SpecViz Documentation

Release 0.1.0a1

STScI

May 18, 2016

Contents

SpecViz is a tool for 1-D spectral visualization and analysis of astronomical spectrograms. It is written in Python thus can be run anywhere Python is supported (see *Installation*). SpecViz is capable of reading data from FITS and ASCII tables (see *Custom Loaders*).

Once ingested, data can be plotted and examined with a large selection of custom settings. SpecViz supports instrument-specific data quality handling, flexible spectral units conversions, custom plotting attributes, plot annotations, tiled plots, etc.

A spectral feature quick measurement tool enables the user, with a few mouse actions, to perform and record measurements on selected spectral features.

SpecViz can be used to build wide-band SEDs, overploting or combining data from the same astronomical source taken with different instruments and/or spectral bands. Data can be further processed with averaging, splicing, detrending, and Fourier filtering tools.

SpecViz has a spectral model fitting capability that enables the user to work with multi-component models in a number of ways, and fit models to data. For more details, see *Model Fitting*.

Support exists for overplotting and interactively renormalize data from spectral templates.

SpecViz can overplot spectral line identifications taken from a variety of line lists.

Note: Some features are in development and not yet available.

CHAPTER 1

Installation and Setup

1.1 Installation

SpecViz is distributed through the Anaconda package manager. Specifically, it lives within Space Telescope Science Institute's AstroConda channel.

If you do not have Anaconda, please follow the instructions here to install it, or scroll down for manual installation of SpecViz.

1.1.1 Install via Anaconda

If you have AstroConda setup, then all you have to do to install SpecViz is simply type the following at any Bash terminal prompt:

```
$ conda install specviz
```

If you do not have AstroConda setup, then you can install SpecViz by specifying the channel in your install command:

```
$ conda install --channel http://ssb.stsci.edu/astroconda specviz
```

At this point, you're done! You can launch SpecViz by typing the following at any terminal:

```
$ specviz
```

PyQt5 version (optional)

While the PyQt4 version of SpecViz is currently the default distributed, it is recommended that you upgrade to using the PyQt5 version if you know that you do not have any system conflicts. Note that this is entirely optional, but encouraged due to the fact that Qt4 development and support has ended.

```
$ conda install --channel https://conda.anaconda.org/spyder-ide pyqt5
$ conda install --channel https://conda.anaconda.org/nmearl pyqt5
```

SpecViz can then be launched via the command line:

```
$ specviz
```

1.1.2 Install via source

SpecViz can also be installed manually using the source code and requires the following dependencies to be installed on your system. Most of these will be handled automatically by the setup functions, with the exception of PyQt/PySide.

- Python 3 (recommended) or Python 2
- PyQt5 (recommended), PyQt4, or PySide
- Astropy
- Numpy
- Scipy
- PyQtGraph

Installing PyQt/PySide

The easiest way to install PyQt/PySide is through some package manager. Please keep in mind that PyQt5 is the recommended PyQt implementation as Qt4 development and support has ended.

Below are instructions for installing using *either* Homebrew *or* Anaconda.

PyQt5

Homebrew Install using Homebrew for Qt5.

Anaconda Installing PyQt5 with Anaconda will require installing from the Spyder-IDE channel as it is not currently a core package (but they're working on it).

Further, Anaconda will panic if you have both PyQt4 and PyQt5 installed in the same environment. To work around this, it is strongly suggested you simply create a new virtual environment and install PyQt5 there:

```
$ conda create -n sviz_env python=3.5
$ source activate sviz_env
$ conda install --channel https://conda.anaconda.org/spyder-ide pyqt5
```

Note: PyQt5 **does not** require Python 3. If you wish, you can create your virtual environment using Python 2 by specifying the version as shown above (e.g. `python=2.7`).

PyQt4

Homebrew Install using Homebrew for Qt4.

Anaconda Install using Anaconda:

```
$ conda install pyqt
```

Installing

Clone the SpecViz repository somewhere on your system, and install locally using `pip`. If you are using an Anaconda virtual environment, please be sure to activate it first before installing: `$ source activate sviz_env`.

```
$ git clone https://github.com/spacetelescope/specviz.git
$ cd specviz
$ git checkout tags/v0.1.1rc3
$ pip install -r requirements.txt
```

Note: This uses the `pip` installation system, so please note that

1. You need to have `pip` installed (included in most Python installations).
2. You do **not** need to run `python setup.py install`.
3. You do **not** need to install the dependencies by hand (except for PyQt).

Likewise, the `pip` command will use your default Python to install. You can specify by using `pip2` or `pip3`, if you're not using a virtual environment.

1.1.3 Known Issues

On a Mac with Qt5, depending on exactly how you have set up Anaconda, you might see the following error after following the above instructions:

```
This application failed to start because it could not find or load the Qt platform plugin "cocoa".

Reinstalling the application may fix this problem.
```

If you see this message, you have encountered an incompatibility between Anaconda's packaging of Qt4 and Qt5. The workaround is to uninstall Qt4 with the following command:

```
$ conda uninstall pyqt qt
```

and SpecViz should now happily run.

1.2 Launching SpecViz

Once you've installed SpecViz, you can launch it via the command line:

```
$ specviz
```

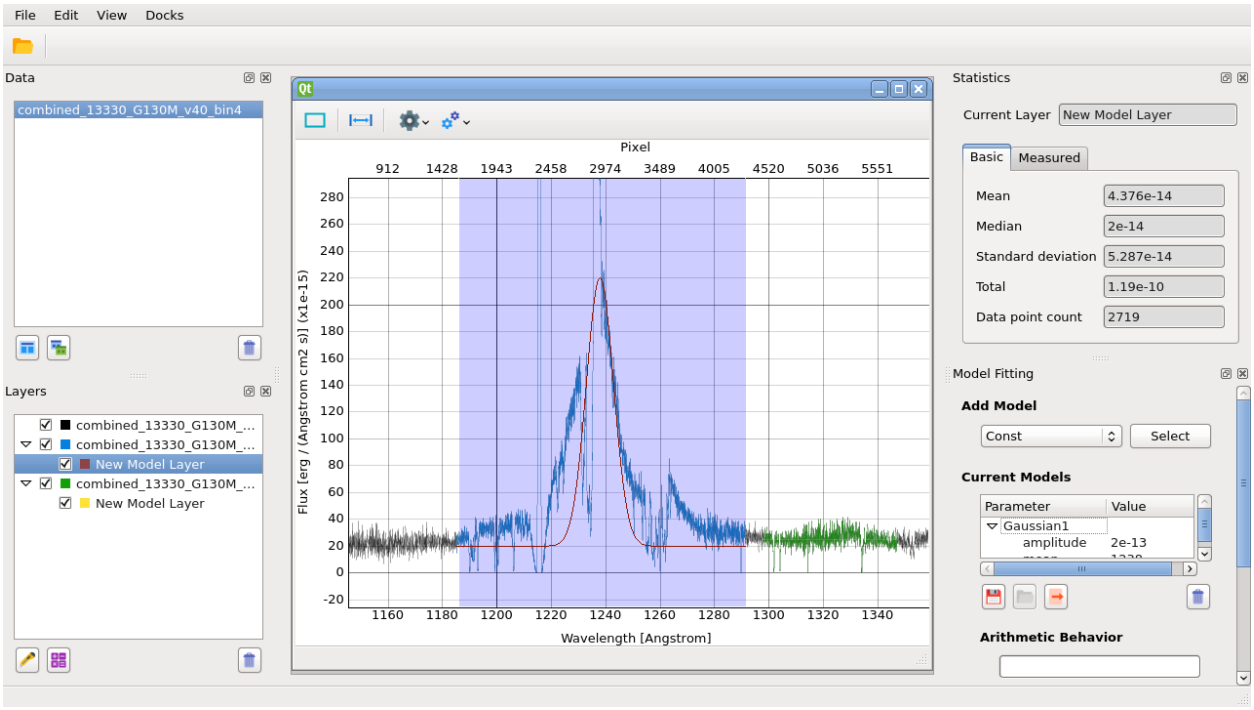
If you only wish to inspect a single FITS or ASCII file using the default *Custom Loaders* file formatting, you can also pass in the filename as a command line argument, as follows:

```
$ specviz filename
```

CHAPTER 2

Using SpecViz

2.1 Viewer



Note: Some buttons and menu options are not functional yet, as this is a work in progress.

To open a file:

1. Click on the folder icon on top left or select `File -> Open` from menu.
2. Select the desired filename and data type in the file dialog and click "Open".
3. The file will be listed under "Data" (top half of left panel).
4. To remove the file from "Data", click on "trash can" icon under "Data".

Once a spectrum is loaded, remember to **click on the plot icon** (left-most button right above "Layers" at the center of the left panel) to display the spectrum. It will be plotted in the center display window. Some basic statistics are shown on top right. Its data layer will be listed under "Layers" (bottom half of left panel).

If multiple files are opened, each file will have its own plot (overplotting is not yet supported). To work on the file of interest, click on its plot window to bring it in focus.

To adjust plot display:

1. If your mouse has center wheel, the wheel can be used to zoom in and out while the cursor is over the plot.
2. Click on the "big wheel" icon on top of the plot to change top axis display or displayed units.
3. Left click on the plot to bring up a context menu.
4. Select "View All" to view all available data (if zoomed in).
5. Select "X Axis" or "Y Axis" to adjust respective axis behaviors.
6. Select "Mouse Mode" to toggle between 3-button and 1-button mouse.
7. More plot options are available under "Plot Options".
8. Select "Export..." to save plot out as an image.

To select a region of interest (ROI):

1. Select the layer from "Layers" list (if you have multiple).
2. Click on the "rectangle" icon on the top left of the plot. An adjustable rectangle will appear on plot display.
3. Click and drag the edges to adjust dispersion coverage. As the region changes, basic statistics under "Statistic" on top right of the viewer will also update accordingly.
4. Click inside the rectangle and drag to reposition it without resizing.
5. Click the left-most "knife" button under "Layers" (bottom left of the left panel) to create a new layer slice from the ROI.
6. To remove a layer slice, select the layer and click on "trash can" button under "Layers".
7. To remove ROI, left click while it is highlighted and select "Remove ROI" from the context menu.

Note: Layer arithmetic (the "calculator" button under "Layers") is work in progress.

To measure equivalent width and related properties:

1. Select the layer from "Layers" list (if you have multiple).
2. Click on the "rectangle with double arrow" icon that is next to the ROI icon on the top left of the plot. Three regions will appear and each can be adjusted like an ROI.
3. Position the orange region over the emission or absorption line of interest and adjust accordingly.
4. Position each of the green regions over two different continuum areas, one on each side of the emission/absorption line.
5. As you adjust the regions, values for equivalent width etc. will be updated under the "Measured" tab under "Statistics" on top right of the viewer.

To fit a model to the selected ROI, see *Model Fitting*.

To quit SpecViz, select `File -> Exit` from menu.

2.2 Model Fitting

SpecViz utilizes Astropy Models and Fitting to fit models to its spectra. For example, you can fit one model to the continuum, another to an emission line of interest, and yet another to an absorption line.

Currently, the following models are available:

SpecViz Model Name	Astropy Model Class
BlackBody	`blackbody_(lambda, nu)`
BrokenPowerLaw	`BrokenPowerLaw1D`
Const	`Const1D`
ExponentialCutoffPowerLaw	`ExponentialCutoffPowerLaw1D`
Gaussian	`Gaussian1D`
GaussianAbsorption	`GaussianAbsorption1D`
Linear	`Linear1D`
LogParabola	`LogParabola1D`
Lorentz	`Lorentz1D`
MexicanHat	`MexicanHat1D`
Trapezoid	`Trapezoid1D`
PowerLaw	`PowerLaw1D`
Redshift	`Redshift`
Scale	`Scale`
Shift	`Shift`
Sine	`Sine1D`
Spline	`UnivariateSpline`
Voigt	`Voigt1D`

The models can be fitted with the following fitters:

SpecViz Fitter Name	Astropy Fitter Class
Levenberg-Marquardt	`LevMarLSQFitter`
Simplex	`SimplexLSQFitter`
SLSQP	`SLSQPLSQFitter`

To add a model:

1. Select the desired layer from `Layers` (left panel). For example, you can choose the layer containing your emission or absorption line. See *Viewer* on how to create a layer for ROI.
2. Select the desired model name from the `Add Model` drop-down box and click `Select` to add it to `Current Models`.
3. If desired, repeat the above step to add additional models.
4. Scroll down (if needed) and click `Create Layer`.
5. A new model layer will be created under `Layers` (left panel) and it is attached to the selected data layer.

To fine-tune model parameters:

1. Select the model layer under `Layers` (left panel) that contains the desired model.
2. If desired, double-click on the model name to rename it. When you see a blinking cursor, enter its new name and press "Enter".
3. Expand the model listing under `Current Models` on the right of the viewer.
4. Double-click on the desired model parameter value in the listing. When you see a blinking cursor, enter the new value and press `Enter`.
5. Scroll down (if needed) and click `Update Layer`.

To fit a model:

1. Select the model layer under `Layers` that contains the model(s) you wish to fit to your data.
2. Select the desired fitter from `Fitting Routine` using its drop-down menu.
3. Click `Perform Fit`. This may take up to a few seconds, depending on the complexity of the fit.
4. The associated model parameters will be adjusted accordingly.

The `Arithmetic Behavior` text box is used to define the relationship between different models for the same layer. If nothing is defined, the default is to add all the models together. To describe a non-default model relationship, enter the model names and math operators, as shown in the examples below and then press `Create Layer` or `Update Layer` to produce the compound model:

```
Linear1 + Gaussian1
```

```
Linear1 * Gaussian1
```

```
Gaussian1 - Gaussian2
```

The entity that results from lumping together all the models, and combining them either using the arithmetic behavior expression, or just adding them all together, is called a "compound model".

2.2.1 Model names

When added to the `Current Models` list, a model will receive a default name that is generated from the model type (as listed in the drop down model selector) plus a running numerical suffix.

These names can be re-defined by clicking on the default name and typing a new name. Note that re-defining names will require that any eventual expression in the `Arithmetic Behavior` text box should be edited accordingly.

For now, we are limited to only alphanumeric characters (and no white spaces) when re-naming models.

2.2.2 Spline model

Note that the Spline model is of an intrinsically different nature than the other models included in the drop down list of models. The Spline model, when added to a pre-existing list of models, or when added by itself to an empty list, will immediately be fitted to the data within the currently defined Regions Of Interest. That is, being a linear model, there is no need to iterate in search of a "best fit" spline. It is just computed once and for all, and kept as part of the compound model that is built from the models in the list and the arithmetic behavior expression.

This implies that, to change the regions of interest that define the spline, one has no other way than removing the spline from the list of models. Then, redefine the regions of interest, and add a new spline to the list. To change a spline parameter, there is no need do discard the spline. Just do it in the same way as with other models: just type in the new value for the parameter and click on `Update Layer`.

Subsequently, when the fitter iterates the compound model in search of a best solution, the spline model will act as a constant. That is, it will be used to compute the global result of the compound model, but its parameters won't be accessed, and varied, by the fitter. Thus, the spline parameters are not fitted, they are just a convenient mechanism that enables user access to the parameter's values.

The documentation for the spline model cane be seen here:

http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.interpolate.UnivariateSpline.html

Note that SpecViz provides access, at this point, to just two of the parameters in the scipy implementation of th spline function. Pay special attention to the `smooth` parameter. SpecViz initializes it to a 'best guess'

(`len(wavelength)`). Too small of a value in here may cause the spline to enter an infinite loop. Change the `smooth` value with care, trying to stay close to the default value.

Note: Model arithmetic is a work in progress.

Saving and Exporting Models to File

Selecting a model layer under "Layers" will enable the *Save* (the "floppy disk" icon) and *Export* (the "out the door" icon) buttons under "Current Models" on the right of the viewer. Saving a model to a file will enable SpecViz to read back that model into a new model layer. Exporting a model to a file wil create a Python script in a `.py` file. This file can be directly imported by Python in a command-line session.

Click on either button to get a file dialog window. Type in a file name. If this file name does not end with the correct suffix, the suffix will automatically be appended. Click "Save", or just the Return/Enter key. The correct suffix for saved and exported files are `.yaml` and `.py`, respectively.

2.2.3 Save and Load

Saving the model to a file works in the same way as *Export*. The difference is that a saved model can be later read back into SpecViz via the "Load" button (the "folder" icon), also under "Current Models".

For the "Load" button to be enabled, a data (spectrum) layer (not a model layer) must be selected under "Layers" listing. The selected `.yaml` model file will generate a model that will be attached to a new model layer associated under the selected data layer.

The file is writen using the YAML format. Being a plain text file with a self-explanatory structure, it can be edited at will by the user, e.g., to add bounds, fixed flags, and ties to the model parameters. Note that these extra, user-defined attributes, won't be accessible from SpecViz's user interface. They will however, be accessible by the fitter when a fit is run on the model. They will also be written ou correctly, either when saving or exporting the model.

Note: YAML format for saved models and usage of advanced features like bounds and fixed flags are work in progress.

2.2.4 Export

This will save the model in the currently selected model layer to a file that can be directly imported by Python. The file is just a plain text file with the model expressed recorded as a Python expression. The model is associated to a variable named `'model1'`.

The following example uses the `'test3.py'` file name, and a model comprised of a constant and a gaussian:

```
>>> import test3
>>> test3
<module 'test3' from '/my/saved/models/test3.py'>
>>> test3.model1
<CompoundModel0(amplitude_0=0.297160787184, amplitude_1=2.25396100263, mean_1=15117.1710847, stddev_1
>>> print(test3.model1)
Model: CompoundModel0
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Parameters:
```

```
 amplitude_0    amplitude_1      mean_1       stddev_1
-------------- ------------- ------------- -------------
0.297160787184 2.25396100263 15117.1710847 948.493577186
```

The file can be edited at will by the user, e.g., to add bounds, fixed flags, and ties to the model parameters.

Note: Security issues importing model this way into Python and usage of advanced features like bounds and fixed flags are work in progress.

2.3 Custom Loaders

SpecViz utilizes Astropy I/O registry and YAML data serialization language to enable flexible support for a variety of data formats both in FITS and ASCII.

When `specviz` is called with a filename as argument (see *Launching SpecViz*), the default formats below are assumed based on file extension:

- `.fits` or `.mits` – *Generic FITS Loader*
- `.txt` or `.dat` – *ASCII Loader*

By examining the YAML definitions in the following sub-sections and their associated example data files, you will be able to create your own YAML file to define most custom data formats (see *Creating a Custom Loader*). In addition, there are also other custom loaders that come with SpecViz that follow the same rules but are modified to load JWST DADF test data, which you can also use as a reference.

While using YAML is very flexible, it is also very sensitive to slight changes in your file format. For instance, if you have two files from the same instrument but processed differently (say, one was extracted using IRAF and another one using your own IDL program), they might have different formats and will need separate YAML files.

2.3.1 Generic FITS Loader

```
--- !CustomLoader
name: Generic Fits
extension: [fits, mits]
wcs:
  hdu: 0
data:
  hdu: 1
  col: 0
uncertainty:
  hdu: 1
  col: 1
  type: 'std'
meta:
  author: Nicholas Earl
```

This is generic.yaml, which is a built-in YAML definition for a "generic" FITS file.

```
--- !CustomLoader
```

The first line states that this YAML file defines our custom loader. This is always the same no matter what kind of format you are defining.

```
name: Generic Fits
extension: [fits, mits]
```

These two lines define the format name and accepted extensions, respectively. In SpecViz GUI, this will translate to "Generic Fits (*.fits *.mits)" in the file type drop-down menu.

```
wcs:
  hdu: 0
```

This instructs the loader to look for WCS information in the `PRIMARY` (Extension 0) header. SpecViz also uses this header for other look-ups, e.g., flux unit from `BUNIT` (see below). Therefore, even if there are no WCS information in your file, you always define this block and point the HDU value to the `PRIMARY` header. If WCS information are available and supported by Astropy WCS, they will be used to establish dispersion values and unit.

In the absence of WCS or the presence of explicit dispersion column, an additional `dispersion:` block (not shown) can be defined similarly as `data:` (see below). Unlike flux, `TNULL` masking is ignored. If its column does not have `TUNIT` and `unit: 'unitname'` is defined, SpecViz will fall back to WCS unit. If all these unit look-ups failed, it defaults to null unit. In the presence of `unit:` definition, it overrides both `TUNIT` and WCS.

```
data:
  hdu: 1
  col: 0
```

This instructs the loader to look for flux values (data) in Extension 1, the first column (column index starts from 0). If the column complies to FITS standards (see Astropy FITS Table), flux unit (inferred from `TUNIT`) and data mask (inferred from `TNULL`) are also extracted from the same column.

If `TUNIT` is not defined, loader will look for `unit: 'unitname'` definition within this block, the unit name must be one that is accepted by Astropy Units (case sensitive). If that is undefined as well, flux unit is extracted from `BUNIT` keyword in the same header that contains WCS information (see `wcs:` block for details). If all unit look-ups failed, flux unit is assumed to be erg $^{-1}$ cm^{-2} s^{-1}.

Note that `TNULL` is not the same as DQ arrays, which can be similarly defined with `mask:` block (not shown). If both `TNULL` and DQ are defined, the masks will be combined.

```
uncertainty:
  hdu: 1
  col: 1
  type: 'std'
```

This instructs the loader to look for flux uncertainty values in Extension 1, the second column. Uncertainty type `'std'` states that the values are standard deviation (as opposed to inverse variance, `'ivar'`). Unlike flux data, its `TNULL` masking is ignored and `unit:` tag is not supported. If `TUNIT` is present, loader will attempt to convert the values to flux unit first. Otherwise, its unit is assumed to be the same as flux unit. If inverse variance is given, square-root is applied to the inversed values before being converted to `StdDevUncertainty`.

```
meta:
  author: Nicholas Earl
```

The `meta:` block can contain any metadata tags you wish to include. They do not affect how SpecViz works. In this example, the `author:` tag identifies Nicholas Earl as the origin author of this YAML file.

2.3.2 ASCII Loader

```
--- !CustomLoader
name: ASCII
extension: [txt, dat]
dispersion:
```

```
  col: 0
  unit: 'Angstrom'
data:
  col: 1
  unit: 'erg / (Angstrom cm2 s)'
uncertainty:
  col: 2
  type: 'std'
meta:
  author: STScI
```

This is ascii.yaml, which is a built-in YAML definition for a "generic" ASCII file.

```
--- !CustomLoader
```

The first line states that this YAML file defines our custom loader. This is always the same no matter what kind of format you are defining.

```
name: ASCII
extension: [txt, dat]
```

These two lines define the format name and accepted extensions, respectively. In SpecViz GUI, this will translate to "ASCII (*.txt *.dat)" in the file type drop-down menu. All ASCII files must comply to Astropy ASCII Table standards.

Any header comments with `KEY = VALUE` format will be extracted as header metadata information (currently not used by SpecViz).

```
dispersion:
  col: 0
  unit: 'Angstrom'
```

Unlike *Generic FITS Loader*, ASCII table does not contain WCS. Therefore, the `dispersion:` block is necessary to define the actual dispersion (e.g., wavelength) values. This instructs the loader to look for dispersion values in the first column (column index starts from 0). Its unit, if not defined in the table itself (e.g., via IPAC table format), will be taken from the `unit:` tag. The given unit name must be one that is accepted by Astropy Units (case sensitive). If unit is defined in both table and tag, the latter is ignore. If unit is not defined anywhere, it defaults to Angstrom.

```
data:
  col: 1
  unit: 'erg / (Angstrom cm2 s)'
```

This instructs the loader to look for flux values (data) in the second column. Flux unit handling is similar to dispersion unit (see above), except that the default unit would be erg $^{-1}$ cm^{-2} s^{-1} if undefined.

If there is an associated DQ column, it can be extracted in a similar fashion using a `mask:` block specifying the column index (unit is not applicable). Like *Generic FITS Loader*, zero mask values signify good data.

```
uncertainty:
  col: 2
  type: 'std'
```

This instructs the loader to look for flux uncertainty values in the third column. Uncertainty type `'std'` states that the values are standard deviation (as opposed to inverse variance, `'ivar'`). Its unit must be the same as flux unit. If inverse variance is given, square-root is applied to the inversed values before being converted to `StdDevUncertainty`.

```
meta:
  author: STScI
```

The `meta:` block can contain any metadata tags you wish to include. They do not affect how SpecViz works. In this example, the `author:` tag identifies STScI as the origin author of this YAML file.

2.3.3 Creating a Custom Loader

In addition to loaders that come pre-packaged with the software, SpecViz also looks for custom loaders that you created and saved in your `~/.specviz` directory, which can be created with the following Unix command:

```
$ mkdir ~/.specviz
```

To create your own loader, you can use either *Generic FITS Loader* or *ASCII Loader* as a template. Your YAML file can have any name of your choosing but must end with a `.yaml` extension.

In this section, we use a MOSFIRE spectrum named spec1d.gds1312_H0.003.emp26177.fits as an example of a custom FITS format for which we must create our own custom YAML definition file from the *Generic FITS Loader* template.

First, we inspect the file format that we have, as follow.

```
>>> from astropy.io import fits
>>> pf = fits.open('spec1d.gds1312_H0.003.emp26177.fits')
>>> pf.info()
Filename: spec1d.gds1312_H0.003.emp26177.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       4   ()
1                BinTableHDU     23   1R x 3C      [2287E, 2287E, 2287E]
```

This opens the FITS file and prints out the overall file structure. From this, it is obvious that the table is in Extension 1.

```
>>> pf[0].header
SIMPLE  =                    T /Dummy Created by MWRFITS v1.4a
BITPIX  =                    8 /Dummy primary header created by MWRFITS
NAXIS   =                    0 /No data is associated with this header
EXTEND  =                    T /Extensions may (will!) be present
>>> pf[1].header
XTENSION= 'BINTABLE'           /Binary table written by MWRFITS v1.4a
BITPIX  =                    8 /Required value
...
TFORM3  = '2287E   '           /
```

This prints all the headers and we find no WCS information in either of the extensions.

```
>>> from astropy.table import Table
>>> tab = Table.read(pf[1], format='fits')
>>> print(tab)
FLUX [2287]      LAMBDA [2287]      IVAR [2287]
---------------- ----------------- ---------------
0.0 .. 0.0989667 14500.0 .. 18223.8 1e-06 .. 422.54
```

This shows that there are three columns in the table in Extension 1, namely flux, wavelength, and inverse variance. The table has 2287 rows. Knowing the wavelength regime that the instrument is sensitive to and looking at the wavelength values, we can safely assume that the wavelength unit is Angstrom.

```
>>> from astropy import units as u
>>> u.electron / u.s / u.pix
Unit("electron / (pix s)")
```

However, the flux unit is not defined anywhere and cannot be easily inferred. So, let's just say that we already know the unit to be electrons/s/pix. The code above shows us how Astropy can ingest the flux unit that we want.

```
--- !CustomLoader
name: Keck/MOSFIRE Fits
```

```
extension: [fits, mits]
wcs:
  hdu: 0
dispersion:
  hdu: 1
  col: 1
  unit: 'Angstrom'
data:
  hdu: 1
  col: 0
  unit: 'electron / (pix s)'
uncertainty:
  hdu: 1
  col: 2
  type: 'ivar'
meta:
  author: STScI
```

Now that we have the format figured out, it is time to write our own YAML file for it. We will name it keck_mosfire.yaml.

```
--- !CustomLoader
```

The first line states that this YAML file defines our custom loader.

```
name: Keck/MOSFIRE Fits
extension: [fits, mits]
```

These two lines define the format name and accepted extensions, respectively. We will keep the extensions from our FITS template but change the name to identify our new format. In SpecViz GUI, this will translate to "Keck/MOSFIRE Fits (*.fits *.mits)" in the file type drop-down menu.

```
wcs:
  hdu: 0
```

We do not have WCS nor `BUNIT` defined, so we will simply leave this the same as our template.

```
dispersion:
  hdu: 1
  col: 1
  unit: 'Angstrom'
```

This instructs the loader to look for our wavelength values in Extension 1, the second column. We explicitly set its unit to Angstrom.

```
data:
  hdu: 1
  col: 0
  unit: 'electron / (pix s)'
```

This instructs the loader to look for our flux values in Extension 1, the first column, like the template. However, we also explicitly set its unit to electrons/s/pix by providing the appropriate Astropy unit name.

```
uncertainty:
  hdu: 1
  col: 2
  type: 'ivar'
```

This instructs the loader to look for flux uncertainty values in Extension 1, the third column. Unlike the template, we define it as inverse variance.

```
meta:
  author: STScI
```

Since this does not affect how SpecViz works, we do the lazy thing here by leaving it the same as our template.

Once you are done writing your YAML file, be sure to save it in `~/.specviz`. Next, start SpecViz as usual. Now, in its open file dialog, you will see your new format listed in the file-type drop-down menu.

CHAPTER 3

References/API

3.1 API

This section documents the underlying code base to aid SpecViz developers.

3.1.1 data

This module handles spectrum data objects.

class specviz.core.data.**Data**(*data*, *dispersion=None*, *dispersion_unit=None*, *name=u''*, **args*, ***kwargs*)

Class of the base data container for all data (of type numpy.ndarray) that is passed around in SpecViz. It inherits from astropy.nddata.NDData and provides functionality for arithmetic operations, I/O, and slicing.

Parameters

- **data** (*ndarray*) – Flux values.
- **dispersion** (ndarray or None) – Dispersion values. If not given, this is calculated from WCS.
- **dispersion_unit** (Unit or None) – Dispersion unit. If not given, this is obtained from WCS.
- **name** (*str*) – Short description of the spectrum.
- **args** (*tuple*) – Additional positional arguments.
- **kwargs** (*dict*) – Additional keyword arguments.

Examples

```
>>> d = Data.read(
...     'generic_spectra.fits', filter='Generic Fits (*.fits *.mits)')
>>> d = Data.read(
...     'generic_spectra.txt', filter='ASCII (*.txt *.dat)')
```

dispersion
Dispersion values.

dispersion_unit
Unit of dispersion.

class specviz.core.data.**Layer**(*source*, *mask*, *parent=None*, *name=u''*)
Class to handle layers in SpecViz.

A layer is a "view" into a *Data* object. It does not hold any data itself, but instead contains a special mask object and reference to the original data.

Since *Data* inherits from astropy.nddata.NDDataBase and provides the astropy.nddata.NDArithmeticMixin mixin, it is also possible to do arithmetic operations on layers.

Parameters

- **source** (*Data*) – Spectrum data object.
- **mask** (*ndarray*) – Mask for the spectrum data.
- **parent** (obj or None) – GUI parent.
- **name** (*str*) – Short description.

data
Flux quantity with mask applied.

dispersion
Dispersion quantity with mask applied.

mask
Mask for spectrum data.

meta
Spectrum metadata.

uncertainty
Flux uncertainty with mask applied.

unit
Flux unit.

wcs
WCS for spectrum data.

class specviz.core.data.**ModelLayer**(*model*, *source*, *mask*, *parent=None*, *name=u''*)
A layer for spectrum with a model applied.

Parameters

- **model** (*obj*) – Astropy model.
- **source** (*Data*) – Spectrum data object.
- **mask** (*ndarray*) – Mask for the spectrum data.
- **parent** (obj or None) – GUI parent.
- **window** (obj or None) – GUI window.
- **name** (*str*) – Short description.

data
Flux quantity from model.

model
Spectrum model.

uncertainty
Models do not need to contain uncertainties; override parent class method.

3.1.2 loaders

This module contains functions that perform the actual data parsing.

specviz.interfaces.loaders.**fits_reader**(*filename*, *filter*, ***kwargs*)
This generic function will query the loader factory, which has already loaded the YAML configuration files, in an attempt to parse the associated FITS file.

Parameters

- **filename** (*str*) – Input filename.
- **filter** (*str*) – File type for YAML look-up.
- **kwargs** (*dict*) – Keywords for Astropy reader.

specviz.interfaces.loaders.**fits_identify**(*origin*, **args*, ***kwargs*)
Check whether given filename is FITS. This is used for Astropy I/O Registry.

specviz.interfaces.loaders.**ascii_reader**(*filename*, *filter*, ***kwargs*)
Like *fits_reader()* but for ASCII file.

specviz.interfaces.loaders.**ascii_identify**(*origin*, **args*, ***kwargs*)
Check whether given filename is ASCII. This is used for Astropy I/O Registry.

3.1.3 statistics

Functions for spectral statistical analysis.

specviz.analysis.statistics.**extract**(*data*, *x_range*)
Extract a region from a spectrum.

Parameters

- **data** (*Data*) – Contains the spectrum to be extracted.
- **x_range** (*tuple*) – A spectral coordinate range as in `(wave1, wave2)`.

Returns **result** – Spectrum data with extracted region.

Return type *Data*

Examples

```
>>> d = Data(...)
>>> d2 = extract(d, (10000, 20000))
```

specviz.analysis.statistics.**stats**(*data*)
Compute basic statistics for a spectral region contained in a *Data* instance.

Parameters **data** (*Data*) – Typically this is returned by the *extract()* function.

Returns **statistics** – Statistics results.

Return type dict

Examples

```
>>> d = Data(...)
>>> d_stats = stats(d)
```

specviz.analysis.statistics.**eq_width**(*cont1_stats*, *cont2_stats*, *line*, *mask=None*)

Compute an equivalent width given stats for two continuum regions, and a *Data* instance with the extracted spectral line region.

This uses for now a very simple continuum subtraction method; i.e., it just subtracts a constant from the line spectrum, where the constant is `(continuum1[mean] + continuum2[mean]) / 2`.

Parameters

- **cont2_stats** (*cont1_stats,*) – This is returned by the *stats()* function.
- **line** (*Data*) – This is returned by the *extract()* function.
- **mask** (*ndarray*) – Boolean mask.

Returns **ew, flux, avg_cont** – Flux and equivalent width values.

Return type float

Examples

```
>>> d = Data(...)
>>> cont1 = extract(d, (100, 5000))
>>> cont2 = extract(d, (18000, 20000))
>>> cont1_stats = stats(cont1)
>>> cont2_stats = stats(cont2)
>>> line = extract(d, (15000, 18000))
>>> flux, ew = eq_width(cont1_stats, cont2_stats, line)
```

specviz.analysis.statistics.**fwzi**(*cont1_stats*, *cont2_stats*, *line*)

Compute full width at zero intensity (FWZI) for the given spectrum. Continuum calculations are similar to *eq_width()*.

Parameters

- **cont2_stats** (*cont1_stats,*) – This is returned by the *stats()* function.
- **line** (*Data*) – This is returned by the *extract()* function.

Returns

- **fwzi_value** (*float*) – FWZI value.
- **w_range** (*tuple*) – Wavelengths used to calculate FWZI.

Examples

```
>>> d = Data(...)
>>> cont1 = extract(d, (100, 5000))
>>> cont2 = extract(d, (18000, 20000))
>>> cont1_stats = stats(cont1)
>>> cont2_stats = stats(cont2)
>>> line = extract(d, (15000, 18000))
>>> fwzi_value, w_range = fwzi(cont1_stats, cont2_stats, line)
```

specviz.analysis.statistics.**centroid**(*data*, *mask=None*)

Compute centroid for the given spectrum.

```
w_cen = integral(wave*flux) / integral(flux)
```

Parameters **data** (*Data*) – Extracted spectrum data.

Returns **wcen** – Centroid wavelength.

Return type float

Examples

```
>>> d = Data(...)
>>> line = extract(d, (15000, 20000))
>>> wcen_em = centroid(line)
```

CHAPTER 4

Indices and tables

- genindex
- modindex

a

c

i

A

C

D

E

F

L

M

S

U

W