dendro_demo Documentation

Release 0.1

Chris Beaumont

March 21, 2013

CONTENTS

1 What is a Dendrogram? 1

2 Dendrogram Generation with IDL 3
- 2.1 Viewing the Dendrogram . . . 4
- 2.2 Description of Dendrogram Structure Fields . . . 4

3 Dendrogram Generation with C++ 7
- 3.1 Installation . . . 7
- 3.2 Running . . . 7
- 3.3 Reading into IDL . . . 8
- 3.4 Pruning . . . 8

4 Dendrogram Generation with Python 9

5 Recipes 11
- 5.1 Dendrogram Indexes . . . 11
- 5.2 Plotting . . . 13
- 5.3 Analysis . . . 15

6 How to Visualize Dendrograms 17
- 6.1 DendroStar (Java Applet) . . . 17
- 6.2 DendroViz (IDL library) . . . 17

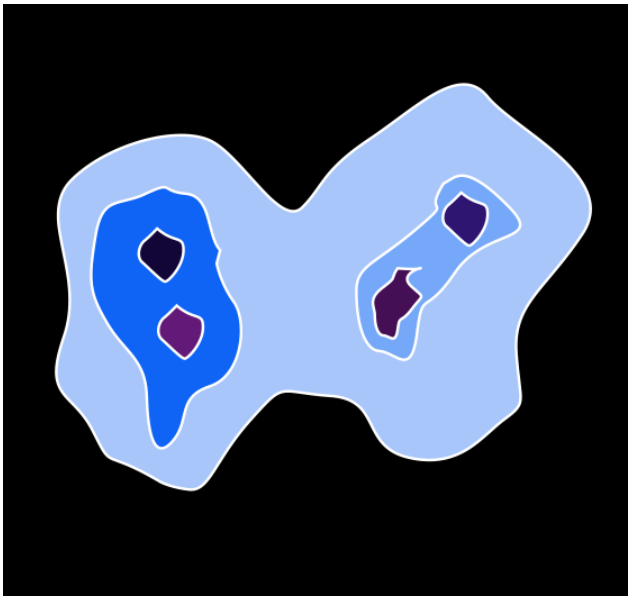
CHAPTER

ONE

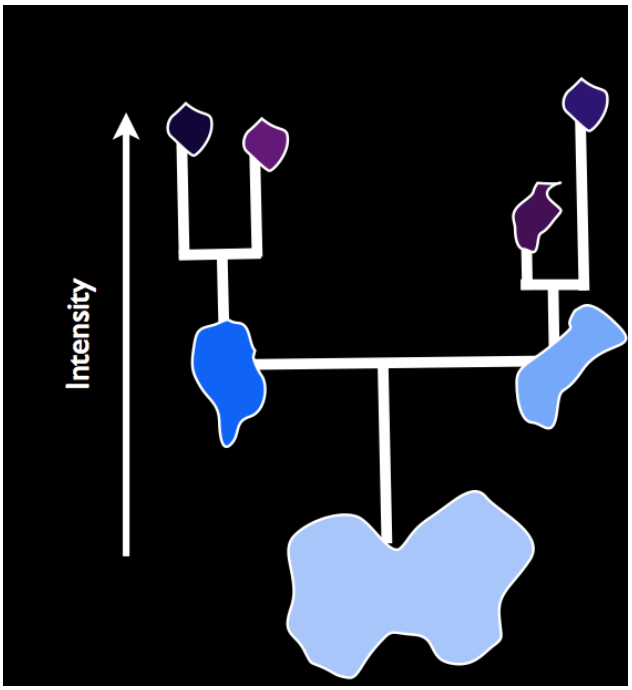
WHAT IS A DENDROGRAM?

In general, a dendrogram is simply a binary tree (i.e. a tree where every node has 0 or 2 children). In the present context, dendrograms are abstract representations of the hierarchical structures in a dataset.

Consider this cartoon, 2-dimensional cloud map:



Imagine drawing several contour lines on this map. Each contour will trace 1 or more closed figures. As we draw contours at increasingly low intensities, these various shapes will grow larger and merge with one another. We can use a dendrogram to represent how these different contour lines merge with one another:



Each point on a vertical line corresponds to a specific, closed contour (the level of that contour is the height of the point). The branching of the tree matches the merger of contours – when two branches join at a height "y", two contour lines merge at a contour level "y".

The procedure works the same way for 3 (or N) dimensional data, but the contour lines become (hyper) surfaces.

CHAPTER

TWO

DENDROGRAM GENERATION WITH IDL

There are currently several codes to generate dendrograms – we are working on unifying these packages. For now, we will look at the original IDL code written by Erik Rosolowsky, available here

Make sure this is in your IDL path

Also download `this file`, which is used in this tutorial.

First, read in the image:

```
im = mrdfits('demo.fits')
```

The code expects a binary mask – an array that has the same shape as the image, where each entry is 1 if that pixel is to be included in the output, and 0 otherwise. For now, lets just use the entire image, and create a mask of all 1s:

```
mask = byte(im * 0) + 1B
```

There are a few other important parameters:

- friends. This is used to control when a pixel is considered a local maximum (i.e., a leaf in the dendrogram). A pixel must be brighter than all of its neighbors within "friends" pixels. If the input is a 3D cube, then friends only applies to the first 2 dimensions.
- specfriends. Just like friends, but applies to the third dimension for 3D cubes.
- delta. The height of each leaf must be at least "delta" above its first merger
- minpix. Every leaf must contain at least minpix pixels before its first merger.
- minpeak. Every leaf must have an intensity of at least minpeak

We will use these values for now:

```
friends = 3
delta = 1.0
minpix = 10
minpeak = 5.0
```

We create the dendrogram with the Topologize procedure:

```
topologize, im, mask, friends = friends, delta = delta, minpix = minpix, minpeak = minpeak, /fast, po
```

This puts the output of the into a pointer variable ptr. For convenience, lets dereference that and store in a normal variable:

```
dendro = *ptr
```

2.1 Viewing the Dendrogram

The cloud-viz package provides some ability to view dendrograms. Simply invoke with:

```
dendroviz, ptr
```

to play around. More coming soon.

2.2 Description of Dendrogram Structure Fields

Lets examine the contents of the pointer output from topologize. The original code represents the dendrogram in a slightly cumbersome way, but there are at least several programs which save you from having to understand too much of what's going on here:

```
IDL> help, *ptr, /struct
** Structure <192e208>, 21 tags, length=3610048, data length=3610041, refs=1:
 MERGER          DOUBLE    Array[49, 49]
 CLUSTER_LABEL   INT       Array[150801]
 CLUSTER_LABEL_H LONG      Array[98]
 CLUSTER_LABEL_RI
                 LONG      Array[60217]
 LEVELS          DOUBLE    Array[96]
 CLUSTERS        LONG      Array[2, 48]
 HEIGHT          DOUBLE    Array[97]
 KERNELS         LONG      Array[49]
 ORDER           LONG      Array[49]
 NEWMERGER       DOUBLE    Array[49, 49]
 X               LONG      Array[150801]
 Y               LONG      Array[150801]
 V               LONG      Array[150801]
 T               FLOAT     Array[150801]
 SZ              LONG      Array[5]
 CUBEINDEX       LONG      Array[150801]
 SZDATA          LONG      Array[5]
 ALL_NEIGHBORS   BYTE         0
 XLOCATION       DOUBLE    Array[97]
 NPIX            LONG      Array[49, 49]
 FAST            INT              1
```

Here's what some of these fields describe:

- kernels : These list the 1D indices (of the input image) of the location of 49 local maxima. Each of these kernels defines a leaf in the dendrogram
- clusters: Describes which two structures merge at each merger in the dendrogram. The number of mergers is always one less than the number of kernels. clusters[*, i] lists the ids of the two structures that merge to form structure i + n_kernel (the first n_kernel structures are the leaves.
- merger : The [i,j] entry lists the intensity below which kernel i and j are contained within a single contour
- x : A 1 dimensional list of x locations in the original data. (Not all pixels are included, if the input mask contains zeros. This is cumbersome, I know)
- y : Like x
- v : The velocity, if the input image was 3 dimensional
- t : The intensity at each (x,y,v) location

- cluster_label : the ID of the highest (most leafward) dendrogram structure that each (x,y,v) point belongs to.
- cubeindex : the 1d index of each (x,y,v) location in the original cube
- height: The height of each structure in the dendrogram, for plotting. The height of the leaves is the intensity of the local maximum. The height of all other structures is the intensity of the relevant contour merger
- xlocation : The xlocation of each structure, for plotting. At the moment, this carries no physical meaning.

CHAPTER

THREE

DENDROGRAM GENERATION WITH C++

If you can get it to work, I recommend generating dendrograms with C++ – the code runs much faster, works direcly with fits files, and is based on a simpler algorithm. This page will walk you through installing and running the program, and getting the output into IDL.

3.1 Installation

The code is available on GitHub here, and comes with a README and INSTALL file. You can either clone the git repository, our download the latest version from the download page.

You will need to install CCfits in order to run this program. If this is installed in a standard location, you should be fine with running:

```
./configure
make
make install
```

If this complains about not finding CCfits, you will need to supply extra `CPPFLAGS` and `LDFLAGS` to specify where to look for CCfits:

```
./configure CPPFLAGS="-I/path/to/CCfits/headers" LDFLAGS="-L/path/to/CCfits/libraries"
```

This will create an executable `dendro`, which is the main program.

3.2 Running

`dendro` runs from the command line as follows:

```
dendro [input_file] -o [output_file] -f [friends] -s [specfriends] -m [minpeak]
```

Note: `dendro` requires that the input fits file be in 32 bit float format.

The `friends`, `specfriends`, and `minpeak` values behve the same way as described in *Dendrogram Generation with IDL*. This produces a multi-extension fits file with the same basic information as the output from the IDL routine:

- Extension 0 is the original data.
- Extension 1 corresponds to the CLUSTER_LABEL tag of the IDL output structure.

- Extension 2 corresponds to the CLUSTERS tag.
- Extension 3 is the KERNELS tag

3.3 Reading into IDL

The routine `ptr = dendrcpp2idl(file_name)` will read a fits file created by the `dendro` C++ program and produce an IDL pointer compatible with the rest of the analysis routines.

Likewise, the dendroviz program accepts as input the name of one of these fits files:

```
dendroviz, 'fits_file_name'
```

3.4 Pruning

The C++ routine lacks parameters like `delta` and `npix`, which are used by IDL to 'prune' dendrograms of insignificant leaves. Instead, pruning is done in 2 passes:

```
[from command line]
dendro in_file -o out_file

idl
IDL> ptr = dendrocpp2idl(out_file)
IDL> to_prune = generate_prunelist(ptr, delta = 1.0, npix = 10, $
                                   minflux = 1.0, minpeak=5, $
                                   out_file = 'new_seeds.txt')
IDL> exit

dendro in_file -o pruned_out_file -k new_seeds.txt
```

The `-k` option in `dendro` specifies a file listing which pixels define the leaves of the dendrogram. The file is a 2-column list, giving "pixel_index, intensity". The intensity is a sanity check, to make sure the pixel locations are specified correctly – if they don't agree, the program exits in error.

`generate_prunelist` is an IDL routine which will generate the required prune file, based on the pruning parameters described in *Dendrogram Generation with IDL*

CHAPTER

FOUR

DENDROGRAM GENERATION WITH PYTHON

An experimental object-oriented pure-Python module (written by Thomas Robitaille) is available here. This code is still under development and should not be used in production mode at this stage. Bug reports and contributions are welcome via GitHub Issue tracking and Pull Requests.

CHAPTER

FIVE

RECIPES

5.1 Dendrogram Indexes

The dendrogram algorithm segments data into nested structures. Each structure is given a unique integer identifier. The following routines help to navigate these indices.

5.1.1 Leafward Mergers

The leafward mergers of structure `i` are all the nodes in the sub-tree rooted at `i`. Each leafward merger passes through structure `i` when traveling to the root. Alternatively, any node that is visited on a trip from `i` to a leaf is a leafward merger of `i`.

To find the IDs of the leafward mergers of structure 60 (Note that 60 is included in the output):

```
print, leafward_mergers(60, (*ptr).clusters)
          60          15          56          55          20          22          53          52
          27          30          50          49          26          29          33
```

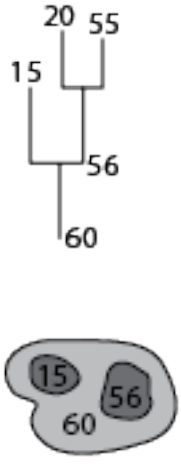
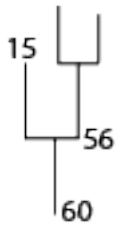


Figure 5.1: Partial dendrogram centered on structure 60, showing the first few leafward mergers.

To find the two immediate substructures that merge to form structure 60, use the `/parents` keyword:

```
IDL> print, leafward_mergers(60, (*ptr).clusters, /parents)
     56          15
```



5.1.2 Rootward Mergers

Rootward mergers are the opposite of leafward mergers. They represent superstructures, or trees that a given node is nested inside.

Rootward mergers of 60:

```
IDL> print, rootward_mergers(60, (*ptr).clusters)
     60          61
```

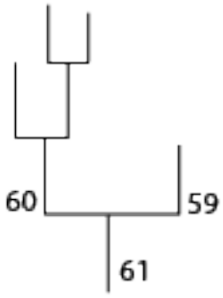
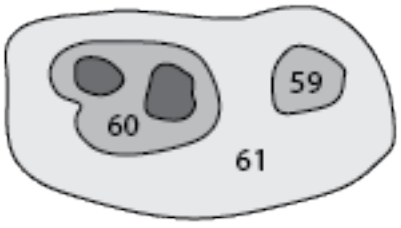


Figure 5.2: The rootward mergers of struture 60.



5.1.3 Siblings

The structure that structure 60 eventually merges with:

```
IDL> print, merger_partner(60, (*ptr).clusters)
      59
```

The structure that defines the merger of structure 59 and 60:

```
IDL> partner = merger_partner(60, (*ptr).clusters, merge = m)
IDL> print, m
      61
```

5.1.4 Mapping between indices and pixels

Use the substruct function to extract which pixels belong to each dendrogram structure.

Find the indices (in the flattened x/y/v/t arrays) of the pixels belonging to structure 60:

```
IDL> ind = substruct(60, ptr)
IDL> help, ind
<Expression>    LONG      = Array[3195]
IDL> intensities = (*ptr).t[ind]
IDL> xpos = (*ptr).x[ind]
IDL> ypos = (*ptr).y[ind]
```

Find only the pixels that belong to structure 60, but none of its substructures (i.e., a slice of the onion):

```
IDL> help, substruct(60, ptr, /single)
<Expression>    LONG      = Array[262]
```

Warning: In the IDL dendrogram implmentation, the x,y,v values **do not** always correspond to the pixel locations in the original data (they are occasionally offset by a few pixels). While this is not the case for C++-generated dendrograms, you shouldn't rely on using the x/y/v values to index into the original data. Instead, use `cube_indices = (*ptr).cubeindex[ind]`. This gives the (1D) indices into the original data

Here's an example of how **not** to index into the original cube (see warning above):

```
IDL> print, im[xpos[0:3], ypos[0:3], zpos[0:3]]
  7.48933      8.81364      10.1202      6.45180
IDL> print, intensities[0:3]
   6.66389      6.71393      6.73340      6.74873  ;- DOES NOT MATCH!
```

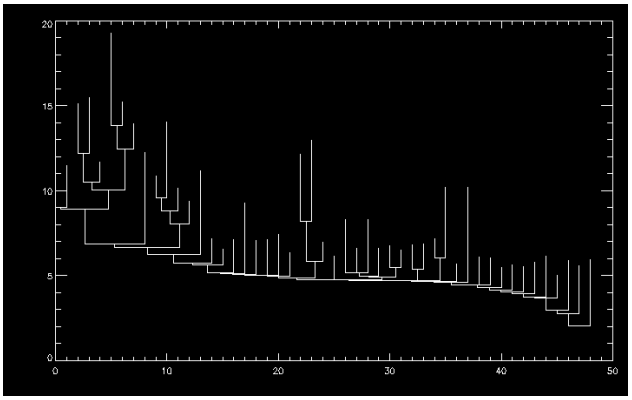
And the proper way:

```
IDL> ci = (*ptr).cubeindex[ind]
IDL> print im[ci[0:3]]
   6.66389      6.71393      6.73340      6.74873
IDL> print, intensities[0:3]
   6.66389      6.71393      6.73340      6.74873 ;- matches.
```

5.2 Plotting

Get a set of (x,y) points to plot a dendrogram:

```
IDL> root = dendro_root(ptr)  ;- grabs the ID of the dendrogram base
IDL> xy = dplot_xy(ptr, root) ;- draw from the root
IDL> plot, xy[0,*], xy[1,*]
```

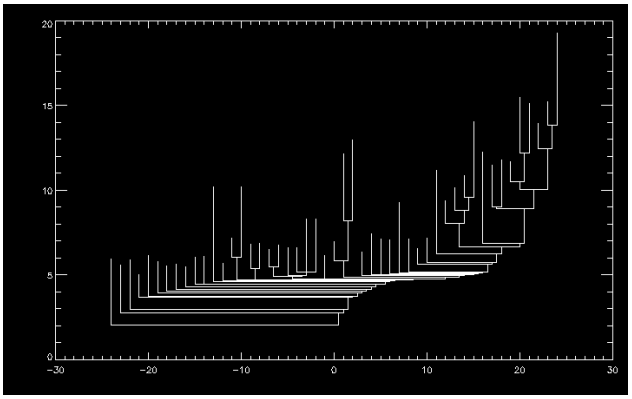


The height of each vertical line is the intensity value at which the two branches above it merge. The height of each leaf is the peak intensity of that leaf. The x ordering is arbitrary (see next section).

5.2.1 Re-ordering

You can re-sort the dendrogram using `dendro_sort`. For example, to sort the dendrogram structure such that the left subtree of any node has an integrated intensity less than the right subtree:

```
IDL> dendro_sort, ptr, /inten
IDL> xy = dplot_xy(ptr, root)
IDL> plot, xy[0,*], xy[1,*]
```



`dendro_sort` also supports sorting based on the height of the tallest leaf in each subtree (i.e. the peak intensity of each substructure) via the `/height` option.

User-defined ordering

`dendro_sort` accepts a keyword, `key`, to support user-defined orderings. This should be an array, whose size is the number of structures in the dendrogram. The dendrogram will be sorted such that the left subtree of any node will have an associated key value less than the right subtree. For example, to re-implement the functionality of the `/height` keyword:

```
function height_key(ptr)
    nst = n_elements((*ptr).height)
    result = fltarr(nst)
    for i = 0, nst - 1 do begin
       result[i] = max( (*ptr).t[substruct(i, ptr)], /nan)
    return, result
```

```
IDL> keys = height_key(ptr)
IDL> dendro_sort, ptr, key = keys
```

5.3 Analysis

The low-level routines described in the *Dendrogram Indexes* section can be used to extract and measure properties of each substructure.

The dendro_catalog function measures some common information:

```
IDL> catalog = dendro_catalog(ptr)
IDL> help, catalog, /struct
** Structure <2dc78d8>, 16 tags, length=64, data length=64, refs=1:
   X               FLOAT           109.922
   Y               FLOAT           7.59273
   V               FLOAT           0.00000
   SIG_MAJ         FLOAT           8.83641
   SIG_MIN         FLOAT           2.96764
   SIG_V           FLOAT           0.00000
   SIG_R           FLOAT           5.12087
   AREA_MASK       FLOAT           265.000
   PERIMETER_MASK  FLOAT           93.8909
   FLUX            FLOAT           639.349
   PEAK_INTEN      FLOAT           5.92300
   VOL             FLOAT           265.000
   VIRIAL          FLOAT           0.00000
   SHOULDER_HEIGHT FLOAT           3.88697
   VOL_LEFT        FLOAT               NaN
   VOL_RIGHT       FLOAT               NaN
```

The function creates an array of structures, one for each dendrogram structure. The fields of this structure are:

- `x`: Intensity-weighted mean x position
- `y`: Intensity-weighted mean y position
- `v`: Intensity-weighted mean v position
- `sig_maj`: Semimajor axis of an ellipsoid approximation to the structure's projection in the x/y plane
- `sig_min`: Semiminor axis of ellispe described above
- `sig_v`: RMS dispersion in velocity
- `sig_r`: sqrt(sig_maj * sig_min)
- `area_mask`: Number of pixels of the structure's 2D projection
- `perimeter_mask`: Perimeter of this projection
- `flux`: Sum of intensities for all pixels inside the structure
- `peak_inten`: Maximum intensity inside the structure
- `vol`: Number of pixels inside the structure
- `virial`: Virial parameter (see note below)
- `shoulder_height`: Height of the vertical line connecting this structure to its merger in a dendrogram plot
- `vol_left`: Volume of left subtree

- `vol_right`: Volume of right subtree

Tip: dendro_catalog works in pixel units by default. The optional `len_scale` defines the linear scale of each pixel. If present, it will convert `sig_maj`, `sig_min`, and `sig_r` into physical units. `vel_scale` gives the velocity width of each channel, and will convert `sig_v` into physical units. `flux2mass` is a multiplicative factor to convert summed intensity into mass. It is used by `virial` (see next tip)

Tip: The virial parameter is defined as `5 eta R v^2 / (G M)`, where `eta = 1.91` (see Rosolowsky et al. 2008). M is calculated by multiplying the `flux` field by the optional `flux2mass` keyword, and `G` is given in CGS units. Thus, to get sensible units for this field, use `len_scale` to convert from pixels to cm, `vel_scale` to convert from velocity pixels to cm/s, and `flux2mass` to convert to g.

This routine depends on functions in the Beaumont IDL library.

CHAPTER

SIX

HOW TO VISUALIZE DENDROGRAMS

6.1 DendroStar (Java Applet)

DendroStar is a demo that shows the connection between dendrograms and the images they describe. You can visit it here. However, it cannot be applied to other datasets.

6.2 DendroViz (IDL library)

DendroViz is an IDL package designed to visualize dendrograms interactively. Documentation is forthcoming, but for now see these links:

- Introduction Video
- Walkthrough
- Download code
- *genindex*
- *search*