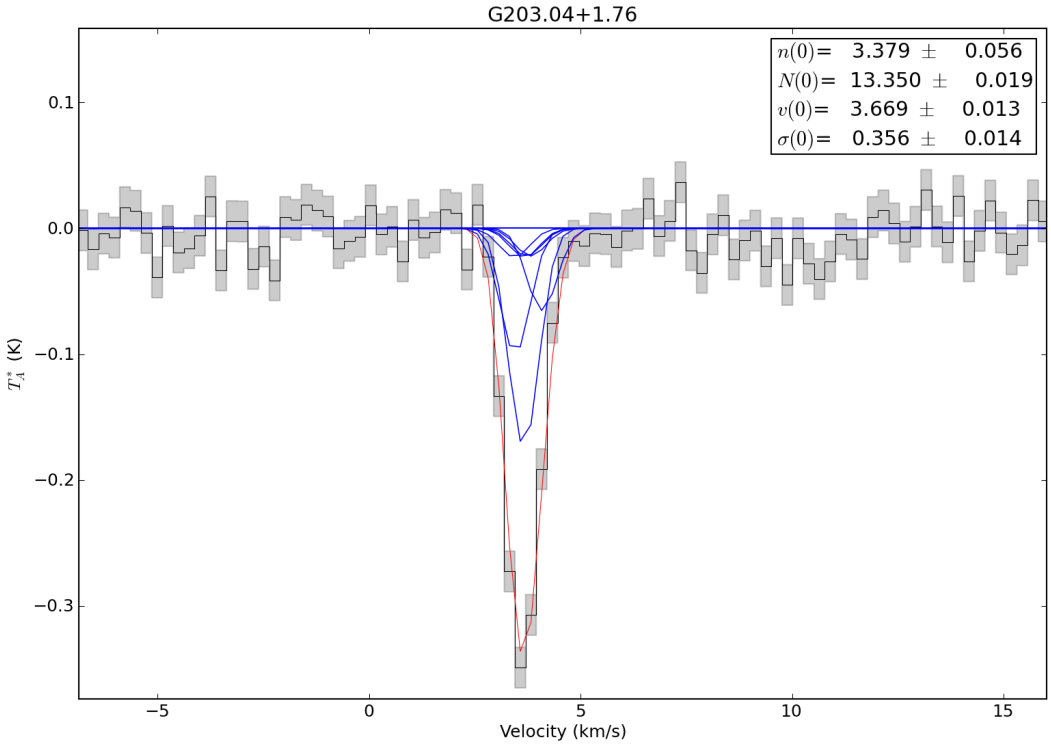


Figure 2.1: Both transitions are fit simultaneously using a RADEX model. The input (fitted) parameters are therefore density, column density, width, and velocity.

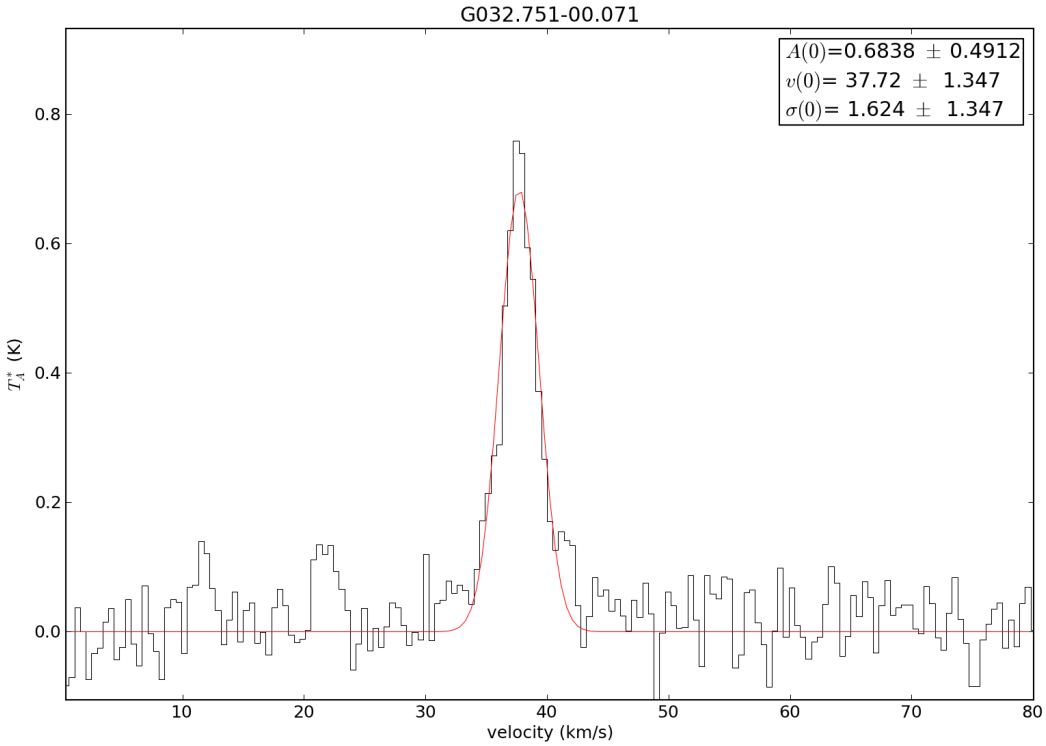


Figure 2.2: The 2-2 transition is used to guess the central velocity and width via gaussian fitting because its hyperfine lines are weaker

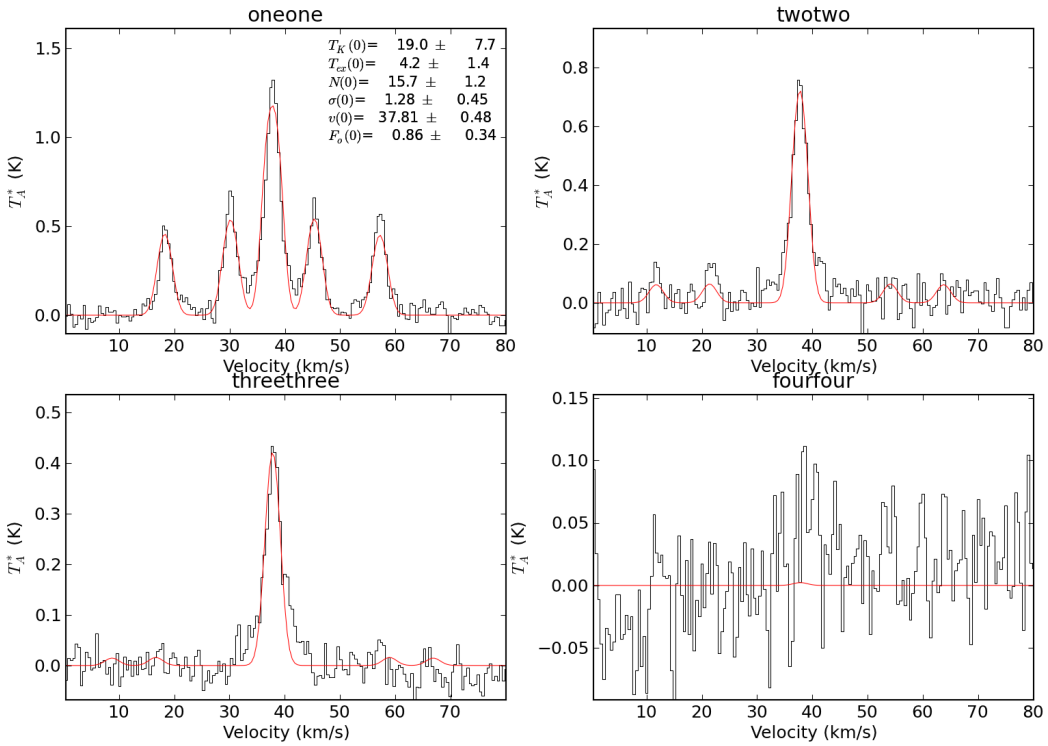


Figure 2.3: Then all 4 lines are simultaneously fit. Even upper limits on the 4-4 line can provide helpful contraints on the model

2.6.3 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")
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")
```

2.6.4 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].
```

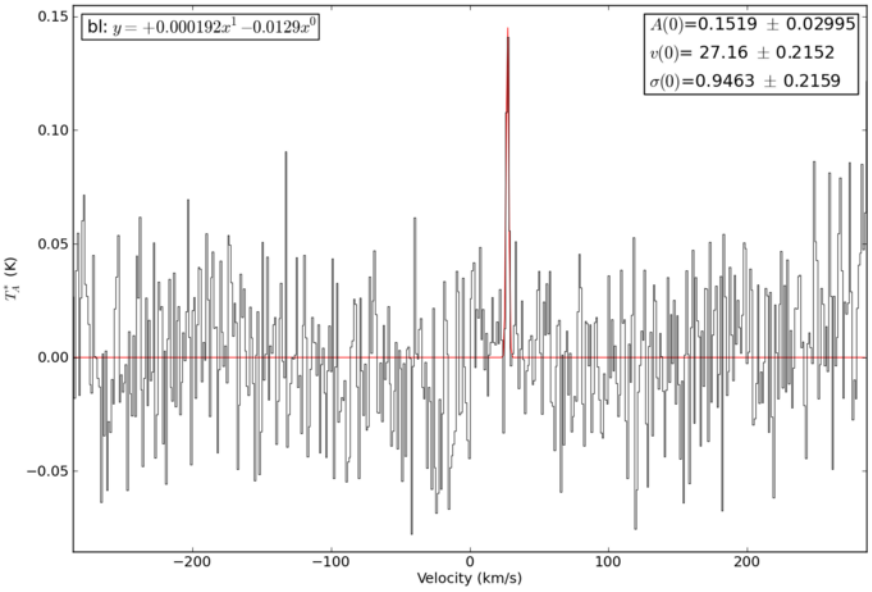


Figure 2.4: Sample HCO+ spectrum fitted with a gaussian

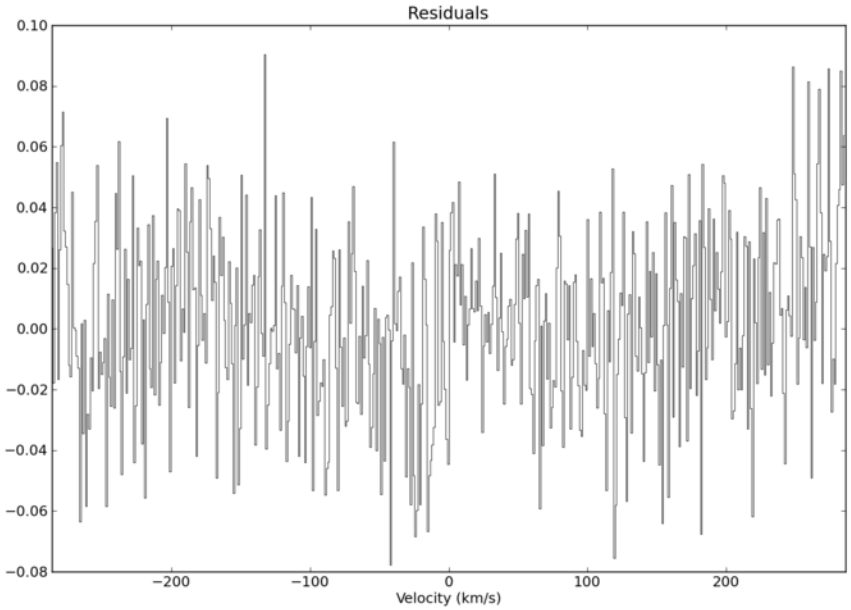


Figure 2.5: Residuals of the gaussian fit from the previous figure

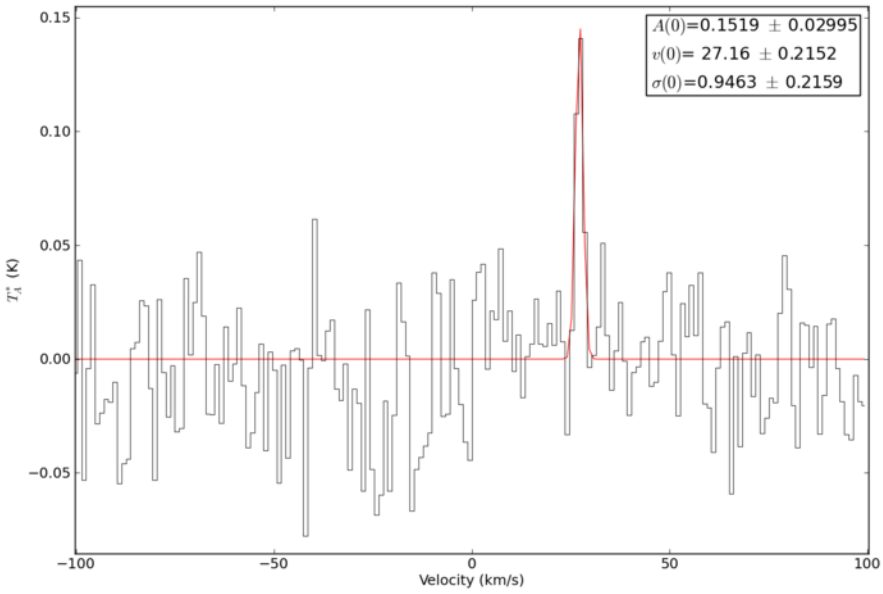
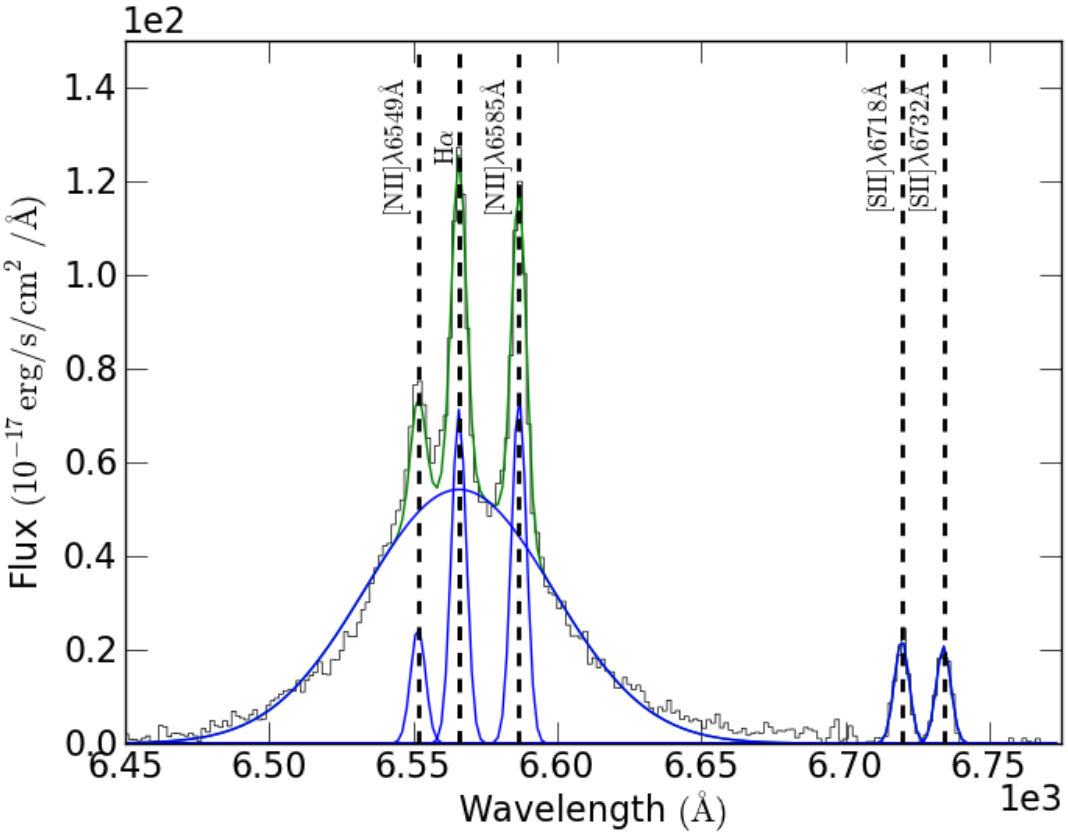


Figure 2.6: A zoomed-in, cropped version of the spectrum. With the 'crop' command, the excess data is discarded.

```
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()
```

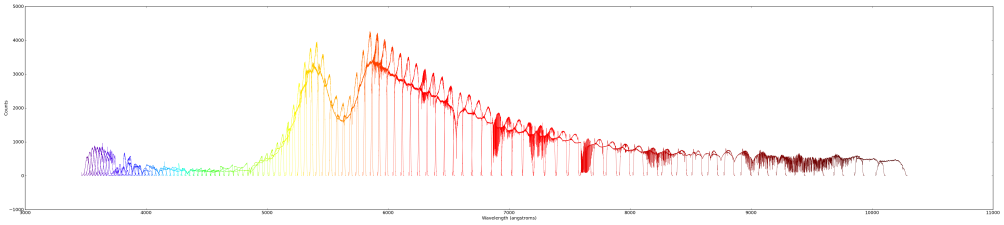
```
# Print out spectral line information
print "Line   Flux (erg/s/cm^2)     Amplitude (erg/s/cm^2)    FWHM (Angstrom)   Luminosity (erg/s)"
for line in spec.measurements.lines.keys():
    print line, spec.measurements.lines[line]['flux'], spec.measurements.lines[line]['amp'], spec.mea
        spec.measurements.lines[line]['lum']

# 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")
```

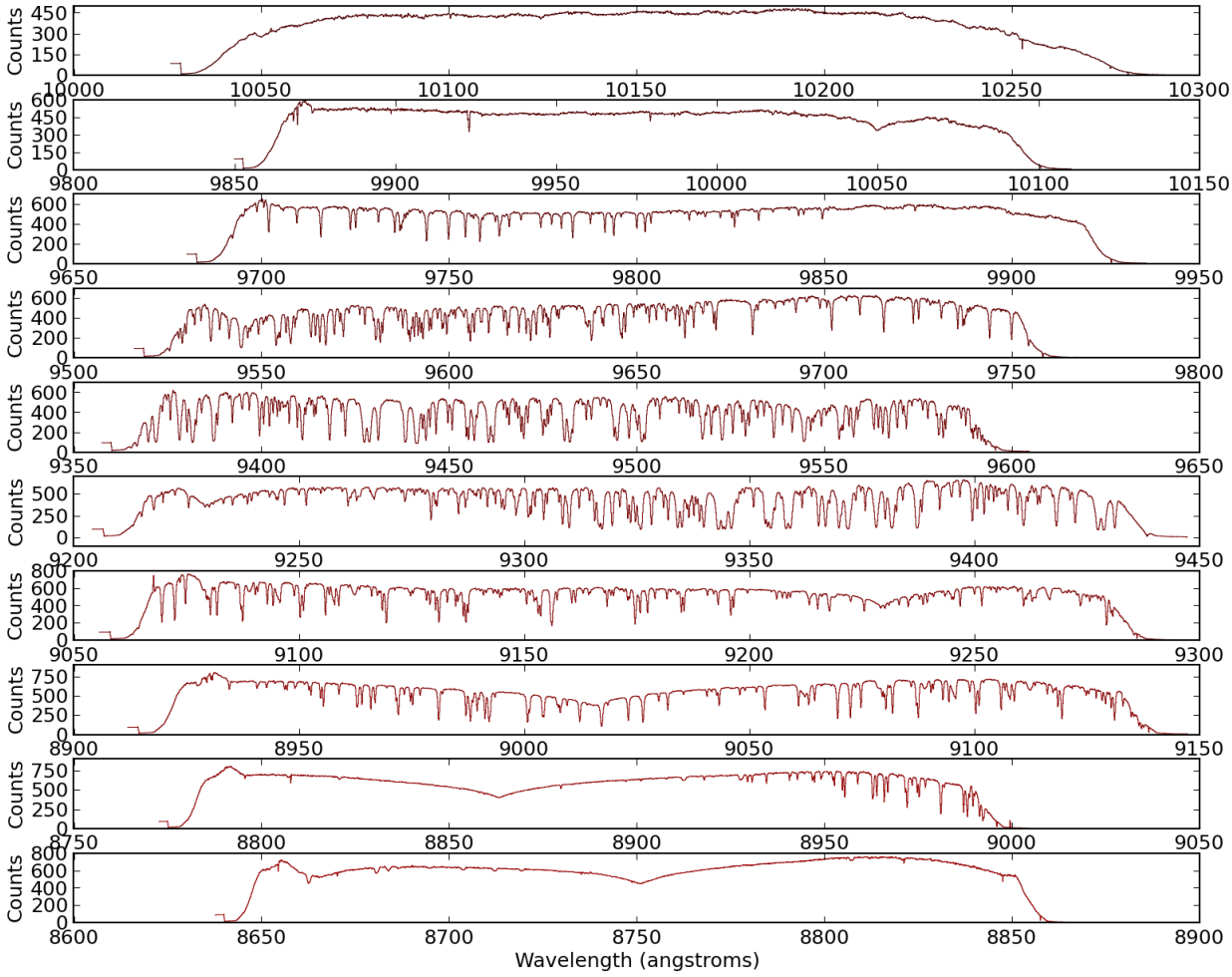


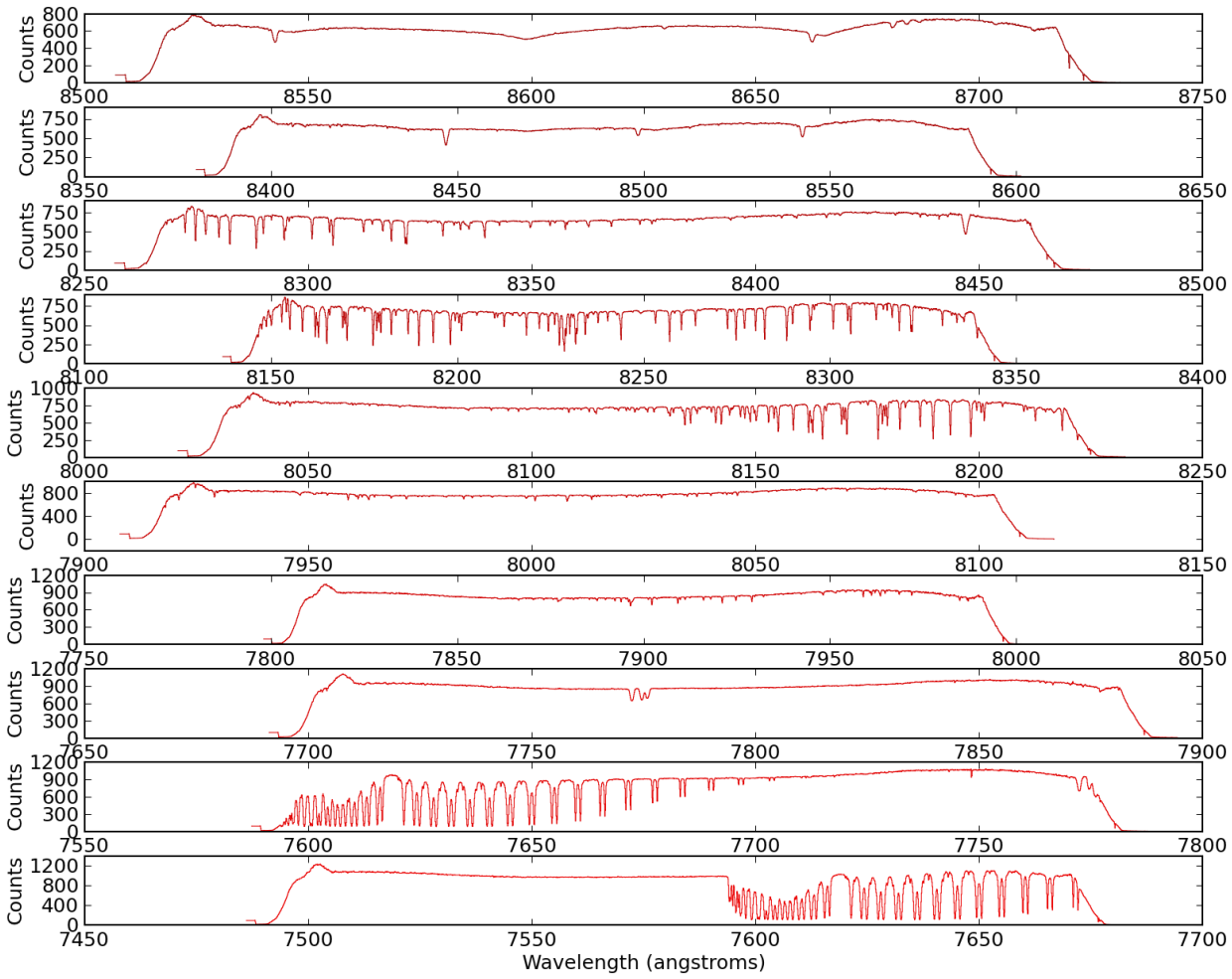
2.6.5 Optical Plotting - Echelle spectrum of Vega (in color!)

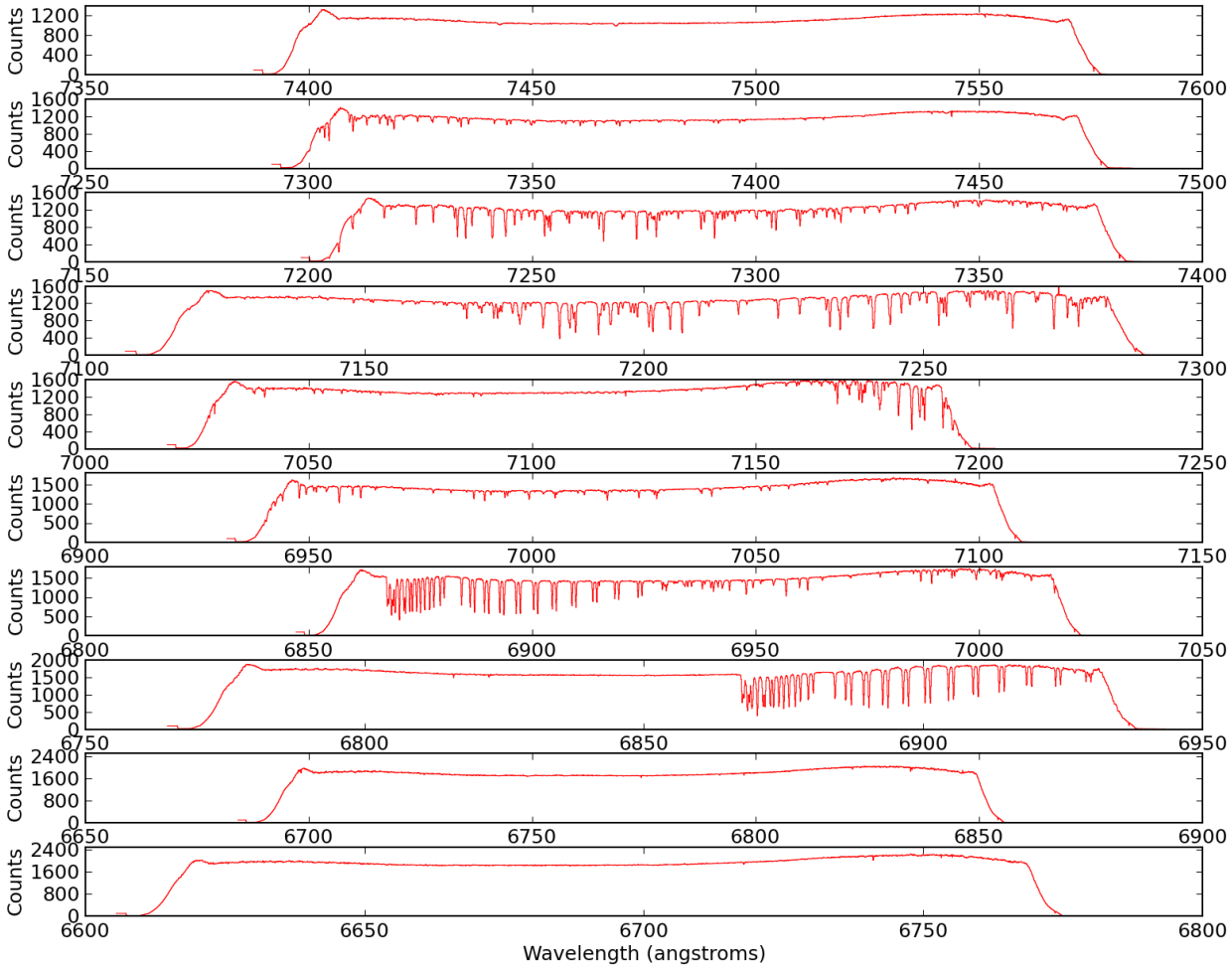
```
import pyspeckit
from pylab import *
import wav2rgb

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

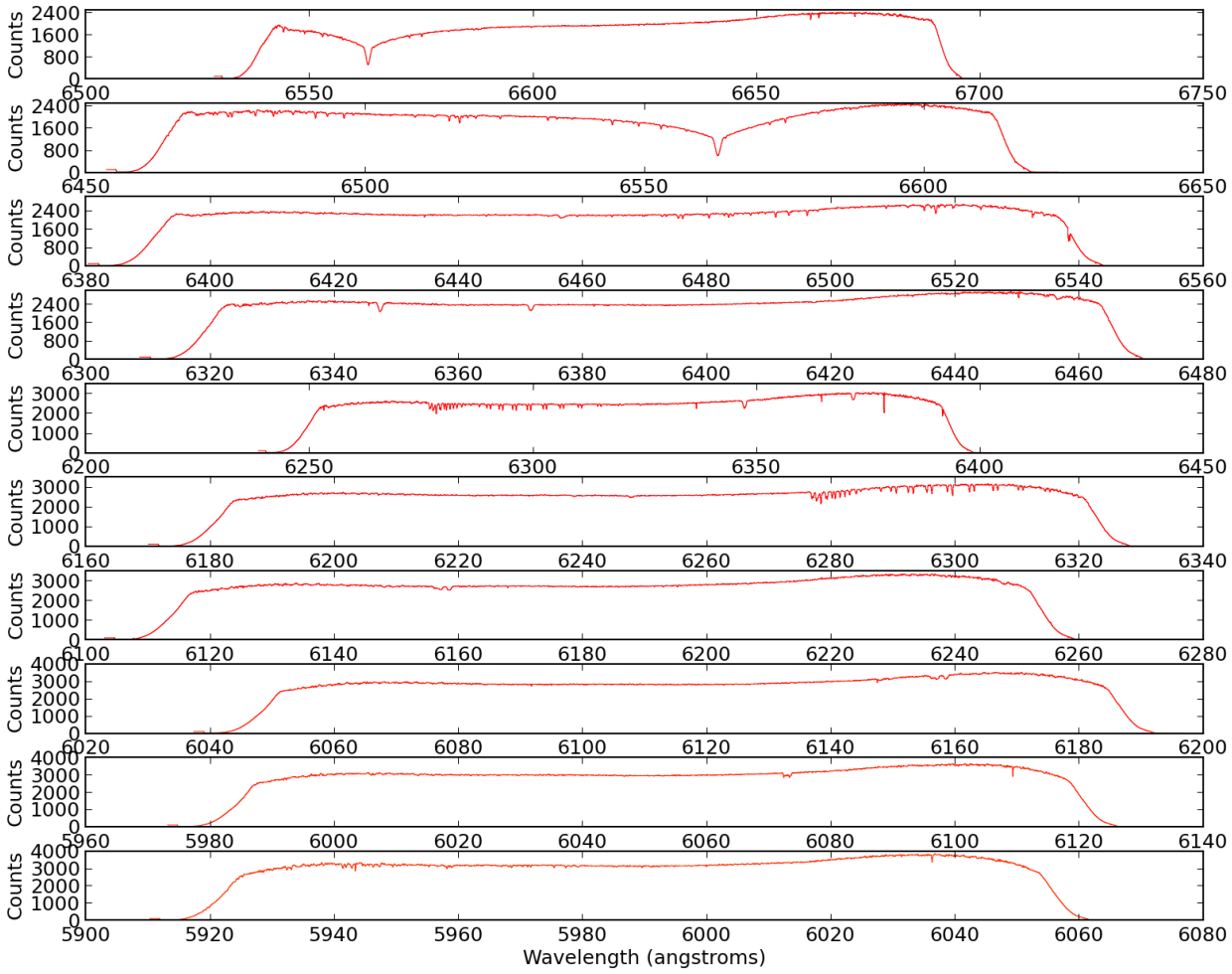
```
for spec in speclist:
    spec.units="Counts"

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()
```

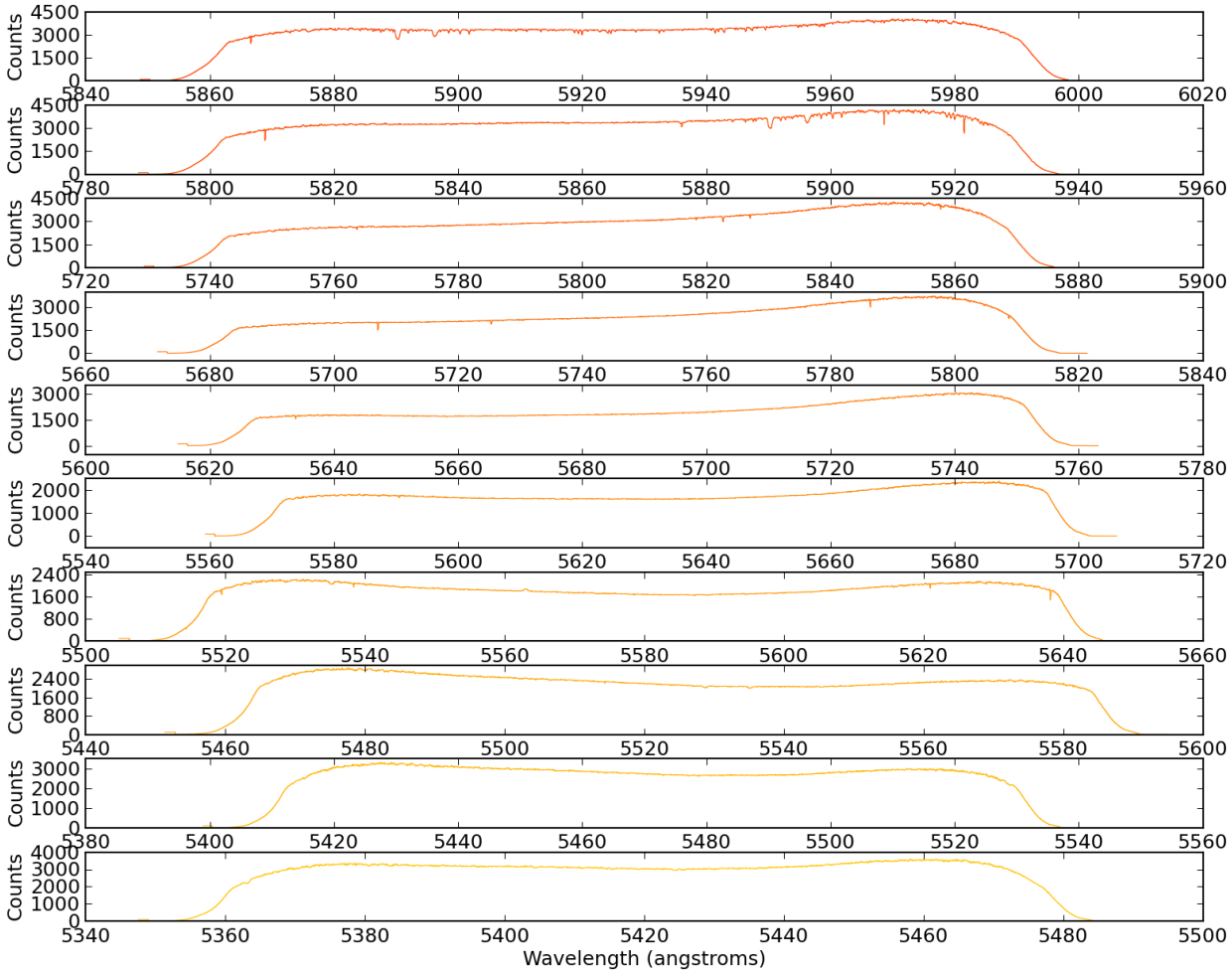


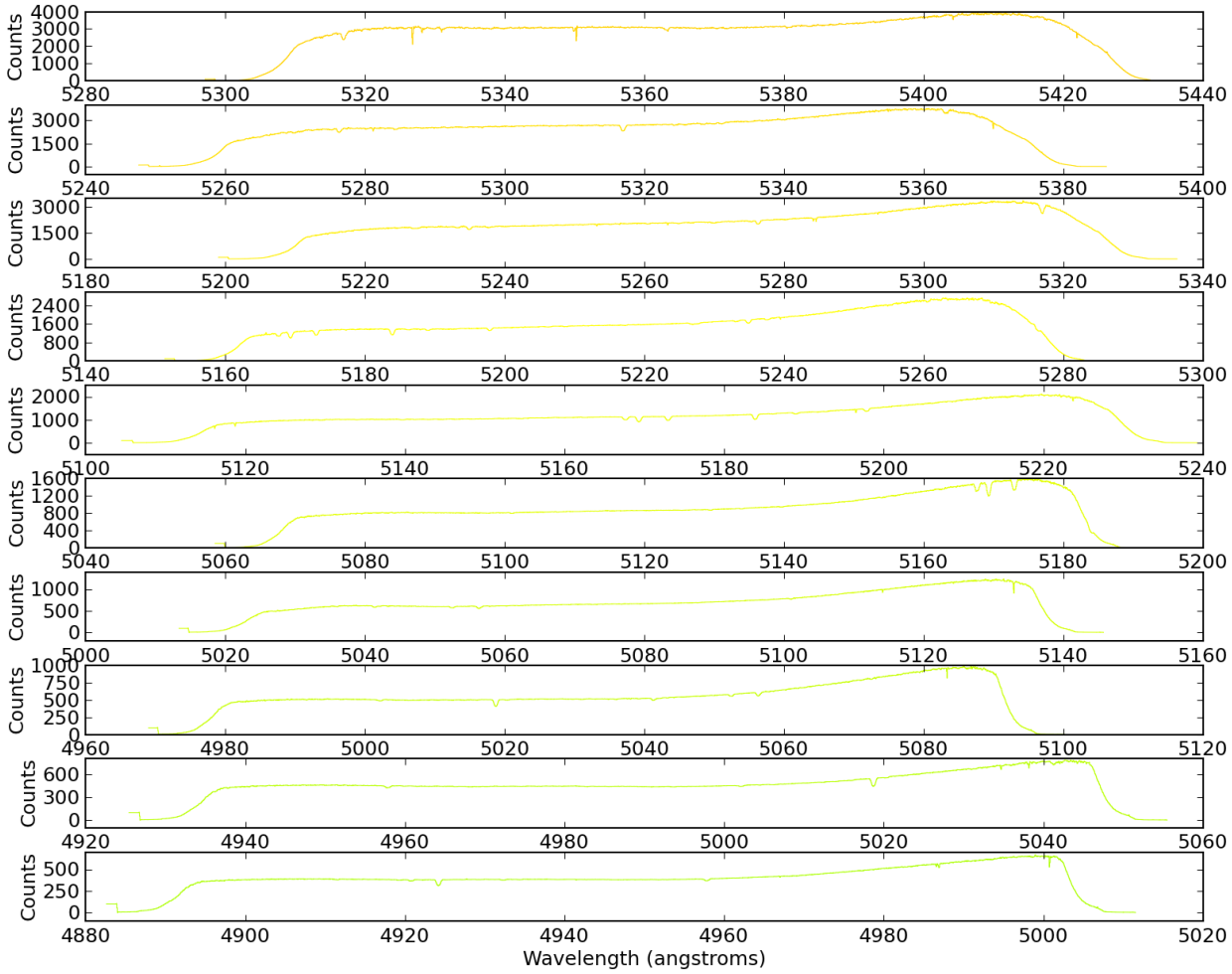

Counts
Wavelength (angstroms)

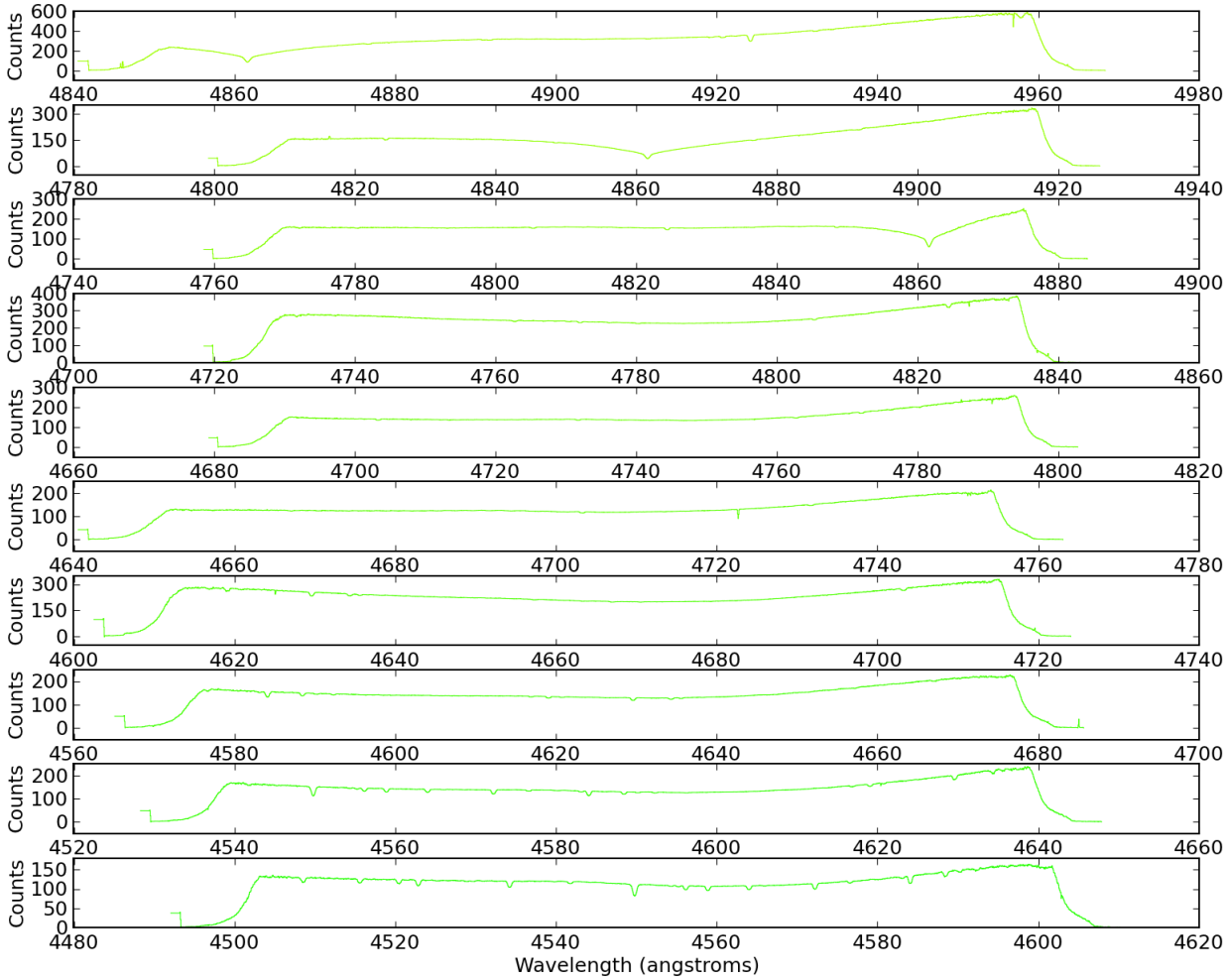

Counts
Wavelength (angstroms)

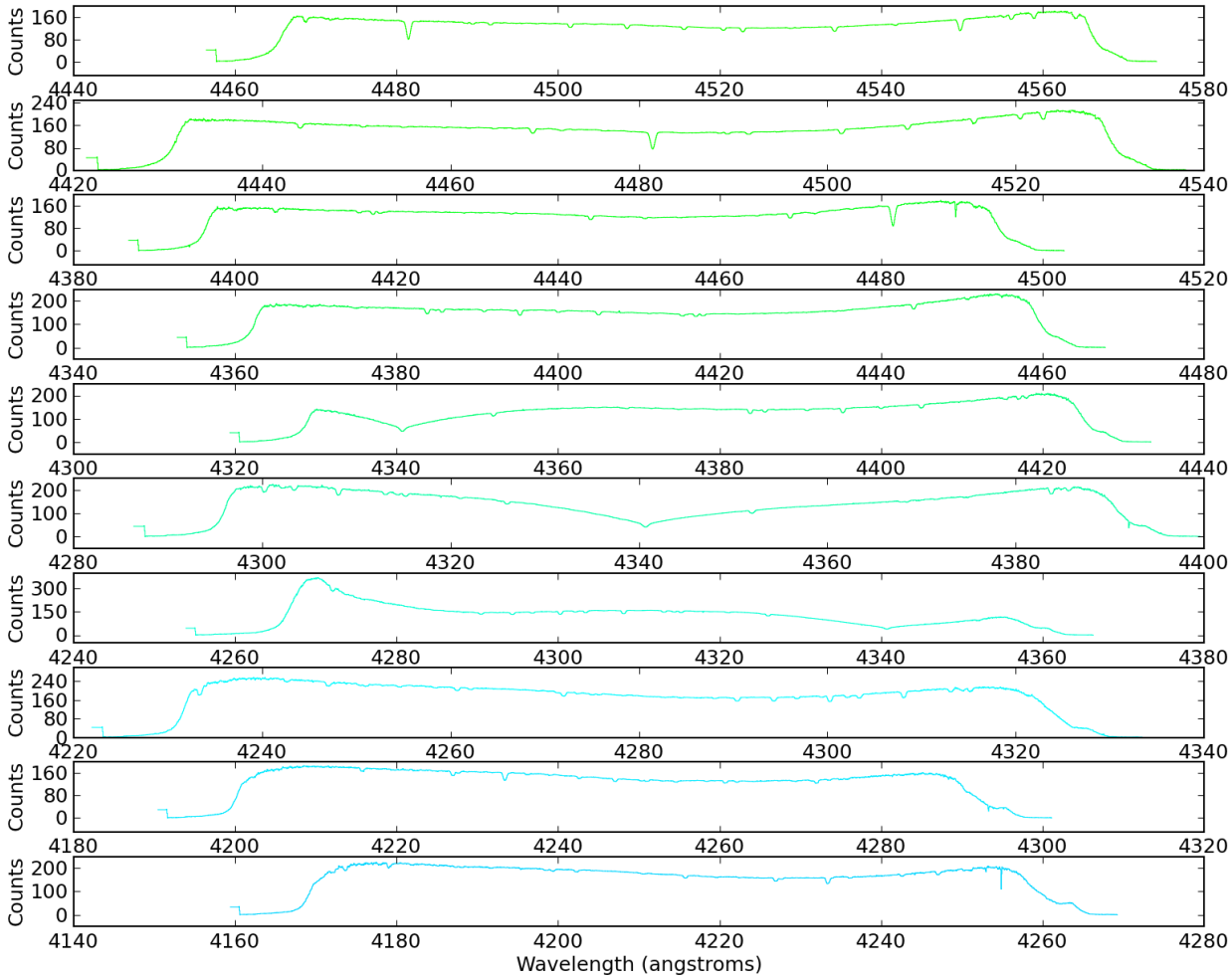

Counts
Wavelength (angstroms)

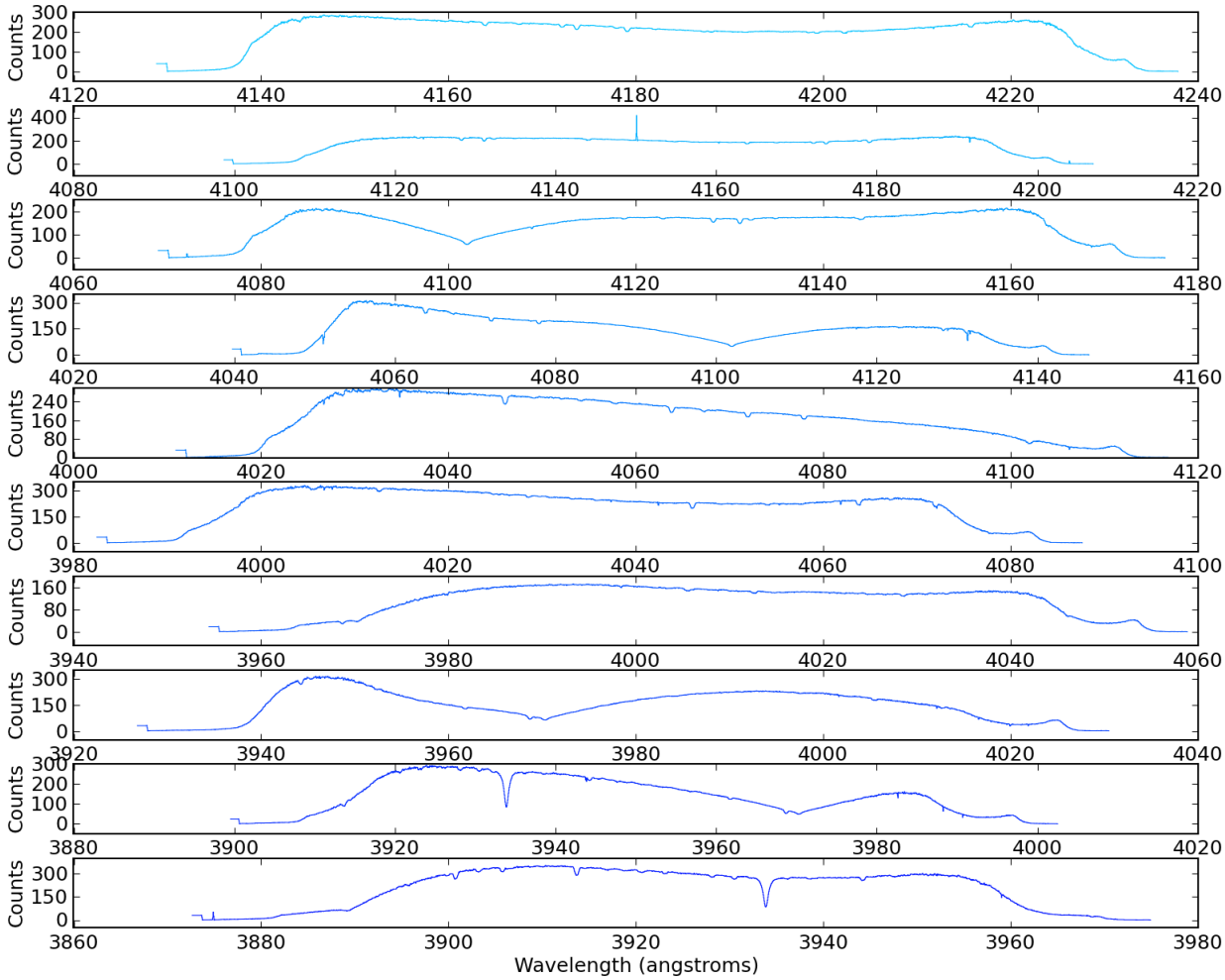

Counts
Wavelength (angstroms)

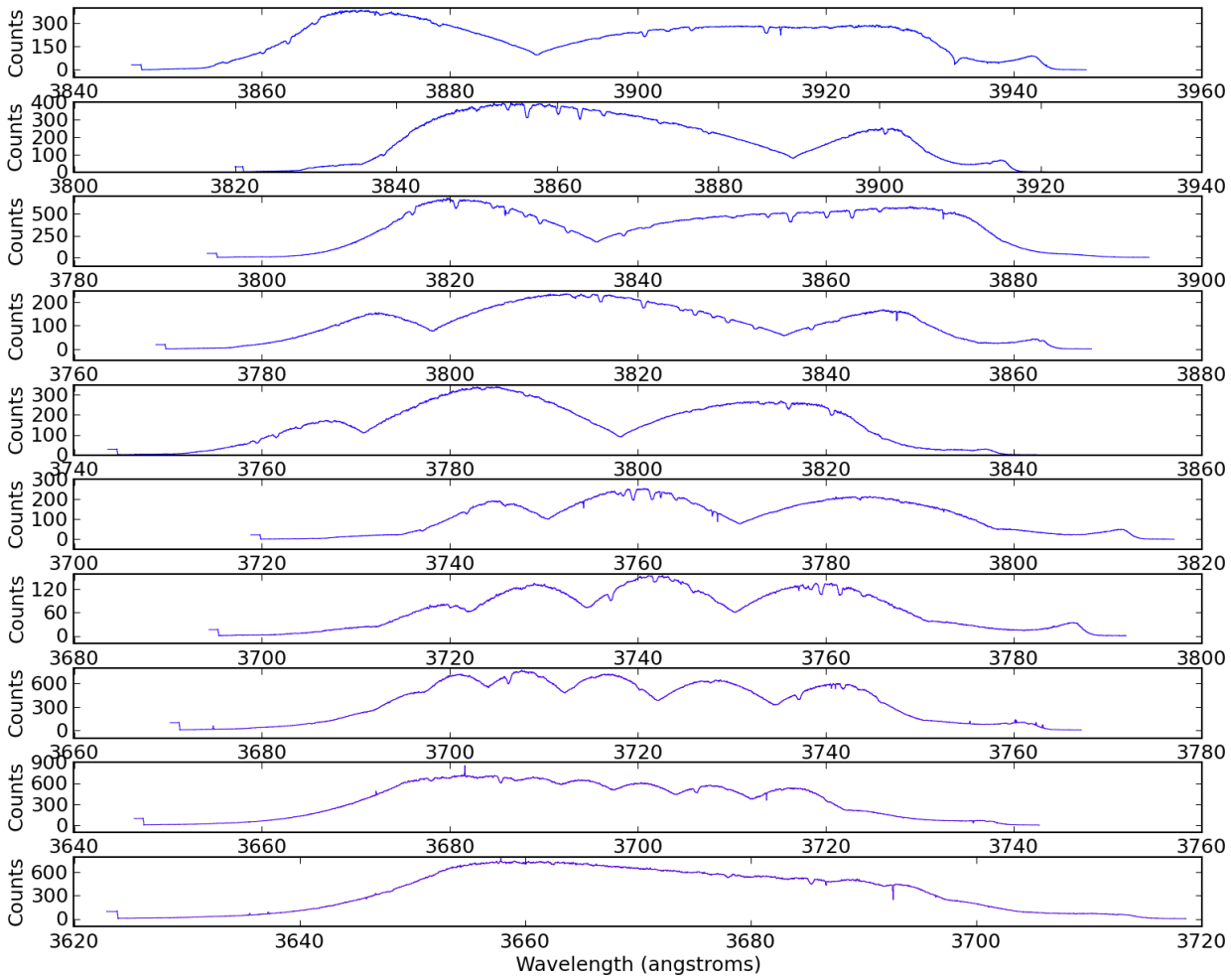

Counts
Wavelength (angstroms)


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)

BIBLIOGRAPHY

[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

PYTHON MODULE INDEX

p

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

X