astrodendro Documentation

Release 0.2.0

Thomas Robitaille, Braden McDonald, Chris Beaumont, Erik Roso

September 29, 2016

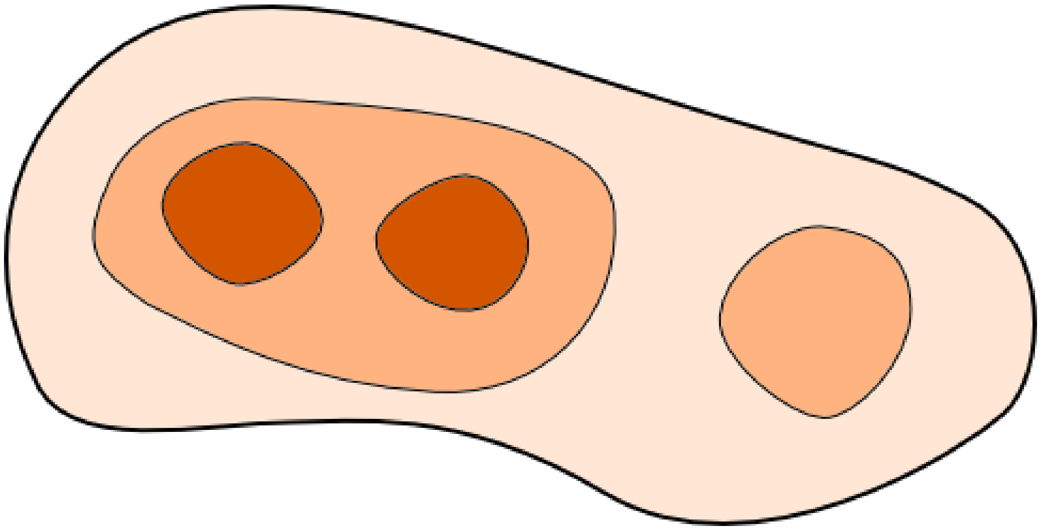
Contents

The `astrodendro` package provides an easy way to compute dendrograms of observed or simulated Astronomical data in Python.

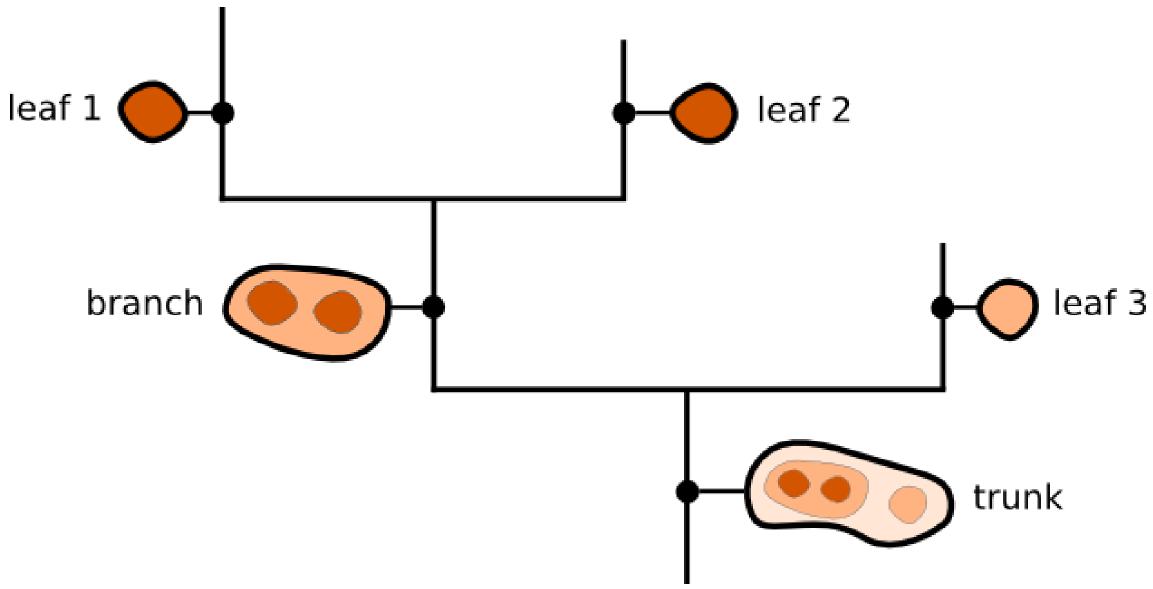
CHAPTER 1

About dendrograms

The easiest way to think of a dendrogram is to think of a tree that represents the hierarchy of the structures in your data. If you consider a two-dimensional map of a hierarchical structure that looks like:

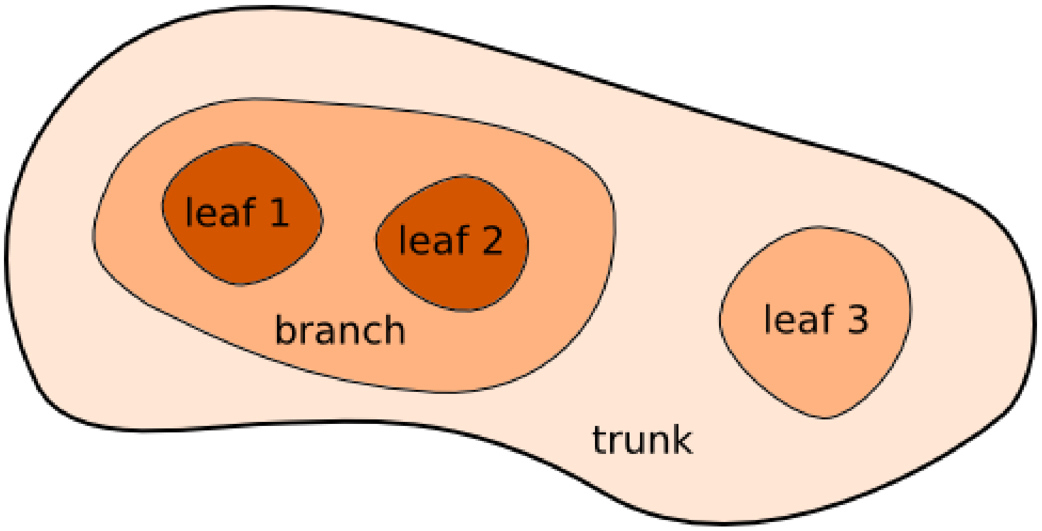


the equivalent dendrogram/tree representation would look like:



A dendrogram is composed of two types of structures: *branches*, which are structures which split into multiple sub-structures, and *leaves*, which are structures that have no sub-structure. Branches can split up into branches and leaves, which allows hierarchical structures to be adequately represented. The term *trunk* is used to refer to a structure that has no parent structure.

Mapping these terms back onto the structure gives the following:



For an example of use of dendrograms on real data, see Goodman, A. et al (2009).

CHAPTER 2

Documentation

2.1 Installing astrodendro

2.1.1 Requirements

This package has the following depdenencies:

- Python 2.6 or later (Python 3.x is supported)
- Numpy 1.4.1 or later
- Astropy 0.2.0 or later, optional (needed for reading/writing FITS files and for analysis code)
- h5py 0.2.0 or later, optional (needed for reading/writing HDF5 files)

2.1.2 Installation

To install the latest stable release, you can type:

```
pip install astrodendro
```

or you can download the latest tar file from PyPI and install it using:

```
python setup.py install
```

2.1.3 Developer version

If you want to install the latest developer version of the dendrogram code, you can do so from the git repository:

```
git clone https://github.com/dendrograms/astrodendro.git
cd astrodendro
python setup.py install
```

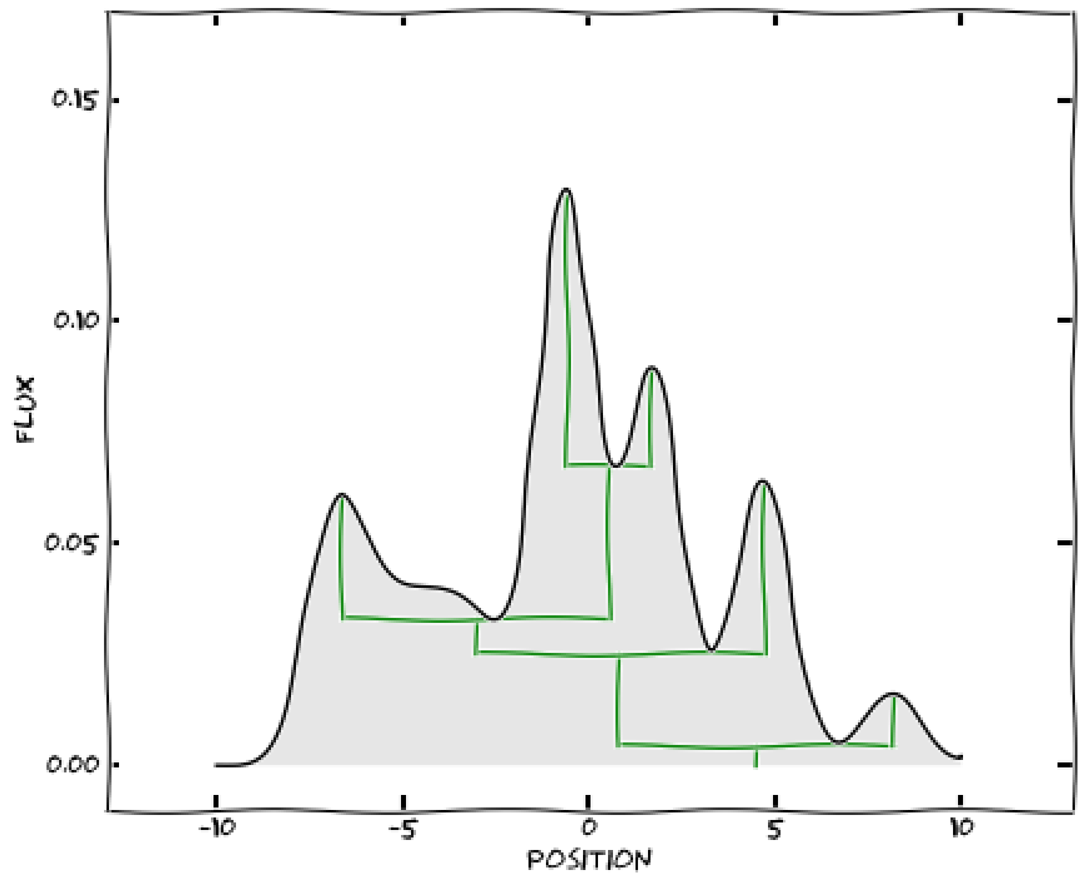
You may need to add the `--user` option to the last line if you do not have root access.

2.2 An Illustrated Description of the Core Algorithm

This page contains an explanation of the algorithm behind the Python dendrogram code. This is demonstrated with a step by step example of how the algorithm constructs the tree structure of a very simple one-dimensional dataset. Even though this dataset is very simple, what is described applies to datasets with any number of dimensions.

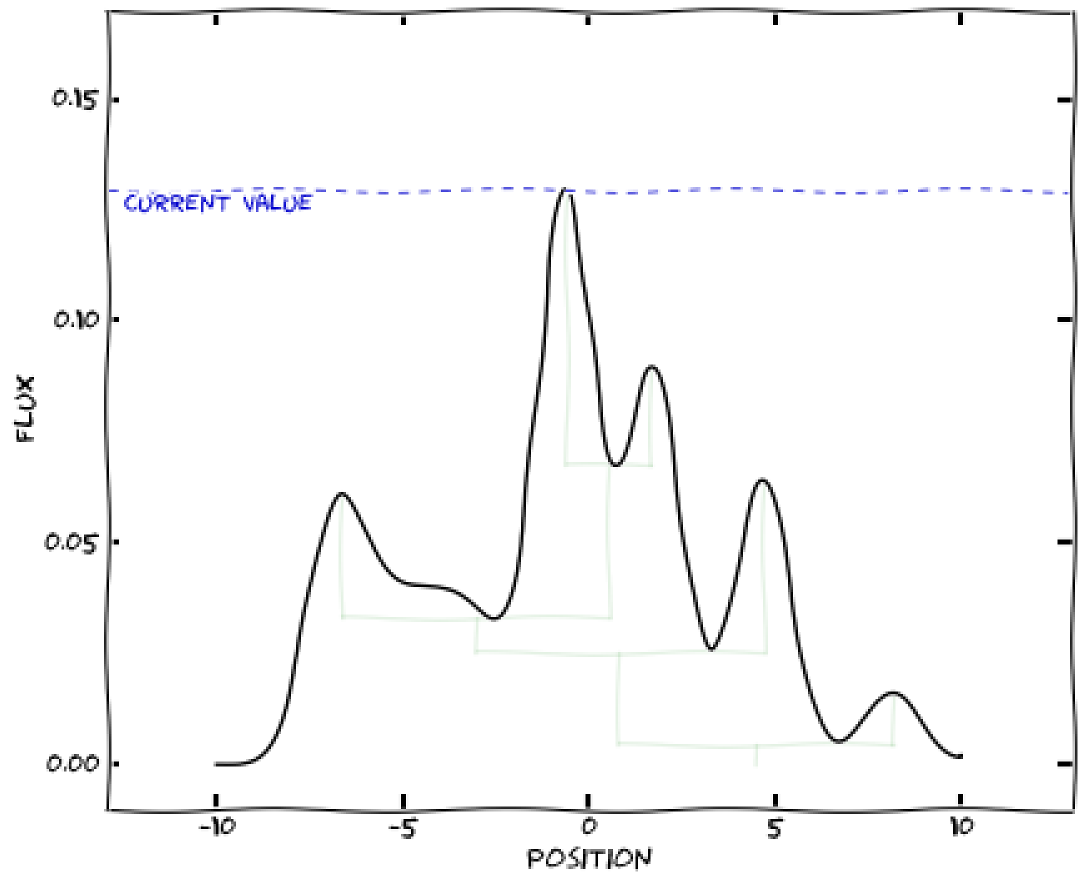
2.2.1 Basic example

The following diagram shows a one-dimensional dataset (with flux versus position) in the solid black line, with the corresponding dendrogram for that dataset overplotted in green:

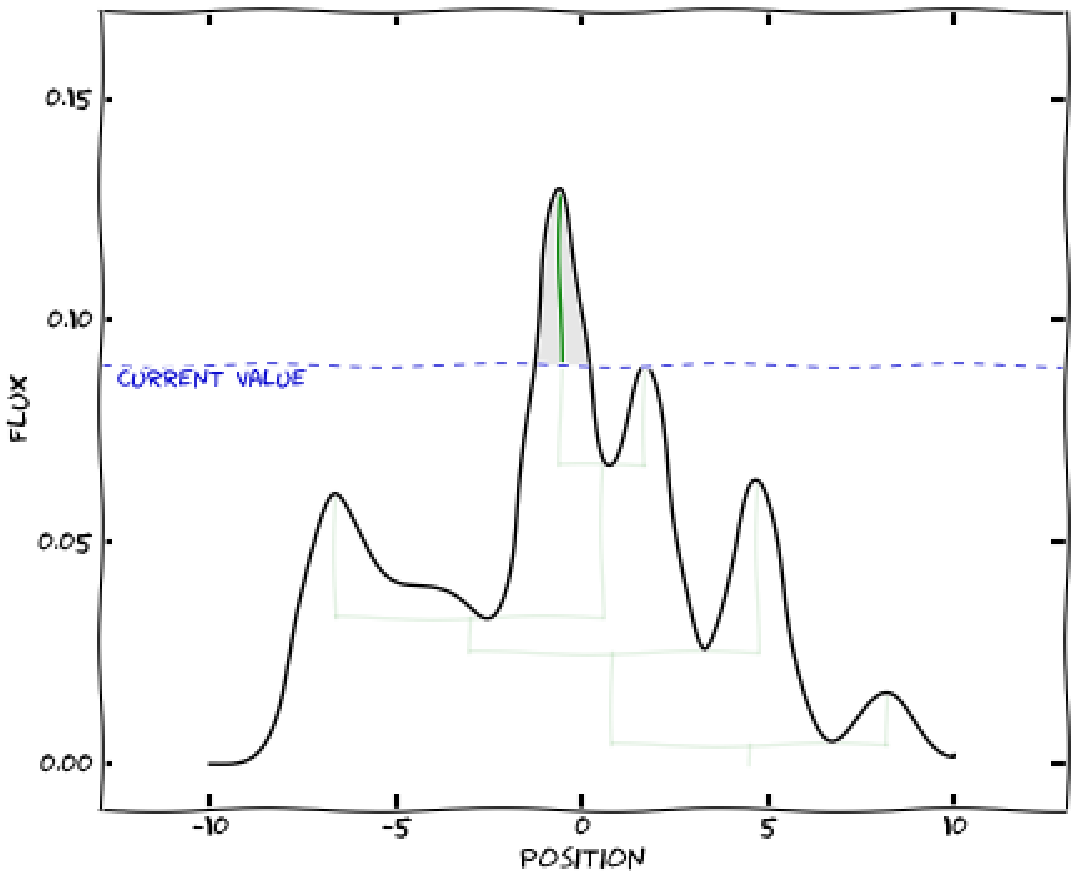


In the rest of this document, we will refer to the individual points in this dataset as *pixels*.

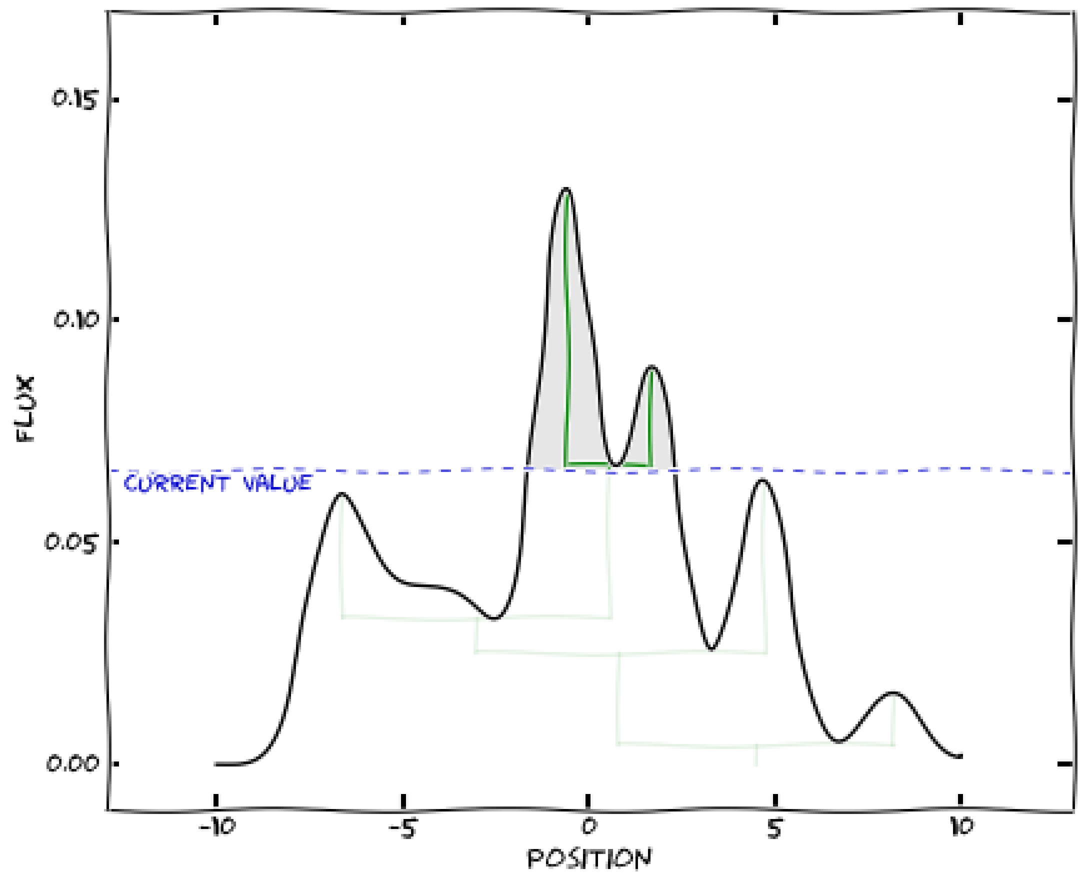
The way the algorithm works is to construct the tree starting from the brightest pixels in the dataset, and progressively adding fainter and fainter pixels. We will illustrate this by showing the current value being considered, with the following blue dashed line:



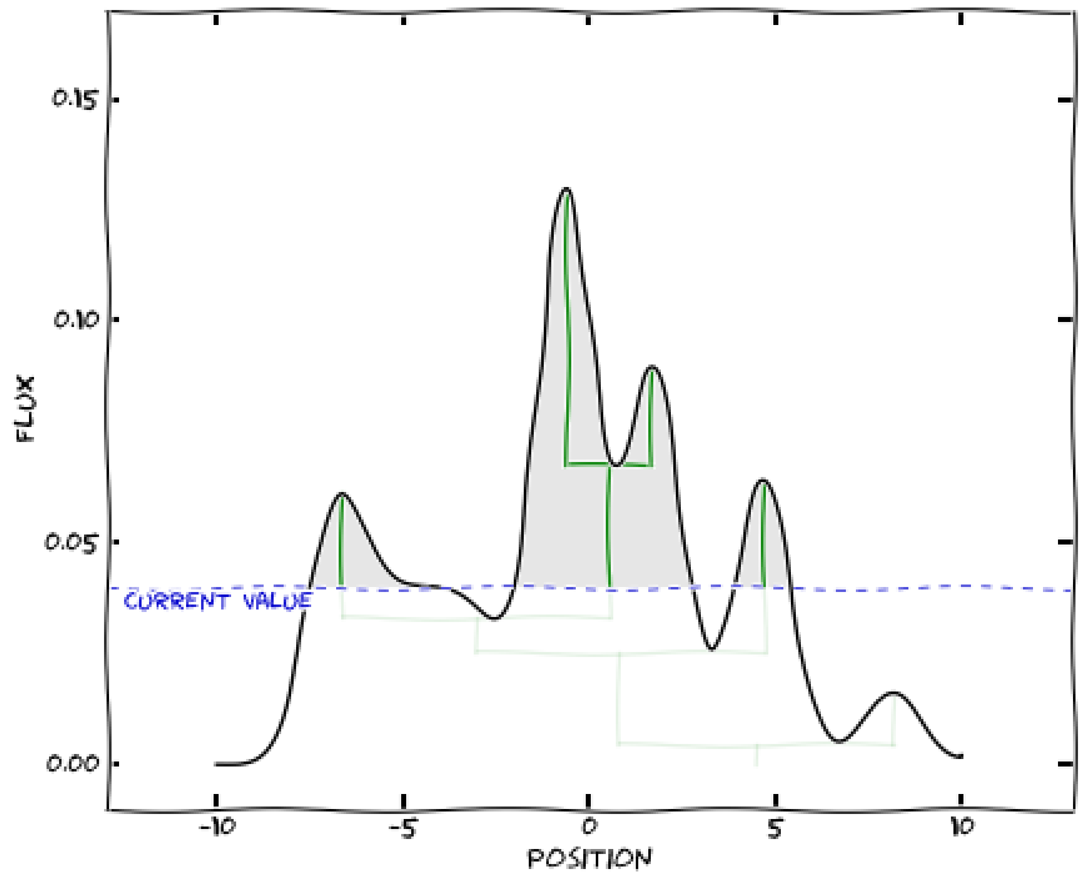
Let's now start moving this line down, starting from the peak pixel in the dataset. We create our first structure from this pixel. We then move to the pixel with the next largest value, and each time, we decide whether to join the pixel to an existing structure, or create a new structure. We only start a new structure if the value of the pixel is greater than its immediate neighbors, and therefore is a local maximum. The first structure being constructed is shown below:



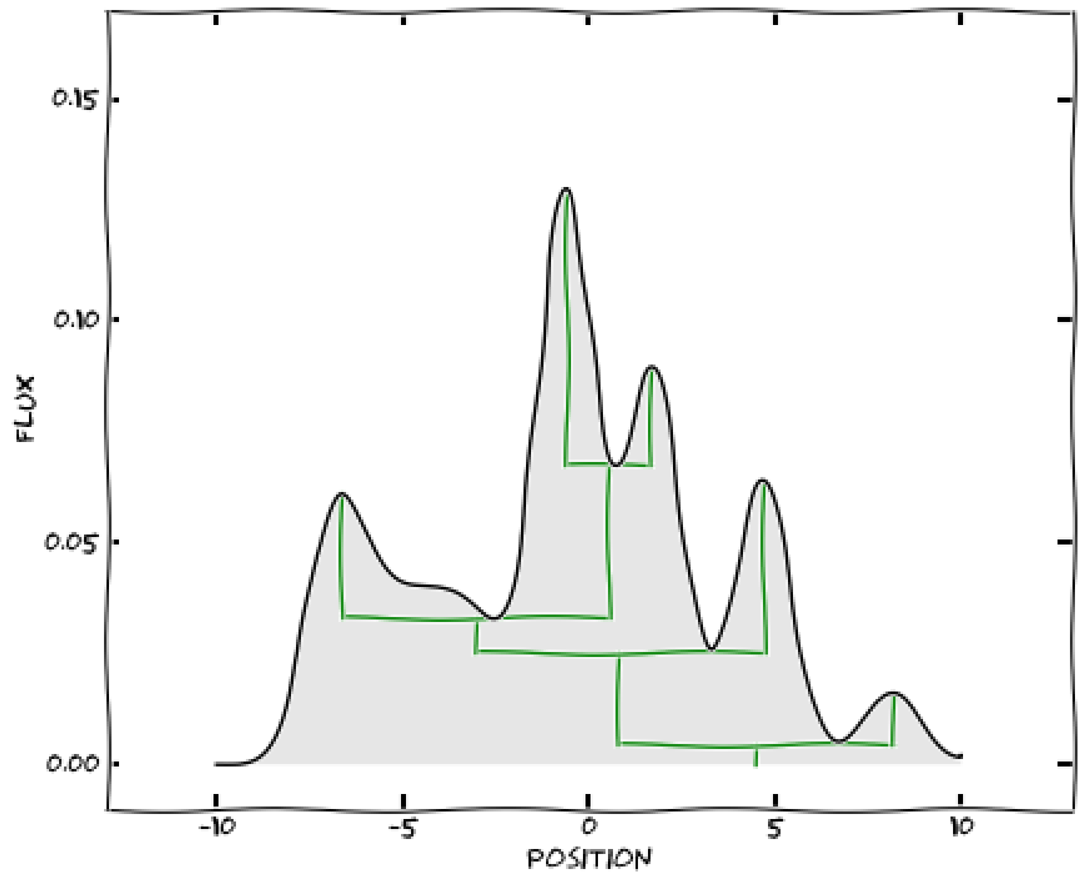
We have now found a local maximum, so rather than add this pixel to the first structure, we create a new structure. As we move further down, both structures keep growing, until we reach a pixel that is not a local maximum, and is adjacent to both existing structures:



At this point, we merge the structures into a branch, which is shown by a green horizontal line. As we move down further, the branch continues to grow, and we very quickly find two more local maxima which cause new structures to be created:



These structures eventually merge, and we end up with a single tree:

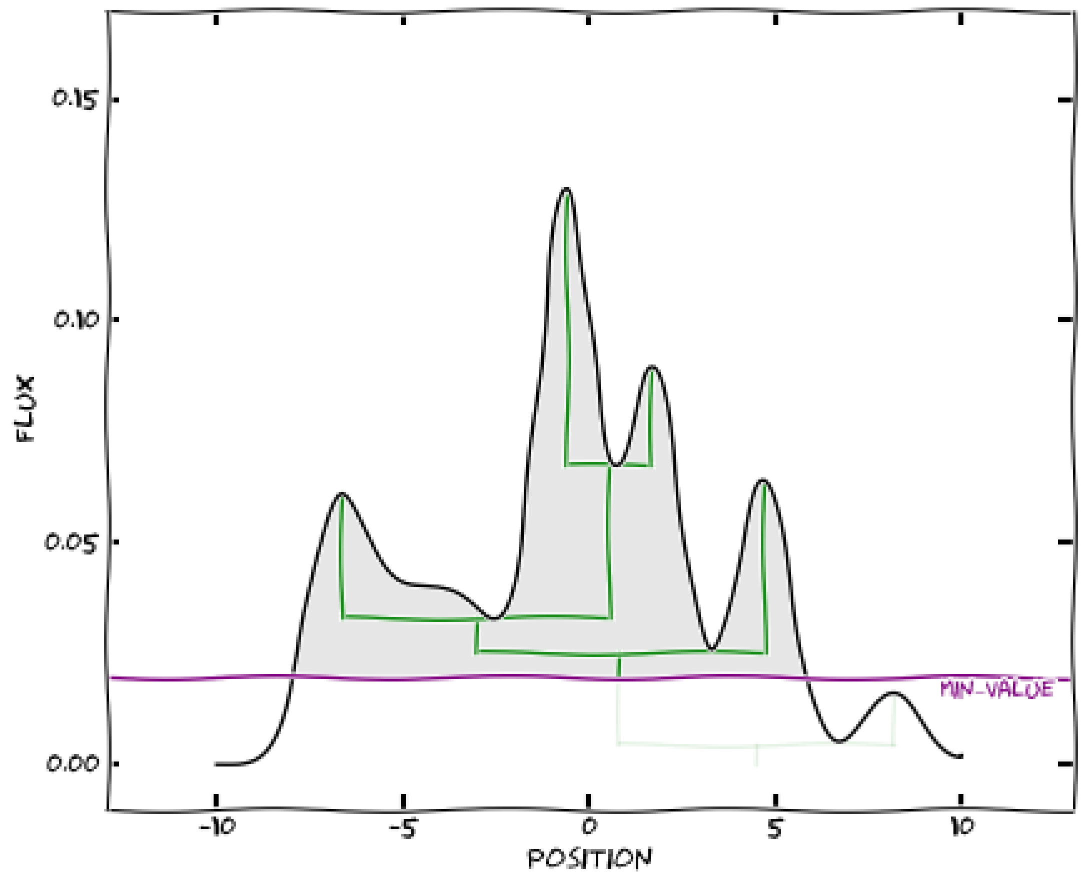


2.2.2 Accounting for noise

Setting a minimum value (`min_value`)

Most real-life datasets are likely to contain some level of noise, and below a certain value of the noise, there is no point in expanding the tree since it will not be measuring anything physical; new branches will be 'noise spikes'. By default, the minimum value is set to negative infinity, which means all pixels are added to the tree. However, you will very likely want to change this so that only significant features above the noise are included.

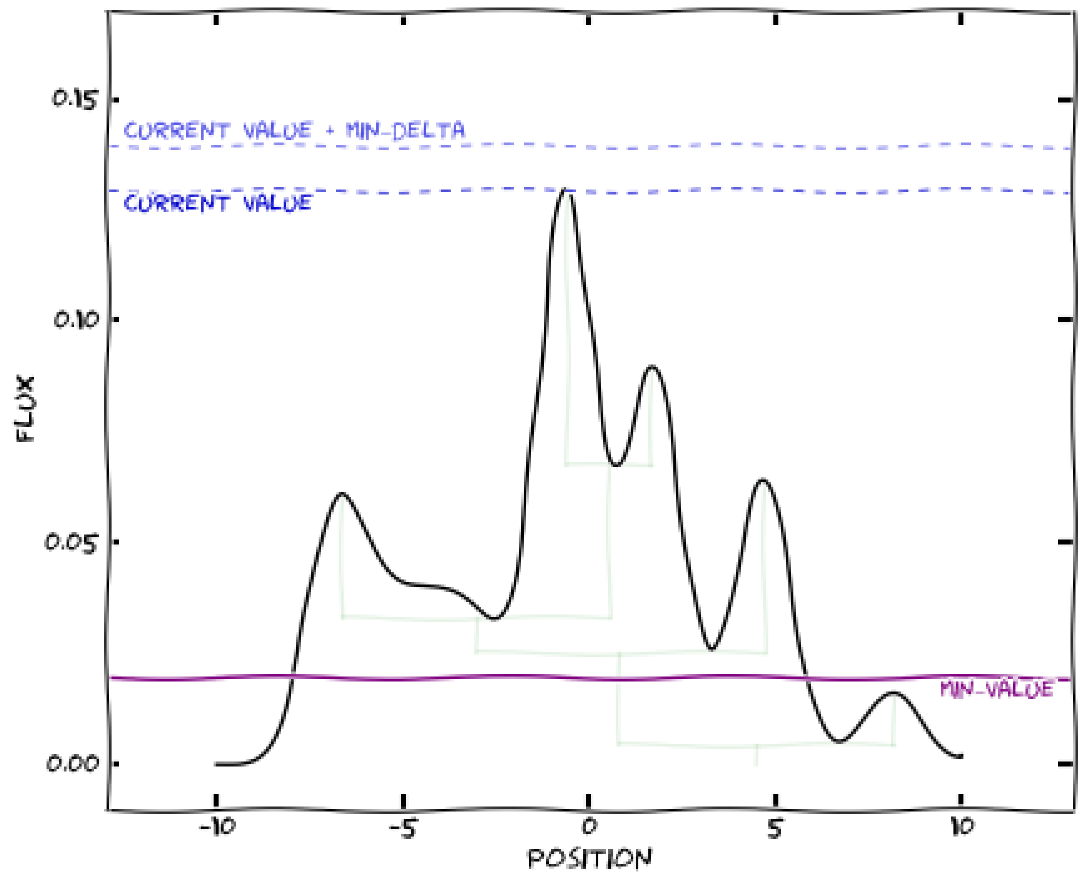
Let's go back to the original data. We have left the outline of the complete tree for reference. We now set a minimum value, which we show below with the purple line. This is controlled by the `min_value` option in *`compute()`*.



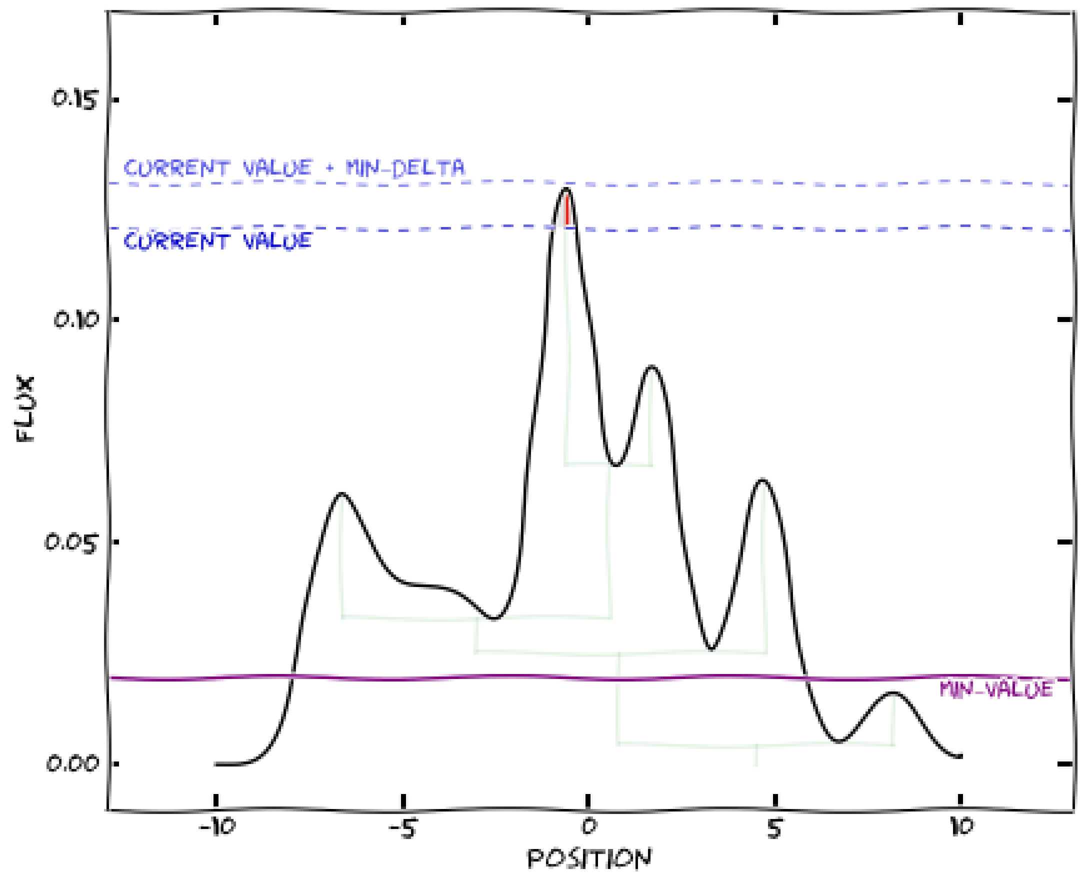
The effect on the tree is simply to get rid of (or *prune*) any structure peaking below this minimum. In this case, the peak on the right is no longer part of the tree since it is below the minimum specified value.

Setting a minimum significance for structures (`min_delta`)

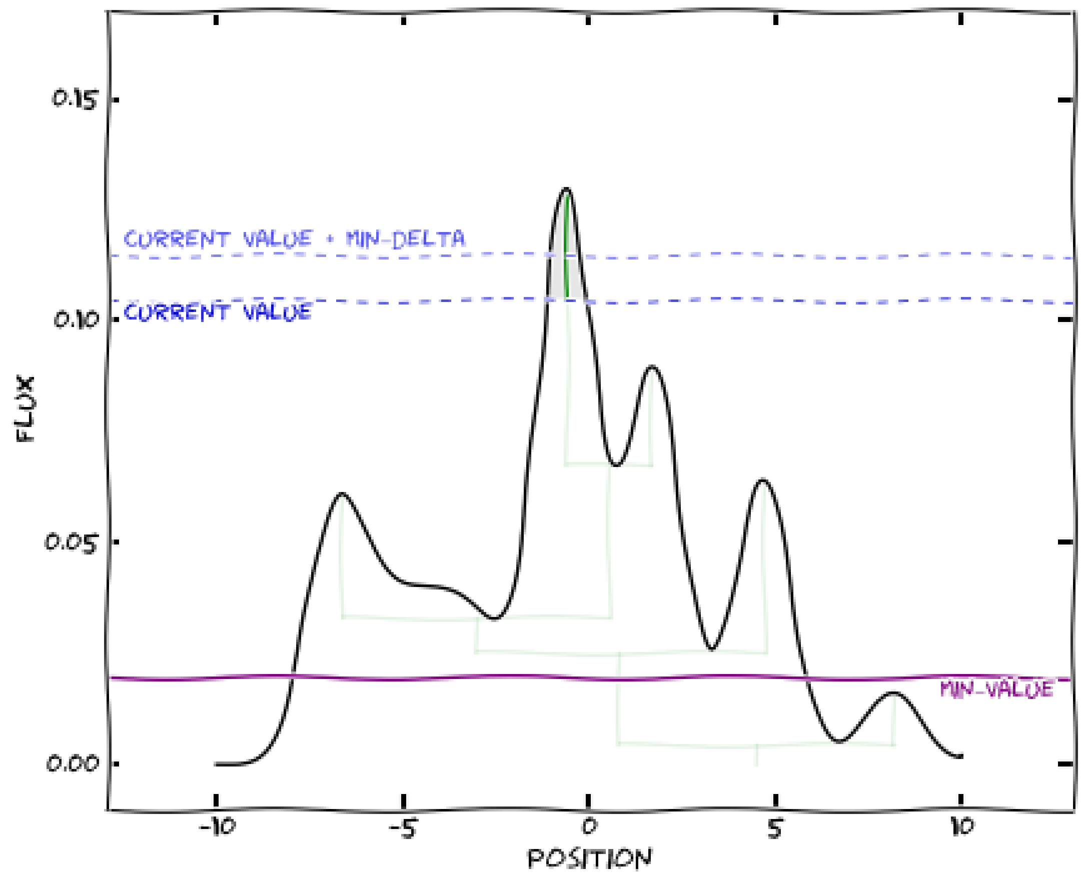
If our data are noisy, we also want to avoid including *local* maxima that - while above the minimum absolute value specified above - are only identified because of noise, so we can also define a minimum height required for a structure to be retained. This is the `min_delta` parameter in `compute()`. We show the value corresponding to the current value being considered plus this minimum height:



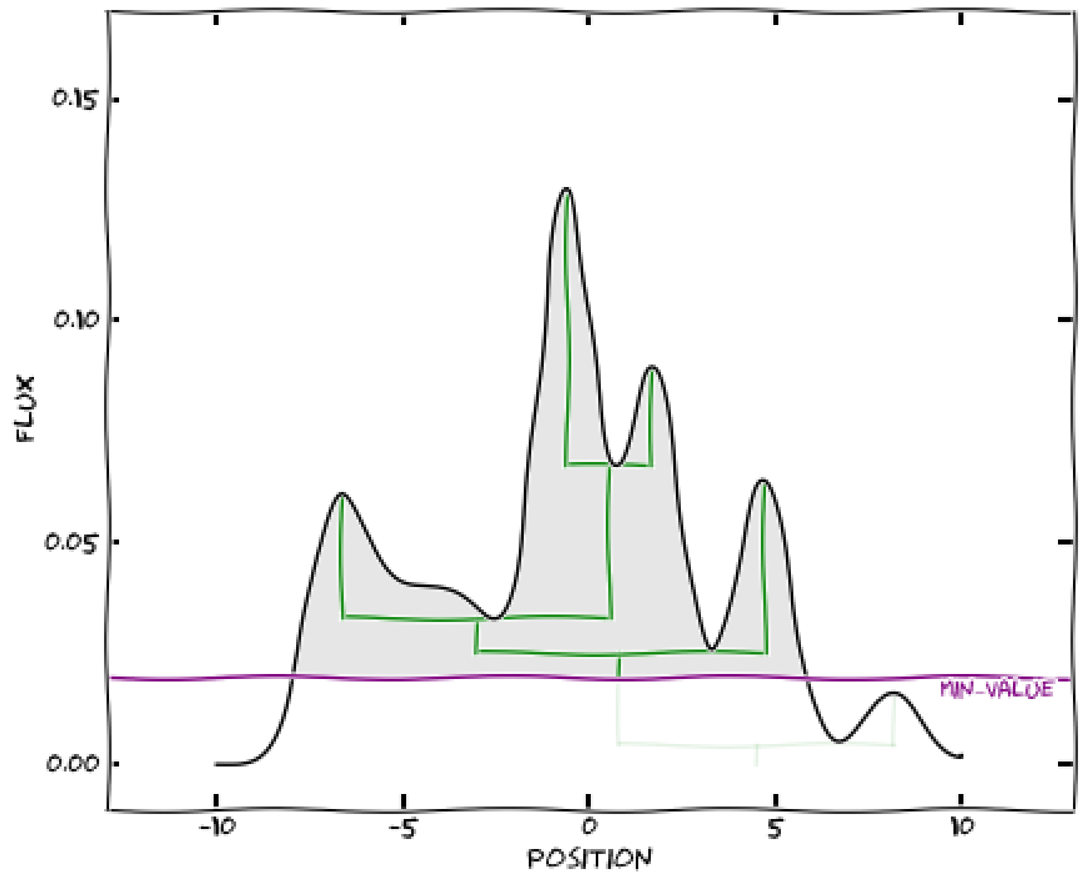
In this case, `min_delta` is set to 0.01. As we now move down in flux as before, the structure first appears red. This indicates that the structure is not yet part of the tree:



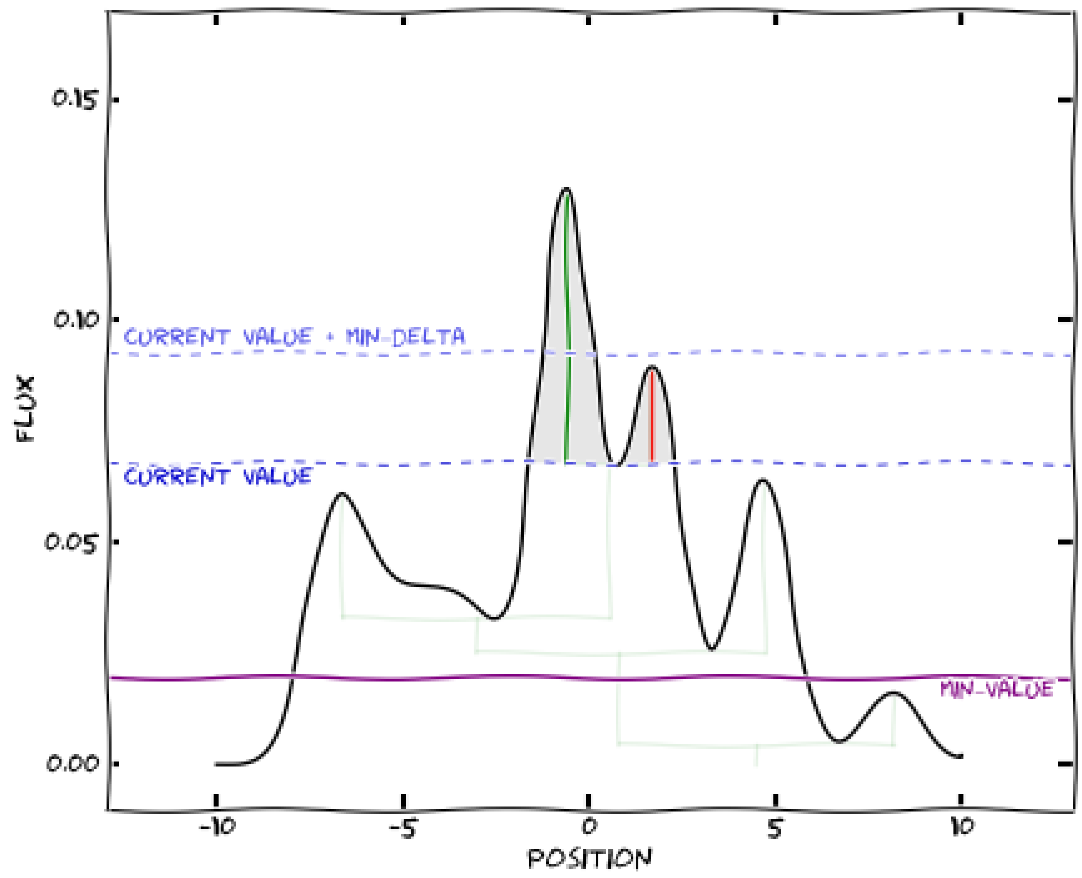
Once the height of the structure exceeds the minimum specified, the structure can now be considered part of the tree:



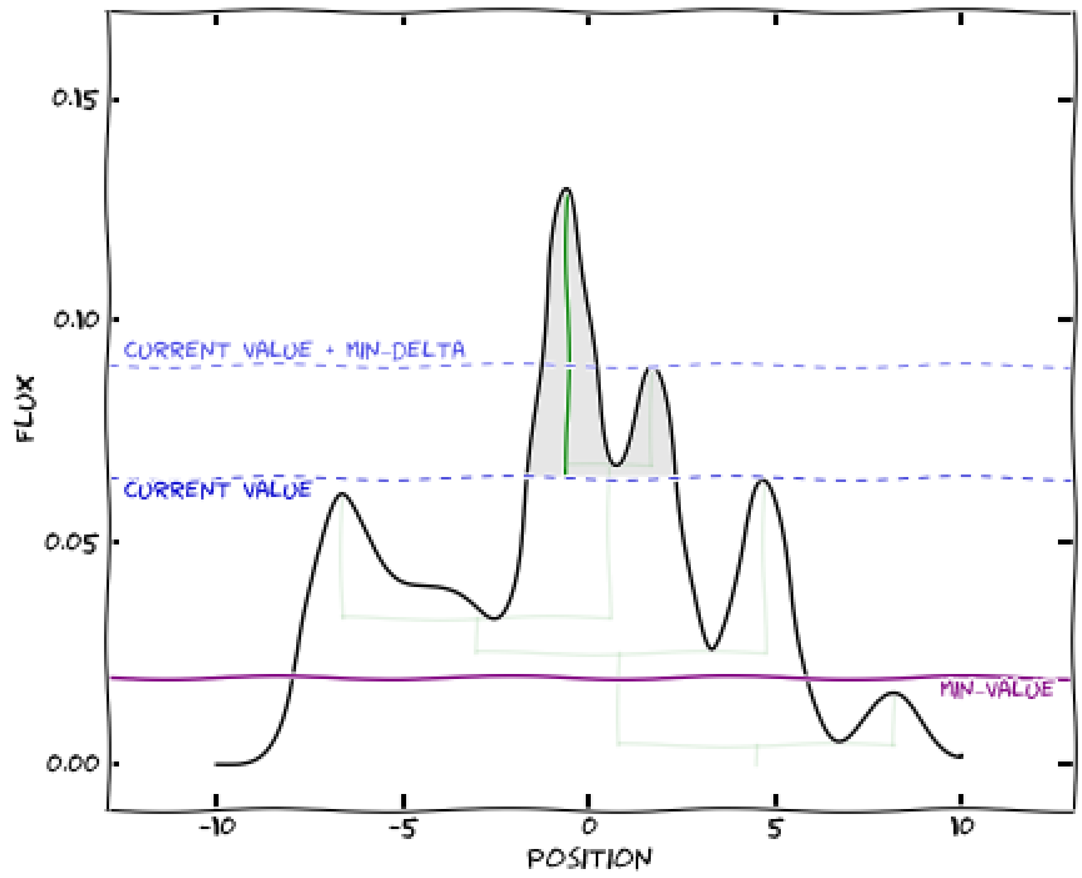
In this case, all structures that are above the minimum value are also all large enough to be significant, so the tree is the same as before:



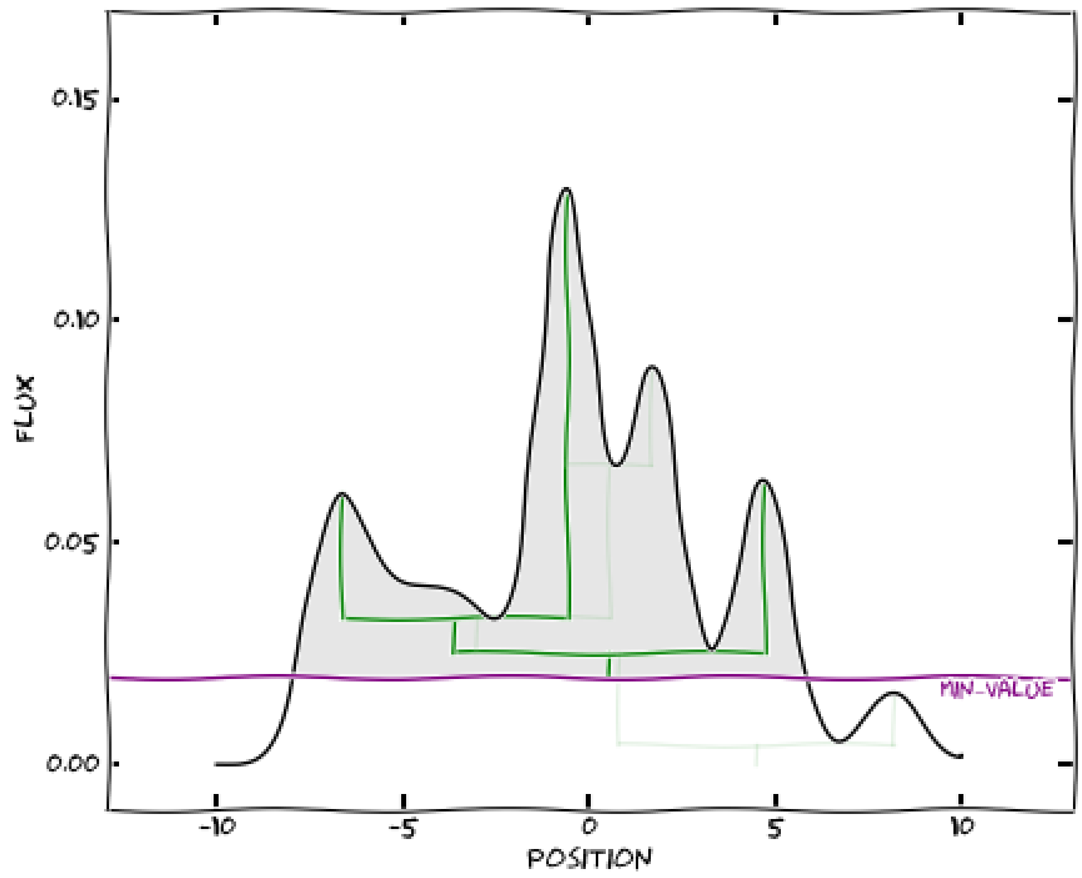
We can now repeat this experiment, but this time, with a larger minimum height for structures to be retained (`min_delta=0.025`). Once we reach the point where the second peak would have been merged, we can see that it is not high enough above the merging point to be considered an independent structure:



and the pixels are then simply added to the first structure, rather than creating a branch:



We can now see that the final tree looks a little different to the original one, because the second largest peak was deemed insignificant:



2.2.3 Additional options

In addition to the minimum height of a structure, it is also possible to specify the minimum number of pixels that a structure should contain in order to remain an independent structure (`min_npix`), and in the future, it will be possible to specify arbitrary criteria, such as the proximity to a given point or set of coordinates.

2.3 Computing and exploring dendrograms

For a graphical description of the actual algorithm used to compute dendrograms, see An Illustrated Description of the Core Algorithm.

2.3.1 Computing a Dendrogram

Dendrograms can be computed from an n-dimensional array using:

```
>>> from astrodendro import Dendrogram
>>> d = Dendrogram.compute(array)
```

where `array` is a Numpy array and `d` is then an instance of the `Dendrogram` class, which can be used to access the computed dendrogram (see *Exploring the Dendrogram* below). Where the `array` comes from is not important - for example it can be read in from a FITS file, from an HDF5 file, or it can be generated in memory. If you are interested in making a dendrogram from data in a FITS file, you can do:

```
>>> from astropy.io import fits
>>> array = fits.getdata('observations.fits')
>>> from astrodendro import Dendrogram
>>> d = Dendrogram.compute(array)
```

- `min_value`: the minimum value to consider in the dataset - any value lower than this will not be considered in the dendrogram. If you are working with observations, it is likely that you will want to set this to the "detection level," for example 3- or 5-sigma, so that only significant values are included in the dendrogram. By default, all values are used.
- `min_delta`: how significant a leaf has to be in order to be considered an independent entity. The significance is measured from the difference between its peak flux and the value at which it is being merged into the tree. If you are working with observational data, then you could set this to, e.g., 1-sigma, which means that any leaf that is locally less than 1-sigma tall is combined with its neighboring leaf or branch and is no longer considered a separate entity.
- `min_npix`: the minimum number of pixels/values needed for a leaf to be considered an independent entity. When the dendrogram is being computed, and when a leaf is about to be joined onto a branch or another leaf, if the leaf has fewer than this number of pixels, then it is combined with the branch or leaf it is being merged with and is no longer considered a separate entity. By default, this parameter is set to zero, so there is no minimum number of pixels required for leaves to remain independent entities.

These options are illustrated graphically in An Illustrated Description of the Core Algorithm.

As an example, we can use a publicly available extinction map of the Perseus star-formation region from the The COordinated Molecular Probe Line Extinction Thermal Emission (COMPLETE) Survey of Star Forming Regions (`PerA_Extn2MASS_F_Gal.fits`, originally obtained from http://hdl.handle.net/10904/10080). The units of the map are magnitudes of extinction, and we want to make a dendrogram of all structures above a minimum value of 2 magnitudes, and we only consider leaves with at least 10 pixels and which have a peak to base difference larger than one magnitude of extinction:

```
>>> from astrodendro import Dendrogram
>>> from astropy.io import fits
>>> image = fits.getdata('PerA_Extn2MASS_F_Gal.fits')
>>> d = Dendrogram.compute(image, min_value=2.0, min_delta=1., min_npix=10)
```

By default, the computation will be silent, but for large dendrograms, it can be useful to have an idea of how long the computation will take:

```
>>> d = Dendrogram.compute(image, min_value=2.0, min_delta=1., min_npix=10,
                           verbose=True)
Generating dendrogram using 6,386 of 67,921 pixels (9% of data)
[=========================>                ] 64%
```

The '9% of data' indicates that only 9% of the data are over the *min_value* threshold.

2.3.2 Exploring the Dendrogram

Once the dendrogram has been computed, you will want to explore/visualize it. You can access the full tree from the computed dendrogram. Assuming that you have computed a dendrogram with:

```
>>> d = Dendrogram.compute(array, ...)
```

you can now access the full tree from the `d` variable.

The first place to start is the *trunk* of the tree (the `trunk` attribute), which is a *list* of all the structures at the lowest level. Unless you left `min_value` to the default setting, which would mean that all values in the dataset are used, it's likely that not all structures are connected. So the `trunk` is a collection of items at the lowest level, each of which could be a leaf or a branch (itself having leaves and branches). In the case of the Perseus extinction map, we get:

```
>>> d.trunk
[<Structure type=leaf idx=101>,
 <Structure type=branch idx=2152>,
 <Structure type=leaf idx=733>,
 <Structure type=branch idx=303>]
```

In the above case, the trunk contains two leaves and two branches. Since `trunk` is just a list, you can access items in it with e.g.:

```
>>> d.trunk[1]
<Structure type=branch idx=2152>
```

Branches have a `children` attribute that returns a list of all sub-structures, which can include branches and leaves. Thus, we can return the sub-structures of the above branch with:

```
>>> d.trunk[1].children
[<Structure type=branch idx=1680>,
 <Structure type=branch idx=5771>]
```

which shows that the child branch is composed of two more branches. We can therefore access the sub-structures of these branch with e.g.:

```
>>> d.trunk[1].children[0].children
[<Structure type=leaf idx=1748>,
 <Structure type=leaf idx=1842>]
```

which shows this branch splitting into two leaves.

We can access the properties of leaves as follows:

```
>>> leaf = d.trunk[1].children[0].children[0]
>>> leaf.indices
(array([143, 142, 142, 142, 139, 141, 141, 141, 143, 140, 140]),
 array([116, 114, 115, 116, 115, 114, 115, 116, 115, 115, 114]))
>>> leaf.values
array([ 2.7043395 ,  2.57071948,  3.4551146 ,  3.29953575,  2.53844047,
        2.59633183,  3.11309052,  2.70936489,  2.81024122,  2.76864815,
        2.52840114], dtype=float32)
```

A full list of attributes and methods for leaves and branches (i.e. structures) is available from the *`Structure`* page, while a list of attributes and methods for the dendrogram itself is available from the *`Dendrogram`* page.

2.3.3 Saving and loading the dendrogram

A *`Dendrogram`* object can be exported to an HDF5 file (requires h5py) or FITS file (requires astropy). To export the dendrogram to a file, use:

```
>>> d.save_to('my_dendrogram.hdf5')
```

or:

```
>>> d.save_to('my_dendrogram.fits')
```

and to load and existing dendrogram:

```
>>> d = Dendrogram.load_from('my_other_dendrogram.hdf5')
```

or:

```
>>> d = Dendrogram.load_from('my_other_dendrogram.fits')
```

If you wish, you can use this to separate the computation and analysis of the dendrogram into two scripts, to ensure that the dendrogram is only computed once. For example, you could have a script `compute.py` that contains:

```
from astropy.io import fits
from astrodendro import Dendrogram

array = fits.getdata('observations.fits')
d = Dendrogram.compute(array)
d.save_to('dendrogram.fits')
```

and a second file containing:

```
from astrodendro import Dendrogram
d = Dendrogram.load_from('dendrogram.fits')

# any analysis code here
```

2.4 Plotting Dendrograms

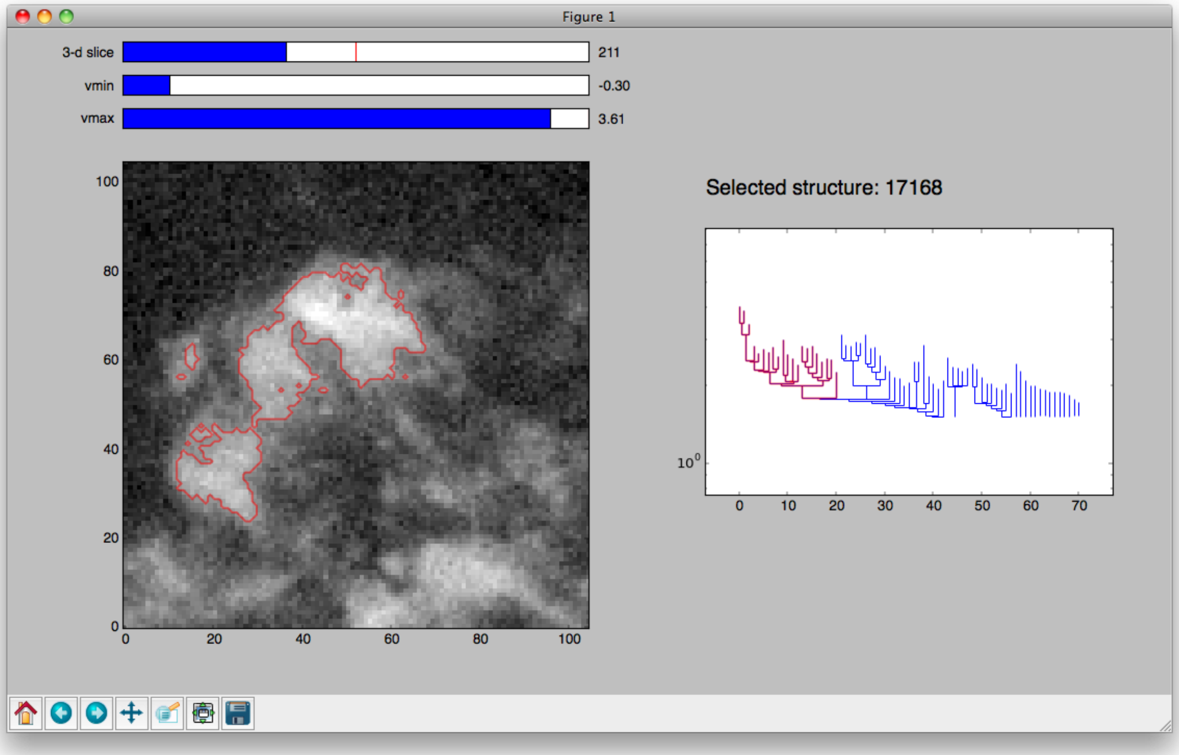
Once you have computed a dendrogram, you will likely want to plot it as well as over-plot the structures on your original image.

2.4.1 Interactive Visualization

One you have computed your dendrogram, the easiest way to view it interactively is to use the *`viewer()`* method:

```
d = Dendrogram.compute(...)
v = d.viewer()
v.show()
```

This will launch an interactive window showing the original data, and the dendrogram itself. Note that the viewer is only available for 2 or 3-d datasets. The main window will look like this:



Within the viewer, you can:

Highlight structures: either click on structures in the dendrogram to highlight them, which will also show them in the image view on the left, or click on pixels in the image and have the corresponding structure be highlighted in the dendrogram plot. Clicking on a branch in the dendrogram plot or in the image will highlight that branch and all sub-structures.

Multiple structures can be highlighted in different colors using the three mouse buttons: Mouse button 1 (Left-click or "regular" click), button 2 (Middle-click or "alt+click"), and button 3 (Right-click/"ctrl+click"). Each selection is independent of the other two; any of the three can be selected either by clicking on the image or the dendrogram.

Change the image stretch: use the `vmin` and `vmax` sliders above the image to change the lower and upper level of the image stretch.

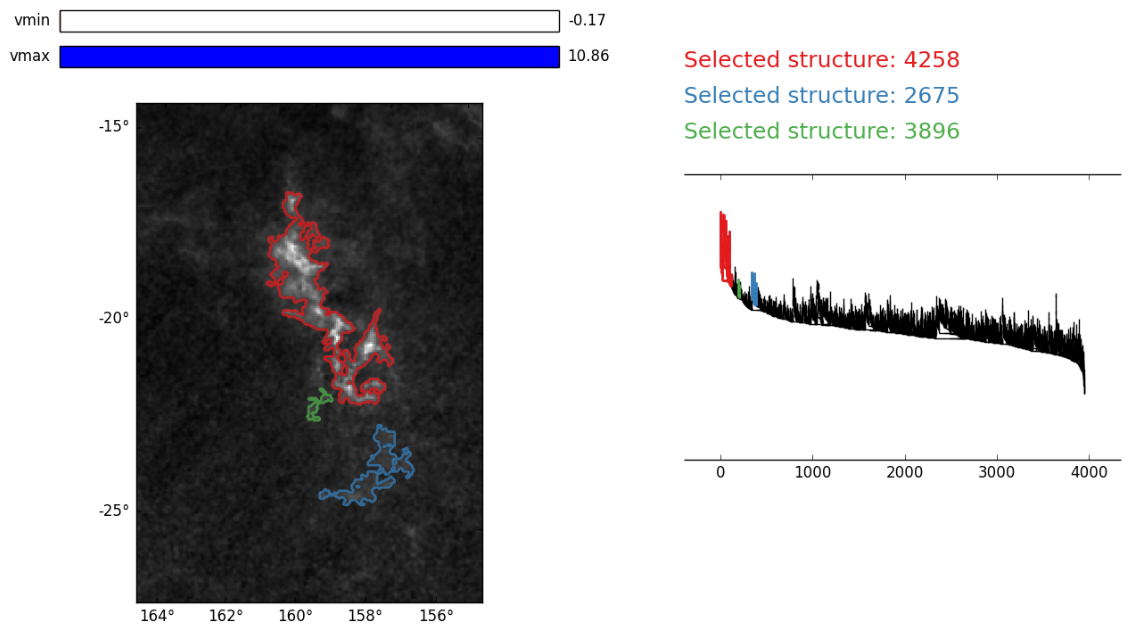
Change slice in a 3-d cube: if you select a structure in the dendrogram for a 3-d cube, the cube will automatically change to the slice containing the peak pixel of the structure (including sub-structures). However, you can also change slice manually by using the `slice` slider.

View the selected structure ID: in a computed dendrogram, every structure has a unique integer ID (the `.idx` attribute) that can be used to recognize the identify the structure when computing catalogs or making plots manually (see below).

Display astronomical coordinates: If your data has an associated WCS object (for example, if you loaded your data from a FITS file with astronomical coordinate information), the interactive viewer will display the coordinates using `wcsaxes`:

```
from astropy.io.fits import getdata
from astropy import wcs

data, header = getdata('astrodendro/docs/PerA_Extn2MASS_F_Gal.fits', header=True)
wcs = wcs.WCS(header)
```

```
d = astrodendro.Dendrogram.compute(data, wcs=wcs)
v = d.viewer()
v.show()
```

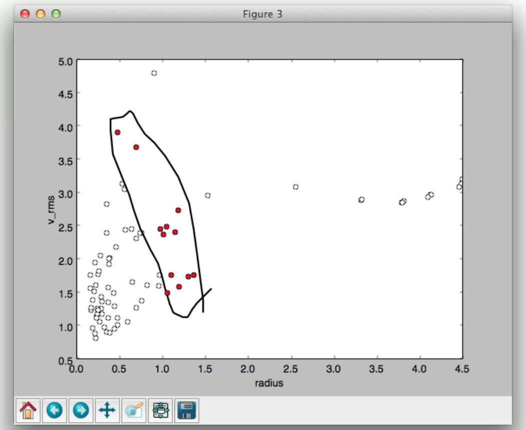


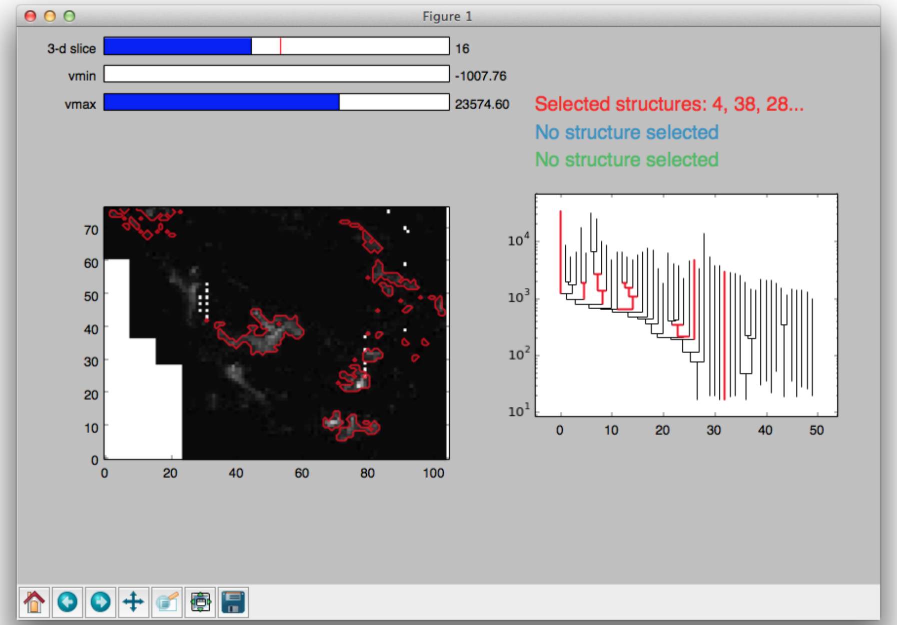
Note that this functionality requires that the `wcsaxes` package is installed. Installation instructions can be found here: http://wcsaxes.readthedocs.org/en/latest/

Linked scatter plots: If you have built a catalog (see Computing Dendrogram Statistics), you can also display a scatterplot of two catalog columns, linked to the viewer. The available catalog columns can be accessed as `` `catalog.colnames` ``. Selections in the main viewer update the colors of the points in this plot:

```
from astrodendro.scatter import Scatter
... code to create a dendrogram (d) and catalog ...
dv = d.viewer()
ds = Scatter(d, dv.hub, catalog, 'radius', 'v_rms')
dv.show()
```

The catalog properties of dendrogram structures will be plotted here. You can select structures directly from the scatter plot by clicking and dragging a lasso, and the selected structures will be highlighted in other plots:





To set logarithmic scaling on either the x axis, the y axis, or both, the following convenience methods are defined:

```
ds.set_semilogx()
ds.set_semilogy()
ds.set_loglog()

# To unset logarithmic scaling, pass `log=False` to the above methods, i.e.
ds.set_loglog(False)
```

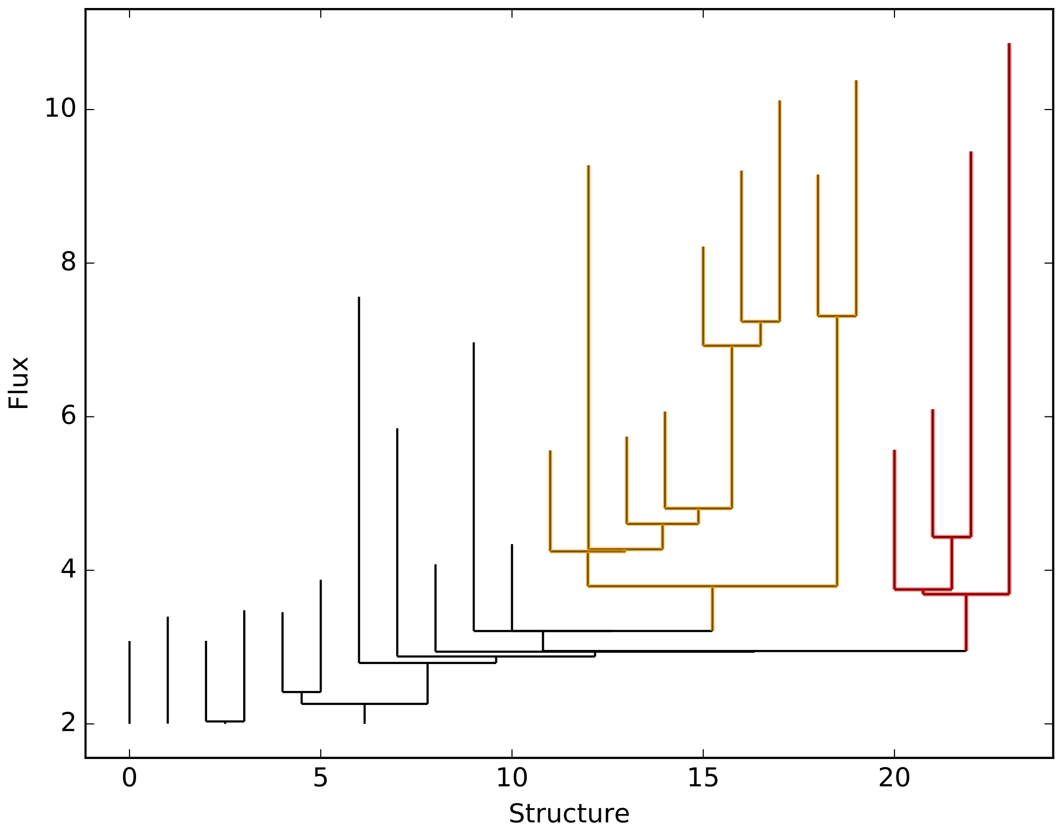
2.4.2 Making plots for publications

While the viewer is useful for exploring the dendrogram, it does not allow one to produce publication-quality plots. For this, you can use the non-interactive plotting interface. To do this, you can first use the `plotter()` method to provide a plotting tool:

```
d = Dendrogram.compute(...)
p = d.plotter()
```

and then use this to make the plot you need. The following complete example shows how to make a plot of the dendrogram of the extinction map of the Perseus region (introduced in Computing and exploring dendrograms) using `plot_tree()`, highlighting two of the main branches:

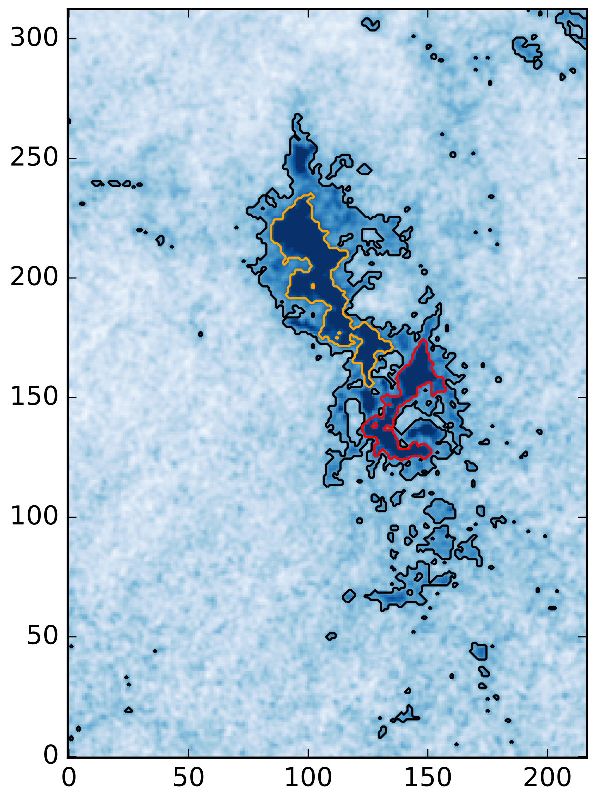
```
import matplotlib.pyplot as plt
from astropy.io import fits
from astrodendro import Dendrogram

image = fits.getdata('PerA_Extn2MASS_F_Gal.fits')
d = Dendrogram.compute(image, min_value=2.0, min_delta=1., min_npix=10)
p = d.plotter()

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

# Plot the whole tree
p.plot_tree(ax, color='black')

# Highlight two branches
p.plot_tree(ax, structure=8, color='red', lw=2, alpha=0.5)
p.plot_tree(ax, structure=24, color='orange', lw=2, alpha=0.5)

# Add axis labels
ax.set_xlabel("Structure")
ax.set_ylabel("Flux")
```



You can find out the structure ID you need either from the interactive viewer presented above, or programmatically by accessing the `idx` attribute of a Structure.

A `plot_contour()` method is also provided to outline the contours of structures. Calling `plot_contour()` without any arguments results in a contour corresponding to the value of `min_value` used being shown.

```
import matplotlib.pyplot as plt
from astropy.io import fits
from astrodendro import Dendrogram

image = fits.getdata('PerA_Extn2MASS_F_Gal.fits')
d = Dendrogram.compute(image, min_value=2.0, min_delta=1., min_npix=10)
p = d.plotter()

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.imshow(image, origin='lower', interpolation='nearest', cmap=plt.cm.Blues, vmax=4.0)

# Show contour for ``min_value``
p.plot_contour(ax, color='black')

# Highlight two branches
p.plot_contour(ax, structure=8, lw=3, colors='red')
p.plot_contour(ax, structure=24, lw=3, colors='orange')
```



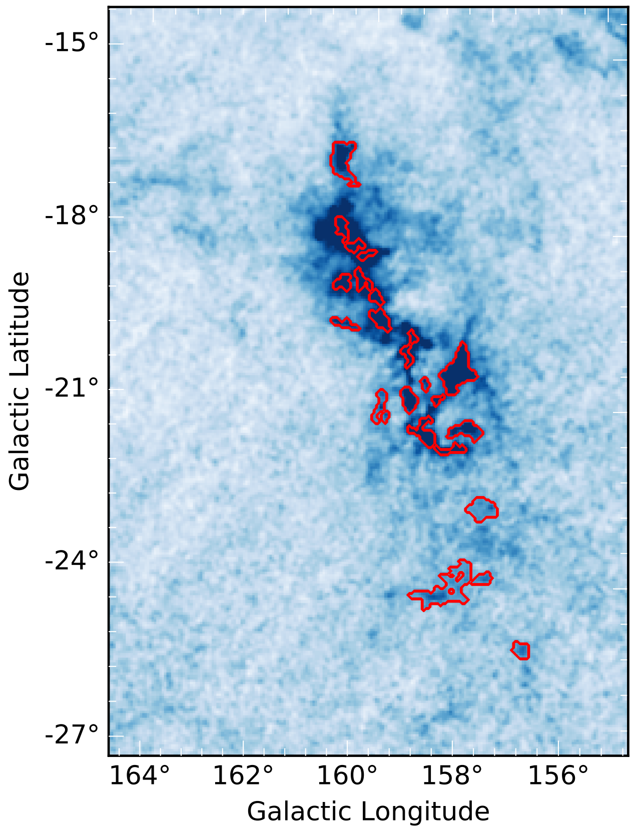
2.4.3 Plotting contours of structures in third-party packages

In some cases you may want to plot the contours in third party packages such as APLpy or DS9. For these cases, the best approach is to output FITS files with a mask of the structures to plot (one mask file per contour color you want to show).

Let's first take the plot above and make a contour plot in APLpy outlining all the leaves. We can use the *get_mask()* method to retrieve the footprint of a given structure:

```
import aplpy
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits
from astrodendro import Dendrogram

hdu = fits.open('PerA_Extn2MASS_F_Gal.fits')[0]
d = Dendrogram.compute(hdu.data, min_value=2.0, min_delta=1., min_npix=10)

# Create empty mask. For each leaf we do an 'or' operation with the mask so
# that any pixel corresponding to a leaf is set to True.
mask = np.zeros(hdu.data.shape, dtype=bool)
for leaf in d.leaves:
    mask = mask | leaf.get_mask()

# Now we create a FITS HDU object to contain this, with the correct header
```

```
mask_hdu = fits.PrimaryHDU(mask.astype('short'), hdu.header)

# We then use APLpy to make the final plot
fig = aplpy.FITSFigure(hdu, figsize=(8, 6))
fig.show_colorscale(cmap='Blues', vmax=4.0)
fig.show_contour(mask_hdu, colors='red', linewidths=0.5)
fig.tick_labels.set_xformat('dd')
fig.tick_labels.set_yformat('dd')
```



Now let's take the example from *Making plots for publications* and try and reproduce the same plot. As described there, one way to find interesting structures in the dendrogram is to use the *Interactive Visualization* tool. This tool will give the ID of a structure as an integer (`idx`).

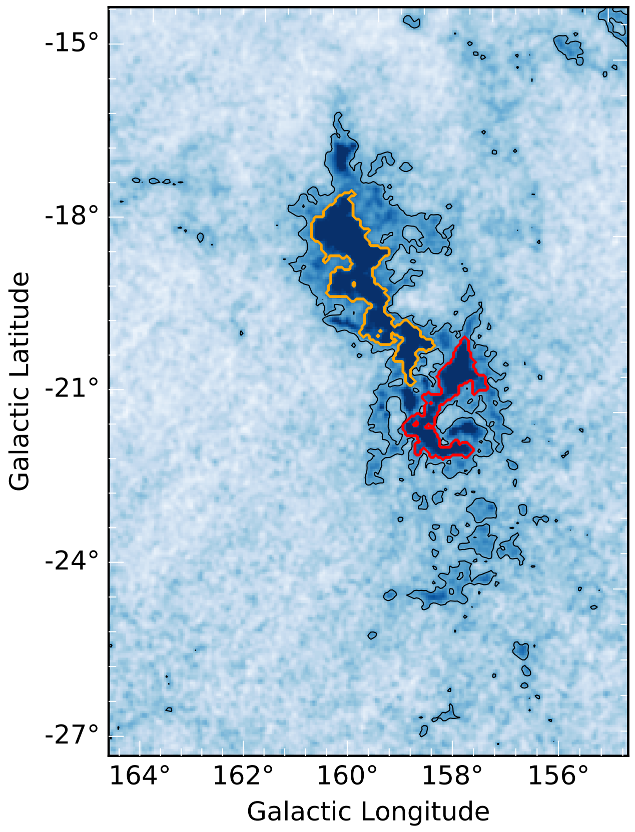
Because we are starting from this ID rather than a *`Structure`* object, we need to first get the structure, which can be done with:

```
structure = d[idx]
```

where `d` is a *`Dendrogram`* instance. We also want to create a different mask for each contour so as to have complete control over the colors:

```
import aplpy
from astropy.io import fits
from astrodendro import Dendrogram

hdu = fits.open('PerA_Extn2MASS_F_Gal.fits')[0]
d = Dendrogram.compute(hdu.data, min_value=2.0, min_delta=1., min_npix=10)
```

```
# Find the structures
structure_08 = d[8]
structure_24 = d[24]

# Extract the masks
mask_08 = structure_08.get_mask()
mask_24 = structure_24.get_mask()

# Create FITS HDU objects to contain the masks
mask_hdu_08 = fits.PrimaryHDU(mask_08.astype('short'), hdu.header)
mask_hdu_24 = fits.PrimaryHDU(mask_24.astype('short'), hdu.header)

# Use APLpy to make the final plot
fig = aplpy.FITSFigure(hdu, figsize=(8, 6))
fig.show_colorscale(cmap='Blues', vmax=4.0)
fig.show_contour(hdu, levels=[2.0], colors='black', linewidths=0.5)
fig.show_contour(mask_hdu_08, colors='red', linewidths=0.5)
fig.show_contour(mask_hdu_24, colors='orange', linewidths=0.5)
fig.tick_labels.set_xformat('dd')
fig.tick_labels.set_yformat('dd')
```



2.5 Computing Dendrogram Statistics

For 2D position-position (PP) and 3D position-position-velocity (PPV) observational data, the astrodendro.analysis module can be used to compute basic properties for each Dendrogram structure. There are two ways to compute statistics - on a structure-by-structure basis and as a catalog, both of which are described below.

2.5.1 Deriving statistics for individual structures

In order to derive statistics for a given structure, you will need to use the *PPStatistic* or the *PPVStatistic* classes from the astrodendro.analysis module, e.g.:

```
>>> from astrodendro.analysis import PPStatistic
>>> stat = PPStatistic(structure)
```

where `structure` is a *Structure* instance from a dendrogram. The resulting object then has methods to compute various statistics. Using the example data from Computing and exploring dendrograms:

```
>>> from astrodendro import Dendrogram
>>> from astropy.io import fits
>>> image = fits.getdata('PerA_Extn2MASS_F_Gal.fits')
>>> d = Dendrogram.compute(image, min_value=2.0, min_delta=1., min_npix=10)
```

we can get statistics for the first structure in the trunk, which is a leaf:

```
>>> from astrodendro.analysis import PPStatistic
>>> d.trunk[0]
<Structure type=leaf idx=101>
>>> stat = PPStatistic(d.trunk[0])
>>> stat.major_sigma  # length of major axis on the sky
<Quantity 1.882980574564531 pix>
>>> stat.minor_sigma  # length of minor axis on the sky
<Quantity 1.4639300383020182 pix>
>>> stat.position_angle  # position angle on the sky
<Quantity 134.61988014787443 deg>
```

Note: The objects returned are Astropy `Quantity` objects, which are Numpy scalars or arrays with units attached. For more information, see the Astropy Documentation. In most cases, you should be able to use these objects directly, but if for any reason you need to access the underlying value, then you can do so with the `value` and `unit` attributes:

```
>>> q = 1.882980574564531 * u.pix
>>> q.unit
Unit("pix")
>>> q.value
1.882980574564531
```

2.5.2 Specifying meta-data when computing statistics

In some cases, meta-data can or should be specified. To demonstrate this, we will use a different data set which is a small section (`L1551_scuba_850mu.fits`) of a SCUBA 850 micron map from the SCUBA Legacy Catalog. This map has a pixel scale of 6 arcseconds per pixel and a circular beam with a full-width at half maximum (FWHM) of 22.9 arcseconds. First, we compute the dendrogram as usual:

```
>>> from astropy.io import fits
>>> from astrodendro import Dendrogram
>>> image = fits.getdata('L1551_scuba_850mu.fits')
>>> d = Dendrogram.compute(image, min_value=0.1, min_delta=0.02)
```

then we set up a Python dictionary containing the required meta-data:

```
>>> from astropy import units as u
>>> metadata = {}
>>> metadata['data_unit'] = u.Jy / u.beam
>>> metadata['spatial_scale'] =  6 * u.arcsec
>>> metadata['beam_major'] =  22.9 * u.arcsec # FWHM
>>> metadata['beam_minor'] =  22.9 * u.arcsec # FWHM
```

Note that the meta-data required depends on the units of the data and whether you are working in position-position or position-position-velocity (see *Required metadata*).

Finally, as before, we use the `PPStatistic` class to extract properties for the first structure:

```
>>> from astrodendro.analysis import PPStatistic
>>> stat = PPStatistic(d.trunk[0], metadata=metadata)
>>> stat.major_sigma
<Quantity 20.34630778380526 arcsec>
>>> stat.minor_sigma
<Quantity 8.15504176035544 arcsec>
>>> stat.position_angle
<Quantity 85.14309012311242 deg>
>>> stat.flux
<Quantity 0.24119688679751278 Jy>
```

Note that the major and minor sigma on the sky of the structures are now in arcseconds since the spatial scale was specified, and the flux (density) has been converted from Jy/beam to Jy. Note also that the flux does not include any kind of background subtraction, and is just a plain sum of the values in the structure, converted to Jy

2.5.3 Making a catalog

In order to produce a catalog of properties for all structures, it is also possible to make use of the `pp_catalog()` and `ppv_catalog()` functions. We demonstrate this using the same SCUBA data as used above:

```
>>> from astropy.io import fits
>>> from astrodendro import Dendrogram, pp_catalog
>>> image = fits.getdata('L1551_scuba_850mu.fits')
>>> d = Dendrogram.compute(image, min_value=0.1, min_delta=0.02)

>>> from astropy import units as u
>>> metadata = {}
>>> metadata['data_unit'] = u.Jy / u.beam
>>> metadata['spatial_scale'] =  6 * u.arcsec
>>> metadata['beam_major'] =  22.9 * u.arcsec
>>> metadata['beam_minor'] =  22.9 * u.arcsec

>>> cat = pp_catalog(d, metadata)
>>> cat.pprint(show_unit=True, max_lines=10)
_idx      flux       major_sigma   minor_sigma  ...    radius        x_cen         y_cen
          Jy           arcsec        arcsec     ...    arcsec         pix           pix
---- -------------- ------------- ------------- ... ------------- ------------- -------------
                                                ...
   7  0.241196886798 20.3463077838 8.15504176036 ... 12.8811874315 168.053017504 3.98809714744
  51  0.132470059814 14.2778133293 4.81100492125 ...  8.2879810685  163.25495657 9.13394216473
  60 0.0799106574322 9.66298008473 3.47364264736 ... 5.79359471511 169.278409915 15.1884110291
 ...             ...           ...           ... ...           ...           ...           ...
1203  0.183438198239 22.7202518034 4.04690367115 ... 9.58888264776 15.3760934458 100.136384362
1384   2.06217635837 38.1060171889  19.766115194 ... 27.4446338168 136.429313911 107.190835447
1504   1.90767291972 8.64476839751 8.09070477357 ... 8.36314946298  68.818705665 120.246719845
```

The catalog function returns an Astropy `Table` object.

Note that *pp_catalog()* and *ppv_catalog()* generate warnings if required meta-data is missing and sensible defaults can be assumed. If no sensible defaults can be assumed (e.g. for `data_unit`) then an exception is raised.

Unlike clumpfind-style algorithms, there is not a one-to-one mapping between identifiers and pixels in the map: each pixel may belong to multiple nested branches in the catalog.

2.5.4 Available statistics

For a full list of available statistics for each type of statistic class, see *PPStatistic* and *PPVStatistic*. For more information on the quantities listed in these pages, consult the paper on Bias Free Measurements of Molecular Cloud Properties or the original dendrogram paper. In the terminology of the dendrogram paper, the quantities in *PPStatistic* and *PPVStatistic* adopt the “bijection” paradigm.

2.5.5 Required metadata

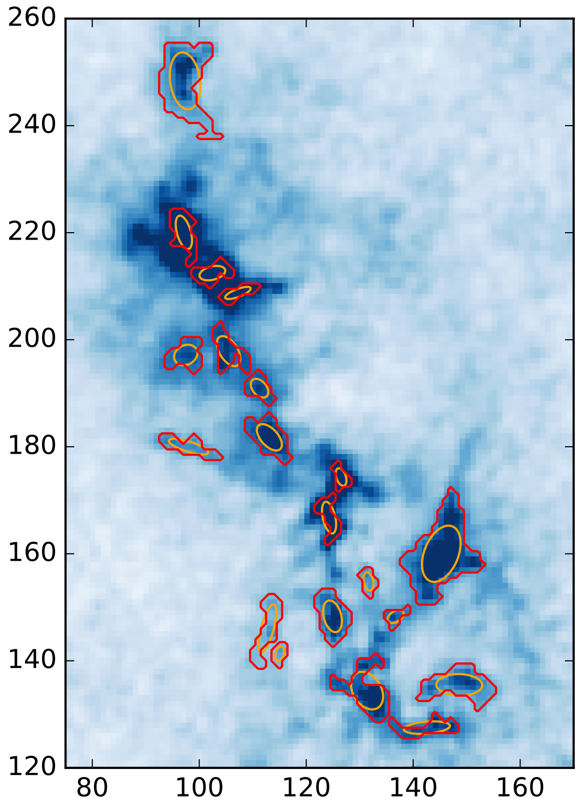
As described above, the metadata needed by the statistic routines depends on what statistics are required and on the units of the data. With the exception of `wcs`, all meta-data should be specified as Astropy Quantity objects (e.g., `3 * u.arcsec`):

- `data_unit` is **required** in order to compute the flux, so it is needed for both the *pp_catalog()* and *ppv_catalog()* functions, as well as for the `flux` attribute of the *PPStatistic* and *PPVStatistic* classes. Note: if `data_unit` is given as `K`, it is interpreted as units of main beam brightness temperature, following the conventions in the astropy Brightness Temperature / Flux Density equivalency .
- `spatial_scale` is **required** if the data are in units of surface brightness (e.g. `MJy/sr`, `Jy/beam`, or `K`) so as to be able to convert the surface brightness to the flux in each pixel. Even if the data are not in units of surface brightness, the `spatial_scale` can **optionally** be specified, causing any derived size (e.g. `major_sigma`) to be in the correct units instead of in pixels.
- `velocity_scale` can **optionally** be specified for PPV data, causing `v_rms` to be in the correct units instead of in pixels.
- `beam_major` and `beam_minor` are **required** if the data units depend on the beam (e.g. `Jy/beam` or `K`).
- `vaxis` can **optionally** be specified when using 3-dimensional data to indicate which dimension corresponds to the velocity. By default, this is `0`, which corresponds to the third axis in a FITS file (because the dimensions are reversed in numpy).
- `wavelength` is **required** if the data are in monochromatic flux densities per unit wavelength because the fluxes need to be converted to monochromatic flux densities per unit frequency. It is also required if the data are in brightness temperature units of `K`.
- `wcs` can **optionally** be specified and should be a `WCS` instance. If specified, it allows `x_cen`, `y_cen`, and `v_cen` to be in the correct world coordinates rather than in pixel coordinates.

2.5.6 Example

The following example shows how to combine the plotting functionality in Plotting Dendrograms and the analysis tools shown above, to overlay ellipses approximating the structures on top of the structures themselves:

```
from astropy.io import fits

from astrodendro import Dendrogram
from astrodendro.analysis import PPStatistic

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

hdu = fits.open('PerA_Extn2MASS_F_Gal.fits')[0]

d = Dendrogram.compute(hdu.data, min_value=2.0, min_delta=1., min_npix=10)
p = d.plotter()

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

ax.imshow(hdu.data, origin='lower', interpolation='nearest',
          cmap=plt.cm.Blues, vmax=6.0)

for leaf in d.leaves:

    p.plot_contour(ax, structure=leaf, lw=3, colors='red')

    s = PPStatistic(leaf)
    ellipse = s.to_mpl_ellipse(edgecolor='orange', facecolor='none')

    ax.add_patch(ellipse)

ax.set_xlim(75., 170.)
ax.set_ylim(120., 260.)
```



As shown above, the *PPStatistic* and *PPVStatistic* classes have a *to_mpl_ellipse()* method to convert the first and second moments of the structures into schematic ellipses.

2.6 Advanced topics

2.6.1 Specifying a custom structure merging strategy

By default, the decision about whether a leaf remains independent (i.e., whether it remains a leaf or its pixels get incorporated into another branch) when merged is made based on the `min_delta` and `min_npix` parameters, but in some cases, you may want to use more specialized criteria. For example, you may want only leaves overlapping with a certain position, or you may want leaves with a certain spatial or velocity extent, or a minimum peak value, to be considered independent structures.

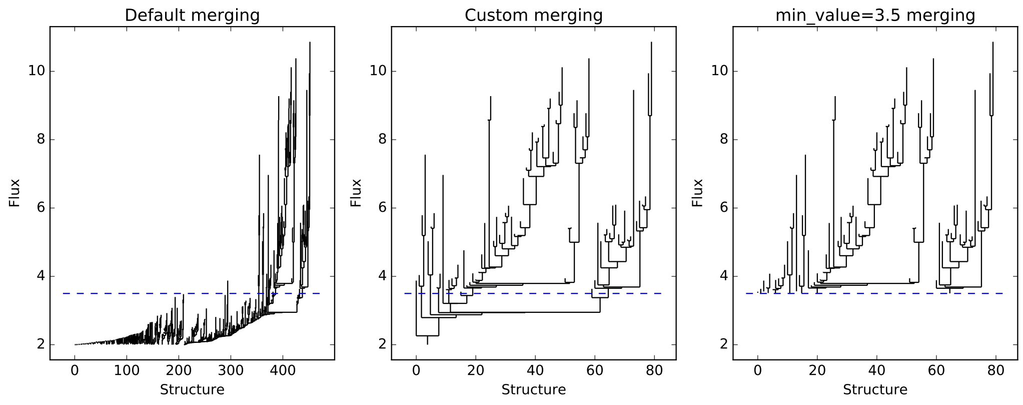
In order to accomodate this, the *compute()* method can optionally take an `is_independent` argument which should be a function with the following call signature:

```
def is_independent(structure, index=None, value=None):
    ...
```

where `structure` is the *Structure* object that is being considered, and `index` and `value` are the pixel index and value of the pixel that is linking the structure to the rest of the tree. These last two values are only set when calling the `is_independent` function during the tree computation, but the `is_independent` function is also used at the end of the computation to prune leaves that are not attached to the tree, and in this case `index` and `value` are not set.

The following example compares the dendrogram obtained with and without a custom `is_independent` function:

```
import matplotlib.pyplot as plt
from astropy.io import fits
from astrodendro import Dendrogram

image = fits.getdata('PerA_Extn2MASS_F_Gal.fits')

fig = plt.figure(figsize=(15,5))

# Default merging strategy

d1 = Dendrogram.compute(image, min_value=2.0)
p1 = d1.plotter()

ax1 = fig.add_subplot(1, 3, 1)
p1.plot_tree(ax1, color='black')
ax1.hlines(3.5, *ax1.get_xlim(), color='b', linestyle='--')
ax1.set_xlabel("Structure")
ax1.set_ylabel("Flux")
ax1.set_title("Default merging")

# Require minimum peak value
# this is equivalent to
# custom_independent = astrodendro.pruning.min_peak(3.5)
def custom_independent(structure, index=None, value=None):
    peak_index, peak_value = structure.get_peak()
    return peak_value > 3.5

d2 = Dendrogram.compute(image, min_value=2.0,
                        is_independent=custom_independent)
p2 = d2.plotter()

ax2 = fig.add_subplot(1, 3, 2)
p2.plot_tree(ax2, color='black')
ax2.hlines(3.5, *ax2.get_xlim(), color='b', linestyle='--')
ax2.set_xlabel("Structure")
ax2.set_ylabel("Flux")
ax2.set_title("Custom merging")

# For comparison, this is what changing the min_value does:
d3 = Dendrogram.compute(image, min_value=3.5)
p3 = d3.plotter()

ax3 = fig.add_subplot(1, 3, 3)
p3.plot_tree(ax3, color='black')
ax3.hlines(3.5, *ax3.get_xlim(), color='b', linestyle='--')
ax3.set_xlabel("Structure")
ax3.set_ylabel("Flux")
ax3.set_title("min_value=3.5 merging")
ax3.set_ylim(*ax2.get_ylim())
```



Several pre-implemented functions suitable for use as `is_independent` tests are provided in *`astrodendro.pruning`*. In addition, the *`astrodendro.pruning.all_true()`* function can be used to combine several criteria. For example, the following code builds a dendrogram where each leaf contains a pixel whose value >=20, and whose pixels sum to >= 100:

```
from astrodendro.pruning import all_true, min_peak, min_sum

custom_independent = all_true((min_peak(20), min_sum(100)))
Dendrogram.compute(image, is_independent=custom_independent)
```

2.6.2 Handling custom adjacency logic

By default, neighbours to a given pixel are considered to be the adjacent pixels in the array. However, not all data are like this. For example, all-sky cartesian maps are periodic along the X axis.

You can specify custom neighbour logic by providing a `neighbours` function to *`Dendrogram.compute()`*. For example, the *`periodic_neighbours()`* utility will wrap neighbours across array edges. To correctly compute dendrograms for all-sky Cartesian maps:

```
periodic_axis = 1  # data wraps along longitude axis
Dendrogram.compute(data, neighbours=periodic_neighbours(periodic_axis))
```

2.7 Migration guide for previous users of `astrodendro`

The `astrodendro` package has been in development for a couple of years, and we have recently undertaken an effort to prepare the package for a first release, which involved tidying up the programming interface to the package, and re-writing large sections. This means that the present version of `astrodendro` will likely not work with scripts you had if you were using the original astrodendro packages from @astrofrog and @brandenmacdonald's repositories.

This page summarizes the main changes in the new code, and how to adapt your code to ensure that it will work correctly. This only covers changes that will *break* your code, but you are encouraged to look through the rest of the documentation to read about new features! Also, only the main backward-incompatible changes are mentioned, but for any questions on changes not mentioned here, please open an issue on GitHub.

2.7.1 Computing a dendrogram

Rather than computing a dendrogram using:

```
d = Dendrogram(data)
d.compute(...)
```

you should now use:

```
d = Dendrogram.compute(data)
```

In addition, the following options for `compute` have been renamed:

- `minimum_flux` is now `min_value` (since we expect dendrograms to be used not only for images, but also e.g. density fields).
- `minimum_delta` is now `min_delta`
- `minimum_npix` is now `min_npix`

2.7.2 Dendrogram methods and attributes

The following dendrogram methods have changed:

- `get_leaves()` has now been replaced by a `leaves` attribute (it is no longer a method.)
- the `to_hdf5()` and `from_hdf5()` methods have been replaced by *`save_to()`*

2.7.3 `Leaf` and `Branch` classes

The `Leaf` and `Branch` classes no longer exist, and have been replaced by a single *`Structure`* class that instead has `is_leaf` and `is_branch` attributes. Thus, if you were checking if something was a leaf by doing e.g.:

```
if type(s) == Leaf:
    # code here
```

or:

```
if isinstance(s, Leaf):
     # code here
```

then you will instead need to use:

```
if s.is_leaf:
     # code here
```

2.7.4 Leaf and branch attributes

The following leaf and branch attributes have changed:

- `f` has been replaced by a method called *`values()`* that can take a `subtree=` option that indicates whether pixels in sub-structures should be included.
- `coords` has been replaced by a method called *`indices()`* that can take a `subtree=` option that indicates whether pixels in sub-structures should be included.
- `height` now has a different definition - it is `vmax` for a leaf, or the smallest `vmin` of the children for a branch - this is used when plotting the dendrogram, to know at what height to plot the structure.

2.7.5 Interactive visualization

Visualizing the results of the dendrogram is now much easier, and does not require the additional `astrocube` package. To launch the interactive viewer (which requires only Matplotlib), once the dendrogram has been computed, you can do:

```
>>> d.viewer()
```

and the interactive viewer will launch. It will however no longer have the option to re-compute the dendrogram from the window, and will also no longer have an IPython terminal. For the latter, we recommend you consider using the Glue package.

CHAPTER 3

Reporting issues and getting help

Please help us improve this package by reporting issues via GitHub. You can also open an issue if you need help with using the package.

CHAPTER 4

Developers

This package was developed by:

- Thomas Robitaille
- Chris Beaumont
- Adam Ginsburg
- Braden MacDonald
- Erik Rosolowsky

CHAPTER 5

Acknowledgments

Thanks to the following users for using early versions of this package and providing valuable feedback:

- Katharine Johnston

CHAPTER 6

Citing astrodendro

If you make use of this package in a publication, please consider adding the following acknowledgment:

This research made use of astrodendro, a Python package to compute dendrograms of Astronomical data (http://www.dendrograms.org/)

If you make use of the analysis code (Computing Dendrogram Statistics) or read/write FITS files, please also consider adding an acknowledgment for Astropy (see http://www.astropy.org for the latest recommended citation).

CHAPTER 7

Public API

7.1 astrodendro.dendrogram.Dendrogram

class `astrodendro.dendrogram.`**`Dendrogram`**

This class is used to compute and represent a dendrogram for a given dataset.

To create a dendrogram from an array, use the `compute()` class method:

```
>>> from astrodendro import Dendrogram
>>> d = Dendrogram.compute(array)
```

Once the dendrogram has been computed, you can explore it programmatically using the `trunk` attribute, which allows you to access the base-level structures in the dendrogram:

```
>>> d.trunk
[<Structure type=leaf idx=101>,
 <Structure type=branch idx=2152>,
 <Structure type=leaf idx=733>,
 <Structure type=branch idx=303>]
```

Structures can then be recursively explored. For more information on attributes and methods available for structures, see the `Structure` class.

The dendrogram can also be explored using an interactive viewer. To use this, use the `viewer()` method:

```
>>> d.viewer()
```

and an interactive Matplotlib window should open.

Finally, the `plotter()` method can be used to facilitate the creation of plots:

```
>>> p = d.plotter()
```

For more information on using the plotter and other aspects of the `Dendrogram` class, see the online documentation.

Attributes

`trunk`	A list of all structures that have no parent structure and form the base of the tree.
`leaves`	A flattened list of all leaves in the dendrogram.
`all_structures`	Yields an iterator over all structures in the dendrogram, in prefix order.

Analysis

`compute`(data[, min_value, min_delta, ...])	Compute a dendrogram from a Numpy array.
`structure_at`(indices)	Get the structure at the specified pixel coordinate.

Input/Output

`save_to`(filename[, format])	Save the dendrogram to a file.
`load_from`(filename[, format])	Load a previously computed dendrogram from a file.

Visualization

`plotter`()	Return a `DendrogramPlotter` instance that makes it easier to construct plots.
`viewer`()	Launch an interactive viewer to explore the dendrogram.

Methods (detail)

static compute (*data*, *min_value='min'*, *min_delta=0*, *min_npix=0*, *is_independent=None*, *verbose=False*, *neighbours=None*, *wcs=None*)

Compute a dendrogram from a Numpy array.

Parameters data : `numpy.ndarray`

The n-dimensional array to compute the dendrogram for

min_value : float or "min", optional

The minimum data value to go down to when computing the dendrogram. Values below this threshold will be ignored. Defaults to the minimum value in the data.

min_delta : float, optional

The minimum height a leaf has to have in order to be considered an independent entity.

min_npix : int, optional

The minimum number of pixels/values needed for a leaf to be considered an independent entity.

is_independent : function or list of functions, optional

A custom function that can be specified that will determine if a leaf can be treated as an independent entity. The signature of the function should be `func(structure, index=None, value=None)` where `structure` is the structure under consideration, and `index` and `value` are optionally the pixel that is causing the structure to be considered for merging into/attaching to the tree.

If multiple functions are provided as a list, they are all applied when testing for independence.

neighbours : function, optional

A function that returns the list of neighbours to a given location. Neighbours is called as `neighbours(dendrogram, idx)`, where `idx` is a tuple describing the n-dimensional location of a pixel. It returns a list of N-dimensional locations of neighbours. This function can implement optional adjacency logic.

Note: `idx` refers to location in a copy of the input data that has been padded with one element along each edge.

---

wcs : WCS object, optional

A WCS object that describes *data*. This is used in the interactive viewer to properly display the data's coordinates on the image axes. (Requires that *wcsaxes* is installed; see http://wcsaxes.readthedocs.org/ for install instructions.)

Notes

More information about the above parameters is available from the online documentation at www.dendrograms.org.

Examples

The following example demonstrates how to compute a dendrogram from an dataset contained in a FITS file:

```
>>> from astropy.io import fits
>>> array = fits.getdata('observations.fits')
>>> from astrodendro import Dendrogram
>>> d = Dendrogram.compute(array)
```

`structure_at` (*indices*)

Get the structure at the specified pixel coordinate.

This will return None if no structure includes the specified pixel coordinates.

Parameters **indices: tuple**

The pixel coordinates of the structure of interest

`save_to` (*filename*, *format=None*)

Save the dendrogram to a file.

Parameters **filename** : str

The name of the file to save the dendrogram to. By default, the file format will be automatically detected from the file extension. At this time, only HDF5 files (extension `.hdf5`) are supported.

format : str, optional

The format to use for the file. By default, this is not used and the format is auto-detected from the file extension. At this time, the only format supported is `'hdf5'`.

static **`load_from`** (*filename*, *format=None*)

Load a previously computed dendrogram from a file.

Parameters **filename** : str

The name of the file to load the dendrogram from. By default, the file format will be automatically detected from the file extension. At this time, only HDF5 files (extension `.hdf5`) are supported.

format : str, optional

The format to use to read the file. By default, this is not used and the format is auto-detected from the file extension. At this time, the only format supported is `'hdf5'`.

`plotter``()`
Return a *`DendrogramPlotter`* instance that makes it easier to construct plots.

`viewer``()`
Launch an interactive viewer to explore the dendrogram.

This functionality is only available for 2- or 3-d datasets.

7.2 astrodendro.dendrogram.periodic_neighbours

`astrodendro.dendrogram.`**`periodic_neighbours`**`(`*axes*`)`
Utility for computing neighbours on datasets with periodic boundaries.

This can be passed to the neighbours keyword of *`Dendrogram.compute()`*

Parameters **axes** : integer, or list of integers

Which axes of the data are periodic

Examples

Build a dendrogram where the 0th axis wraps from top-to-bottom:

```
Dendrogram.compute(data, neighbours=periodic_neighbours(0))
```

7.3 astrodendro.structure.Structure

class `astrodendro.structure.`**`Structure`**`(`*indices*, *values*, *children=[]*, *idx=None*, *dendrogram=None*`)`
A structure in the dendrogram, for example a leaf or a branch.

A structure that is part of a dendrogram knows which other structures it is related to. For example, it is possible to get the parent structure containing the present structure `s` by using the `parent` attribute:

```
>>> s.parent
<Structure type=branch idx=2152>
```

Likewise, the `children` attribute can be used to get a list of all sub-structures:

```
>>> s.children
[<Structure type=branch idx=1680>, <Structure type=branch idx=5771>]
```

A number of attributes and methods are available to explore the structure in more detail, such as the `indices` and `values` methods, which return the indices and values of the pixels that are part of the structure. These and other methods have a `subtree=` option, which if `True` (the default) returns the quantities related to structure and all sub-structures, and if `False` includes only the pixels that are part of the structure, but excluding any sub-structure.

Attributes

`is_leaf`	Whether the present structure is a leaf.
`is_branch`	Whether the present structure is a branch.
`vmin`	The minimum value of pixels belonging to the branch (excluding sub-structure).
`vmax`	The maximum value of pixels belonging to the branch (excluding sub-structure).
`height`	This is defined as the minimum value in the children structures, or the peak value of the present structure if it has n
`ancestor`	Find the ancestor of this leaf/branch non-recursively.
`parent`	The parent structure containing the present structure.
`children`	A list of all the sub-structures contained in the present structure.
`descendants`	Get a flattened list of all child leaves and branches.
`level`	The level of the structure, i.e.
`newick`	

Methods

`indices`([subtree])	The indices of the pixels in this branch.
`values`([subtree])	The values of the pixels in this branch.
`get_npix`([subtree])	Return the number of pixels in this structure.
`get_peak`([subtree])	Return (index, value) for the pixel with maximum value.
`sorted_leaves`([sort_key, reverse, subtree])	Return a list of sorted leaves.
`get_mask`([shape, subtree])	Return a boolean mask outlining the structure.

Methods (detail)

`indices`(*subtree=True*)
The indices of the pixels in this branch.

Parameters **subtree** : bool, optional

Whether to recursively include all sub-structures

`values`(*subtree=True*)
The values of the pixels in this branch.

Parameters **subtree** : bool, optional

Whether to recursively include all sub-structures

`get_npix`(*subtree=True*)
Return the number of pixels in this structure.

Parameters **subtree** : bool, optional

Whether to recursively include all sub-structures when counting the pixels.

Returns **n_pix** : int

The number of pixels in this structure

`get_peak`(*subtree=True*)
Return (index, value) for the pixel with maximum value.

Parameters **subtree** : bool, optional

Whether to recursively include all sub-structures when searching for the peak.

Returns **index** : tuple

The n-dimensional index of the peak pixel

value : float

The value of the peak pixel

sorted_leaves (*sort_key=<function <lambda>>*, *reverse=False*, *subtree=True*)
Return a list of sorted leaves.

Parameters **sort_key** : function, optional

A function which given a structure will return a scalar that is then used for sorting. By default, this is set to a function that returns the peak value of a structure (including descendants).

reverse : bool, optional

Whether to reverse the sorting.

subtree : bool, optional

Whether to recursively include all sub-structures in the list.

Returns **leaves** : list

A list of sorted leaves

get_mask (*shape=None*, *subtree=True*)
Return a boolean mask outlining the structure.

Parameters **shape** : tuple, optional

The shape of the array upon which to compute the mask. This is only required if the structure is not attached to a dendrogram.

subtree : bool, optional

Whether to recursively include all sub-structures in the mask.

Returns **mask** : `ndarray`

The mask outlining the structure (`False` values are used outside the structure, and `True` values inside).

7.4 astrodendro.plot.DendrogramPlotter

class `astrodendro.plot.`**`DendrogramPlotter`**(*dendrogram*)
A class to plot a dendrogram object.

Methods

sort([sort_key, reverse])	Sort the position of the leaves for plotting.
set_custom_positions(custom_position)	Manually set the positon on the structures for plotting.
plot_tree(ax[, structure, subtree, autoscale])	Plot the dendrogram tree or a substructure.
plot_contour(ax[, structure, subtree, slice])	Plot a contour outlining all pixels in the dendrogram, or a specific.
get_lines([structures, subtree])	Get a collection of lines to draw the dendrogram.

Methods (detail)

`sort`(*sort_key=None*, *reverse=False*)

Sort the position of the leaves for plotting.

Parameters **sort_key** : function, optional

This should be a function that takes a *~astrodendro.structure.Structure* and returns a scalar that is then used to sort the leaves. If not specified, the leaves are sorted according to their peak value.

reverse : bool, optional

Whether to reverse the sorting

`set_custom_positions`(*custom_position*)

Manually set the positon on the structures for plotting.

Parameters **custom_position** : function

This should be a function that takes a *~astrodendro.structure.Structure'returns the position of the leaves to use for plotting. If the dataset has more than one dimension, using this may cause lines to cross. If this is used, then ''sort_key'* and `reverse` are ignored.

`plot_tree`(*ax*, *structure=None*, *subtree=True*, *autoscale=True*, ***kwargs*)

Plot the dendrogram tree or a substructure.

Parameters **ax** : `Axes` instance

The Axes inside which to plot the dendrogram

structure : int or *~astrodendro.structure.Structure*, optional

If specified, only plot this structure. This can be either the structure object itself, or the ID (`idx`) of the structure.

subtree : bool, optional

If a structure is specified, by default the whole subtree will be plotted, but this can be disabled with this option.

autoscale : bool, optional

Whether to automatically adapt the window limits to the tree

Notes

Any additional keyword arguments are passed to *~matplotlib.collections.LineCollection* and can be used to control the appearance of the plot.

`plot_contour`(*ax*, *structure=None*, *subtree=True*, *slice=None*, ***kwargs*)

Plot a contour outlining all pixels in the dendrogram, or a specific. structure.

Parameters **ax** : `Axes` instance

The Axes inside which to plot the dendrogram

structure : int or *~astrodendro.structure.Structure*, optional

If specified, only plot this structure. This can be either the structure object itself, or the ID (`idx`) of the structure.

subtree : bool, optional

If a structure is specified, by default the whole subtree will be plotted, but this can be disabled with this option.

slice : int, optional

If dealing with a 3-d cube, the slice at which to plot the contour. If not set, the slice containing the peak of the structure will be shown

Notes

Any additional keyword arguments are passed to *~matplotlib.axes.Axes.contour* and can be used to control the appearance of the plot.

get_lines (*structures=None*, *subtree=True*, ***kwargs*)

Get a collection of lines to draw the dendrogram.

Parameters **structures** : *Structure*

The structures to plot. If not set, the whole tree will be plotted.

subtree : bool, optional

If a structure is specified, by default the whole subtree will be retrieved, but this can be disabled with this option.

Returns **lines** : `StructureCollection`

The lines (sub-class of LineCollection) which can be directly used in Matplotlib

Notes

Any additional keyword arguments are passed to the *~matplotlib.collections.LineCollection* class.

7.5 astrodendro.analysis

class `astrodendro.analysis.`**`PPStatistic`** (*stat*, *metadata=None*)

Compute properties of structures in a position-position (PP) cube.

Parameters **structure** : *Structure* instance

The structure to compute the statistics for

metadata : dict

Key-value pairs of metadata

Available statistics

`flux`	The integrated flux of the structure, in Jy (note that this does not include any kind of background su
`major_sigma`	Major axis of the projection onto the position-position (PP) plane, computed from the intensity wei
`minor_sigma`	Minor axis of the projection onto the position-position (PP) plane, computed from the intensity we
`position_angle`	The position angle of sky_maj, sky_min in degrees counter-clockwise from the +x axis.
`radius`	Geometric mean of `major_sigma` and `minor_sigma`.
`area_exact`	The exact area of the structure on the sky.

Table 7.8 – continued from previous page

area_ellipse	The area of the ellipse defined by the second moments, where the semi-major and semi-minor axes
x_cen	The mean position of the structure in the x direction (in pixel coordinates, or in world coordinates i
y_cen	The mean position of the structure in the y direction (in pixel coordinates, or in world coordinates i
to_mpl_ellipse(**kwargs)	Returns a Matplotlib ellipse representing the first and second moments of the structure.

Methods (detail)

PPStatistic.**to_mpl_ellipse**(***kwargs*)

Returns a Matplotlib ellipse representing the first and second moments of the structure.

Any keyword arguments are passed to Ellipse

class astrodendro.analysis.**PPVStatistic**(*stat*, *metadata=None*)

Compute properties of structures in a position-position-velocity (PPV) cube.

Parameters **structure** : *Structure* instance

The structure to compute the statistics for

metadata : dict

Key-value pairs of metadata

Available statistics

flux	The integrated flux of the structure, in Jy (note that this does not include any kind of background su
major_sigma	Major axis of the projection onto the position-position (PP) plane, computed from the intensity wei
minor_sigma	Minor axis of the projection onto the position-position (PP) plane, computed from the intensity we
position_angle	The position angle of sky_maj, sky_min in degrees counter-clockwise from the +x axis (note that t
radius	Geometric mean of major_sigma and minor_sigma.
area_exact	The exact area of the structure on the sky.
area_ellipse	The area of the ellipse defined by the second moments, where the semi-major and semi-minor axes
x_cen	The mean position of the structure in the x direction.
y_cen	The mean position of the structure in the y direction.
v_cen	The mean velocity of the structure (where the velocity axis can be specified by the vaxis metadat
v_rms	Intensity-weighted second moment of velocity (where the velocity axis can be specified by the va
to_mpl_ellipse(**kwargs)	Returns a Matplotlib ellipse representing the first and second moments of the structure.

Methods (detail)

PPStatistic.**to_mpl_ellipse**(***kwargs*)

Returns a Matplotlib ellipse representing the first and second moments of the structure.

Any keyword arguments are passed to Ellipse

astrodendro.analysis.**pp_catalog**(*structures*, *metadata*, *fields=None*, *verbose=True*)

Iterate over a collection of position-position (PP) structures, extracting several quantities from each, and building a catalog.

Parameters **structures** : iterable of Structures

The structures to catalog (e.g., a dendrogram)

metadata : dict

The metadata used to compute the catalog

fields : list of strings, optional

The quantities to extract. If not provided, defaults to all PPV statistics

verbose : bool, optional

If True (the default), will generate warnings about missing metadata

Returns table : a `Table` instance

The resulting catalog

`astrodendro.analysis.`**`ppv_catalog`**`(`*structures*, *metadata*, *fields=None*, *verbose=True*`)`

Iterate over a collection of position-position-velocity (PPV) structures, extracting several quantities from each, and building a catalog.

Parameters structures : iterable of Structures

The structures to catalog (e.g., a dendrogram)

metadata : dict

The metadata used to compute the catalog

fields : list of strings, optional

The quantities to extract. If not provided, defaults to all PPV statistics

verbose : bool, optional

If True (the default), will generate warnings about missing metadata

Returns table : a `Table` instance

The resulting catalog

7.6 astrodendro.pruning

The pruning module provides several functions to perform common pruning via the `is_independent` keyword in the Dendrogram *`compute()`* method.

Examples:

```
#prune unless leaf peak value >= 5
Dendrogram.compute(data, is_independent=min_peak(5))

#prune unless leaf contains 10 pixels
Dendrogram.compute(data, is_independent=min_npix(10))

#apply both criteria
is_independent = all_true((min_peak(5), min_npix(10)))
Dendrogram.compute(data, is_independent=is_independent)
```

`astrodendro.pruning.`**`min_delta`**`(`*delta*`)`

Minimum delta criteria

Parameters delta : float

The minimum height of a leaf above its merger level

`astrodendro.pruning.`**`min_sum`**`(`*sum*`)`

Minimum sum criteria

Parameters **sum** : float

The minimum sum of the pixel values in a leaf

`astrodendro.pruning.`**`min_peak`**`(`*peak*`)`

Minimum peak criteria

Parameters **peak** : float

The minimum peak pixel value in a leaf

`astrodendro.pruning.`**`min_npix`**`(`*npix*`)`

Minimum npix criteria

Parameters **npix** : int

The minimum number of pixels in a leaf

`astrodendro.pruning.`**`contains_seeds`**`(`*seeds*`)`

Critieria that leaves contain at least one of a list of seed positions

Parameters **seeds** : tuple of array-like

seed locations. The ith array in the tuple lists the ith coordinate for each seed. This is the format returned, e.g., by np.where

`astrodendro.pruning.`**`all_true`**`(`*funcs*`)`

Combine several `is_independent` functions into one

Parameters **funcs** : list-like

A list of `is_independent` functions

Returns **combined_func** : function

A new function which returns true of all the input functions are true

Python Module Index

Index