Transient-Finding Pipeline

Joshua Burt

June 27, 2014

1 Data Unpacking

The data unpacking class *tfdata* prepares the time-frequency data for analysis. A single second of data is prepared at a time and stored as a two-dimensional array of shape $N \times M$, where N is the number of frequency channels and M is the number of time samples.

If the source of the data is a .fits file, the header/data unit (HDU) is opened using the package *PyFITS*. The user specifies the zero-based index of the second of data to process. If the source is a simulated data set, it is opened using the *NumPy* package and the entire file is processed at once; for simulated data, the user must additionally supply the observing frequencies associated with each frequency channel, as well as the time resolution, since this data cannot be found in an HDU like in the case of a .fits file.

For both real and simulated data, the user specifies the desired smoothings, separated into constituent time-bin samples and frequency samples, as well as the number of channels to chop off each end of the bandpass. The smoothing is done before any further processing (including edge chopping), and the numbers of smoothing samples are stored as attributes of the *tfdata* instance. The specified number of channels are then excised from the data array.

The root-mean-square (RMS) and the mean of the raw, full-resolution time-frequency data, **after** smoothing and bandpass-chopping have been applied, are also stored as class attributes. They are computed iteratively, by removing high-signal outliers, re-computing the RMS and mean using the remaining samples, and specifying a tolerance for convergence. When the difference between consecutive iterations is within the tolerance, the RMS and mean are reported. This prevents strong or weak signals from biasing effects.

Once the data have been unpacked and prepared, the *tfdata* instance is passed to the friends-of-friends algorithm.

2 Friends-of-friends

The friends-of-friends algorithm is used to identify significant signal-to-noise (S/N) regions within the frequency-time plane. The function, named *fof* in its implementation, takes two required arguments (in addition to the *tfdata* instance), and up to three optional keyword arguments.

2.1 Function arguments

The two required arguments are signal thresholds – the single pixel signal threshold and the combined blob S/N threshold. The thresholds are the number of units of the RMS, in excess of the mean, which the pixel/blob intensity value must exceed. The single pixel signal threshold is a low threshold ($\sim 0.25 - 1.5\sigma$), intended to return all pixels potentially indicating the presence of an astrophysical signal, in addition to many unavoidable spurious noise pixels. The pixels returned by the single pixel threshold are then grouped into contiguous blobs in the frequency-time plane. Contiguous in this sense means that all pixels in a given blob share at least one neighbor in the 2D array of pixels that was also above the first threshold. These contiguous 'blobs', as they will henceforth be referred to, are isolated islands of pixels that have exceeded the first threshold and that are all interconnected in the frequency-time plane. Blobs have their combined S/N ratio computed in the following way:

$$I_N = \frac{1}{N}\sum_{k=1}^{N} I_k, \tag{1}$$

$$\sigma_N = \frac{\sigma_1}{\sqrt{N}}, \tag{2}$$

$$s \equiv \frac{I_N}{\sigma_N} = \frac{\frac{1}{N}\sum_{k=1}^{N} I_k}{\frac{\sigma_1}{\sqrt{N}}} = \frac{\sum_{k=1}^{N} I_k}{\sigma_1 \times \sqrt{N}}, \tag{3}$$

where I_k is the intensity of the k-th pixel in the blob, σ_1 is the single-pixel RMS described in section 1, and s is the reported quantity associated with the S/N ratio for the blob. Blobs are only kept if their combined S/N ratio, s, exceeds the second user-provided threshold. The second threshold is necessarily much higher than the first, in practice as high as $\sim 50 - 200\sigma$; looking at equation (3), we see that if the mean of the constituent pixel intensities within a blob is constant, then s will scale as $\sqrt{N}$.

2.2 Function keyword arguments

There are three additional input parameters the user may specify, but which he or she is not required to. These parameters revert to default values of zero if they are not specified. The first two arguments are allowed pixel gaps, one for each dimension. Recall from the previous section that blobs are defined as **contiguous**, isolated islands of pixels whose individual pixels all exceed the first threshold and whose

combined, accumulated S/N exceeds the second threshold. This user may relax the definition of contiguous by allowing a given number of non-threshold-exceeding pixels to lie between adjacent threshold-exceeding pixels, while still considering the threshold-exceeding pixels to be contiguous. For example, if two nearby pixels exceeding the first threshold were separated by one intervening frequency channel and two intervening time bins, and the user had specified pixel gaps in the time and frequency dimensions of one and two, respectively, then these pixels would be considered members of the same blob. The motive behind allowing gaps is to help mitigate the effects of noise, as some pixels will necessarily by random chance lie below the threshold even in regions where a true astrophysical signal is strong. The actual implementation of this procedure involves the dilation and erosion operations, which are described in a later section. The way it works is described below, although it may be necessary to read the later section on mathematical morphology to fully grasp this procedure:

1. I generate a mask of all pixels exceeding the first threshold, so the 2-D frequency-time plane is mapped to a binary array of ones and zeros, with ones representing pixels which satisfied the first threshold criterion.

2. I construct a structuring element whose shape is determined by the particular values of allowed pixel gaps.

3. I use this structuring element to dilate the binary image, introducing 'fake' ones within the binary mask which now bridge any gaps equal to or smaller than the allowed gaps.

4. I use the *SciPy.ndimage* package (for multi-dimensional image processing) to identify and label truly contiguous islands within this dilated binary mask array. This package returns a mask, identical to the dilated mask that was passed to it, with the first contiguous blob's constituent pixels assigned the value one, the second contiguous blob's constituent pixels assigned the value two, and so on.

5. Finally I apply the original, undilated pixel mask to this labelled blob mask to remove all the 'fake' pixels that were introduced during the dilation operation to conjoin blobs within the allowed pixel gaps.

The final keyword argument is a minimum cutoff threshold for the number of constituent pixels comprising a blob. For example, if the cutoff is set to 10, then no blobs will be returned with fewer than 10 members. The motive for this function parameter was to help omit small blobs arising from noise fluctuations, although its effect is very similar to that of dialing up the second combined S/N threshold.

Once all blobs have been identified, individual binary masks for each blob, representing their 'spatial' location in the two dimensional frequency-time plane, are passed to a blob generator class where they are

characterized.

3 Blob Attributes

After threshold-satisfying regions are identified by the friends-of-friends algorithm, binary masks specifying their locations in the frequency-time plane are passed to a blob generator class, where the blob objects are instantiated and their attributes populated. The specific attributes computed for each blob are described in the following sections.

3.1 S/N Ratio

See section 2.1.

3.2 Pixel Count

Trivially, the number of constituent pixels within the blob.

3.3 Frequency channel and time bin indexes

While the binary mask is useful as a 'spatial' representation of the blob in the two-dimensional plane, the actual indexes of pixels within the array are necessary for most of the computations in the blob characterization. Hence, I store two arrays of the frequency channel and time bin indexes associated with each pixel in the blob, where the i-th element of each array provides the frequency channel and time bin index of the i-th pixel in the blob. They are essentially the outputs of *numpy.where*.

3.4 Frequency and Time Extent

The indexes of the highest and lowest frequency channels (e.g. rows), and the earliest and latest time bins (e.g. columns), in which the blob appears. These values are stored in two tuples.

3.5 Centroid

The centroid is defined by computing the first moment in both the time and frequency dimensions, using the indexes of the channels and time bins in which constituent blob pixels appear. That is,

$$f_c = \frac{1}{N} \sum_{i=1}^{N} f_i, \tag{4}$$

$$t_c = \frac{1}{N} \sum_{i=1}^{N} t_i, \tag{5}$$

where (f_c,t_c) defines the centroid and f_i and t_i are the frequency channels and time bins for the i-th pixel, respectively. This value pair is stored in a tuple.

3.6 Slope

The slope for each blob is computed using orthogonal distance regression. A first-order polynomial, of the form $f = a + b * t$, where f is the frequency channel and t is the time bin, is found such that the sum of the orthogonal distance residuals is minimized. To treat both frequency and time on equal grounds, I scale the time axis by a factor $c = \frac{N_\nu}{N_t}$, where N_ν is the number of frequency channels and N_t the number of time bins. At the end of the procedure, I scale the b parameter (the slope) by the same factor to appropriately reflect the raw, unscaled data.

Because the scipy.odr package requires an initial estimate of the parameters a and b, a horizontal line ($b = 0$) through the centroid of the blob (where $f_c = \frac{1}{N} \sum_{i=1}^{N} f_i$) is used as the starting point for the minimization procedure. The scipy.odr package iteratively adjusts these parameters, testing for convergence upon a minimum, for a given number of iterations (I believe 50 is the default). If the algorithm converged, I use whatever parameters a and b were returned.

If the algorithm did not converge, and the iteration limit was reached, I invert the axes and try again, e.g., I attempt to find the parameters describing a first-order polynomial of the form $t = a' + b' * f$. Again, I use an initial estimate of a horizontal line, this time passing through $t = \frac{c}{N} \sum_{i=1}^{N} t_i$, where c is the scale factor. After returning the parameters a and b, I invert them in the following way:

$$a = -\frac{a'}{b'}, \tag{6}$$

$$b = \frac{1}{b'}, \tag{7}$$

upon which b is scaled by the scale factor c, just like in the non-inverted case, to reflect the raw, unscaled data.

Following this procedure, the quantities b and a are stored in a tuple. I also include two additional relevant attributes: a boolean flag for the convergence of the scipy.odr computation, and the variance in the residuals (each stored as separate, additional attributes).

3.7 Curvature

The curvature is calculated nearly identically to the slope. A second-order polynomial is used, of the form $f = a + b * t^2$. For the initial estimate of the parameters a and b, I use those returned by the slope computation. I don't invert the axes if the convergence fails. The b parameter is scaled by two factors of c, reflecting the fact that this curve is quadratic in t. Other than these differences, the procedure is the same. I include the two attributes for convergence and residual variance in this case as well.

3.8 Roundness

The roundness is a measure of how disk-like a blob is. Each blob has some orientation, e.g. the angle made between the horizontal (the time axis) and the line minimizing the second moment. I find the orientation angle θ minimizing the quantity

$$E = \sum_{i=1}^{N} r_i^2 = a \sin^2\theta - b \sin\theta \cos\theta + c \cos^2\theta, \tag{8}$$

where r_i is the distance from the orientation axis to the i-th pixel, and a, b, and c are the three second moments,

$$a = \sum_{i=1}^{N} \overline{t_i}^2 \tag{9}$$

$$b = \sum_{i=1}^{N} \overline{t_i} \times \overline{f_i} \tag{10}$$

$$c = \sum_{i=1}^{N} \overline{f_i}^2, \tag{11}$$

where $\overline{t_i}$ and $\overline{f_i}$ are the deviations of the i-th pixel's time bin and frequency channel from their respective means. Using some trigonometric identities to get

$$E = \frac{1}{2}(a + c) - \frac{1}{2}(a - c)\cos 2\theta - \frac{1}{2} b \sin 2\theta \tag{12}$$

and setting $\frac{dE}{d\theta} = 0$, we can obtain

$$\sin 2\theta = \pm \frac{b}{\sqrt{b^2 + (a - c)^2}}, \tag{13}$$

where equation (12) tells us the positive solution minimizes E and the negative solution maximizes E. Then we can define the ratio

$$e = \frac{E_{min}}{E_{max}} \tag{14}$$

to be a measure of roundness, lying somewhere between zero (a perfect line) and one (a perfect disk). In the case of a perfect line, the second moment about the orientation axis is zero, so the roundness is zero. In the case of a perfect disk, the orientation axis would be undefined, and any set of orthogonal axes would yield the same second moment, yielding a roundness of one.

3.9 RFI Flag

To identify blobs likely to be broadband RFI, I flag blobs whose slopes are particularly vertical. The user specifies the minimum theoretical dispersion measure allowed, which by default is 5 pc cm^{-3}. This dispersion measure is used in conjunction with the channel frequencies and the time resolution (determined during the data unpacking stage) to derive an expected time bin delay across the entire bandwidth. The number of channels is divided by the expected time bin delay, effectively 'linearizing' the ν^{-2} dispersion relation, to get a threshold in units of channels per time bin, which are the same units used for the slope in section 3.6. Mathematically, the equations are

$$\Delta t = 4.15ms \times \mathrm{DM} \times \left[\left(\frac{\nu_1}{\mathrm{GHz}} \right)^{-2} - \left(\frac{\nu_2}{\mathrm{GHz}} \right)^{-2} \right], \tag{15}$$

$$N_\mathrm{t} = \Delta t \times \left(\frac{N_\mathrm{t}}{s} \right), \tag{16}$$

$$\left(\frac{dN_\nu}{dN_t} \right)_\mathrm{max} = \frac{N_\nu}{N_t}, \tag{17}$$

where ν_1 and ν_2 are the observing frequencies at the edges of the bandpass, (N_t/s) is the time resolution, and N_ν is the number of frequency channels. To flag narrowband RFI, I use a minimum slope threshold

$$\left(\frac{dN_\nu}{dN_t} \right)_\mathrm{min} = \left(\frac{dN_\nu}{dN_t} \right)^{-1}_\mathrm{max} \tag{18}$$

3.10 Widths

To get a measure of the temporal and spectral widths of each blob in units of indexes, I provide the user with a few different options. The time bin and frequency channel deviations for all pixels in a blob can be found relative to either the blob centroid or the blob's best fit line (defined by the slope and y-intercept computed in section 3.6). The user can then either use the maximum deviation along each dimension, or

the rms deviation, to define the blob's widths. To clarify, relative to the blob centroid, the deviations are

$$df_i = f_i - f_c \tag{19}$$

$$dt_i = t_i - t_c \tag{20}$$

where f_c and t_c are the frequency channel and time bin of the blob centroid, and f_i and t_i are the frequency channel and time bin of the i-th pixel in the blob, respectively. Relative to the best-fit line, the deviations are

$$df_i = f_i - (a + b * t_i) \tag{21}$$

$$dt_i = t_i - \frac{f_i - a}{b} \tag{22}$$

where parameters a and b are defined in section 3.6. If the user chooses to use the maximum deviation as the width, then the widths are trivially $max(df_i)$ and $max(dt_i)$. If the user chooses the rms deviations as the widths, then they are computed as

$$w_\nu = \sigma_\nu \equiv \sqrt{\frac{1}{N}\sum_{i=1}^{N}(df_i)^2} \tag{23}$$

$$w_t = \sigma_t \equiv \sqrt{\frac{1}{N}\sum_{i=1}^{N}(dt_i)^2}, \tag{24}$$

where w_ν and w_t are the widths in each dimension.

3.11 Linear Extrapolation Mask

The motive behind the linear extrapolation mask is to identify other blobs lying within some distance of the extrapolation of a given blob's slope. I construct a mask of all pixels in the frequency-time plane lying within n_ν intervals of the spectral width w_ν and n_t intervals of the temporal width w_t from the extrapolated best-fit linear line. This involves constructing two meshgrids, or coordinate matrices, for both frequency channels and time bins, and computing df_i and dt_i, as defined in section 3.10, for all pixels in the array. I then construct masks for points satisfying $df_i < n_\nu \times w_\nu$ and $dt_i < n_t \times w_t$. Basically, I'm just computing a time bin and frequency channel separation relative to the best-fit line for each pixel in the frequency-time plane, and using those pixels within the specified thresholds to construct a binary mask. In section 4, I look for overlap between the binary extrapolation mask of blob i and the binary mask representing the location of blob j, for all i and j.

3.12 Quadratic Extrapolation Mask

In a similar vein, I wanted to identify other blobs lying within some distance of the extrapolation of a given blob's second-order best-fit line. The process is nearly identical to that of section 3.11, using the quadratic best-fit line as opposed to the linear one, but with one measure of added complexity; because of the quadratic time-dependence, the computation for dt_i is

$$dt_i = t_i - \sqrt{\frac{f_i - a}{b}}, \tag{25}$$

where a and b are defined in section 3.7. Because this equation will return imaginary values if the argument of the square root operation is negative, I only consider frequency channels in which the quantity $(f_i - a)/b$ is positive.

When all attributes have been computed for all blobs identified by the friends-of-friends algorithm, the list of *blob* class instances is passed to the *best_friends* class.

4 Best friends

After *blob* objects have been initialized, I set out to determine whether any subsets of blobs exhibit a sufficient degree of similarity to justify consolidating them into a single, larger blob. Analogous to allowing pixel gaps in response to fluctuations on the individual sample scale, this procedure helps mitigate the impact of larger modulational effects, such as scintillation, which may dissociate sub-regions of a larger underlying extended signal. To carry out this analysis, I compute pairwise relational quantities between blobs, and assign each pair of blobs a similarity score. If the score exceeds a user-defined threshold, the blobs are considered part of the same underlying feature; when many such pairwise associations occur, constituent blobs are transitively linked. That is, if blob i is similar to blob j, and blob j is similar to blob k, then blobs i, j, and k together constitute a single larger blob – they're best friends!

4.1 Scoring

The scoring system currently utilizes only the blobs' centroid, slope, curvature, roundness, and extrapolation masks. Pairwise distances are computed using the centroids; percent differences are computed for each of the slope, curvature, and roundness; and overlaps are determined between one blob's identification mask and the extrapolation mask of another. The possible scores, as well as their thresholds, are discussed in the sections that follow.

4.1.1 Centroid distance

The centroid distance is trivially determined by the frequency channel separation, df, and the time bin separation, dt, of the centroids of two disparate blobs.

The scores are determined in the following way[1]:

$$S_{df} = \begin{cases} 1, & \text{if } df < 100. \\ 2, & \text{if } df < 50. \end{cases}$$

$$S_{dt} = \begin{cases} 1, & \text{if } dt < 1000. \\ 2, & \text{if } dt < 800. \\ 3, & \text{if } dt < 600. \\ 4, & \text{if } dt < 400\ . \\ 5, & \text{if } dt < 200. \end{cases}$$

In this implementation, blob pairs can be awarded up to seven points for their centroids' proximity. All scores other than the dt score have a maximum value of two; I intentionally allocate more possible points to dt because that seems to be the strongest indicator of blob similarity, at least for the fast transients I am seeking.

4.1.2 Percent differences

Percent differences are computed between pairs of blobs for their respective slopes, curvatures, and roundnesses. While I call it a 'percent' difference, I express the computations and thresholds as fractions. Each of the three is scored identically, in the following manner:

$$S_{frac} = \begin{cases} 1, & \text{if } frac < 1. \\ 2, & \text{if } frac < 0.5. \end{cases}$$

In this implementation, blob pairs can be awarded up to six points for exhibiting similar values for their slope, curvature, and roundness.

4.1.3 Mask extrapolation overlap

Members of the set of blob mask pixels for blob 1 are checked for inclusion in the set of extrapolation mask pixels for blob 2, and vice-versa ($1 \leftrightarrow 2$):

$$S_{\alpha\beta} = \begin{cases} 1, & \text{if } B_i^\alpha \in E_j^\beta; \text{ for } \alpha, \beta = 1, 2,\ \alpha \neq \beta \end{cases}$$

[1] I should probably convert the dt cutoffs to units of time, and perhaps units of frequency for df as well.

Thus, for one pair of blobs, up to two points may be awarded per extrapolation mask: one point if the mask of blob 1 shares pixels with the extrapolation mask of blob 2, and another point if the mask of blob 2 shares pixels with the extrapolation mask of blob 1. The combined maximum possible awarded score then, for two extrapolation masks of linear and quadratic origin, is four points. This implementation does not consider the actual number or percentage of pixels in a blob mask that overlaps with an extrapolation mask – one pixel is just as good as all of them.

4.2 Blob concatenation

Blobs are concatenated (consolidated) in the following way:

1. The scores for all possible blob pairs are computed.
2. The scores are compared against a user-provided threshold score, above which blobs are deemed significantly similar.
3. Blobs are transitively grouped; that is, if the score for blob pair (1,2) exceeded the threshold, and the score for blob pair (1,5) exceeded the threshold, then the set of blobs $\{1, 2, 5\}$ is now considered a single blob.
4. New blobs are characterized using the procedure in section 3. The original constituent blobs are discarded, and the new blob object is added to the current list of blobs.
5. Steps 1 through 4 are repeated until no pairs exceed the threshold. On each subsequent iteration, scores are only computed between blobs who haven't already had their score computed, e.g. between one newly generated blob and one old blob surviving the previous iteration.

5 Additional tools

I've written some additional functions that haven't made it into the pipeline, primarily because of their computational costs and relative unnecessity. Nonetheless, they may prove to be useful in later applications. The functions described below allow the user to 'fill in' the interior regions of blobs. These functions are motivated by the assumption that many of the null pixels in a blob's sparse interior regions are simply due to noise fluctuations veiling whatever true signal is present (assuming one is). Using the entire region where a blob occurs in the frequency-time plane may provide a more realistic estimate of the true significance of the signal in the midst of a sea of noise.

5.1 OpenCV filling function

OpenCV is an **Open** source **C**omputer **V**ision package. The specific task for which I employ OpenCV is that of identifying the two-dimensional contour enclosing the blob in the frequency-time plane. This is enabled by my use of binary masks to represent the locations of pixels within blobs, which I can cast to 8-bit integers to generate an equivalent 2-D binary image. When processed by OpenCV, all contours necessary to reconstruct the blob in its entirety are located, which includes the contours of all void regions interior to the blob's outer boundaries. Since each contour has a length associated with it (e.g. the number of nodes), I simply select the longest contour under the assumption that it is the true outer contour of the blob. I have not done any robust testing to validate this assumption. Once the contour has been identified, OpenCV can easily provide an n-pixel wide reconstruction of the contour, as well as a filled-in contour; these functionalities are used to provide the user with either the single-pixel-wide contour or the filled-in version of the contour, according to user-defined inputs.

5.2 Mathematical morphology filling function

This method of contour- finding and filling is a bit more contrived. I'll first provide some background on mathematical morphological operations. Two such operations are borrowed from mathematics for my purposes: binary dilation and binary erosion.

According to the *SciPy* documentation for its pythonic implementation, "the binary dilation of an image by a structuring element is the locus of the points covered by the structuring element, when its center lies within the non-zero points of the image." Mathematically, if A is a binary image located on an integer grid, and B is a structuring element, then the dilation of A by B is given by

$$A \oplus B = \bigcup_{b \in B} A_b. \tag{26}$$

For both dilation and erosion in two dimensions, I use a fully connected 3×3 matrix of ones as my structuring element. In words, upon every binary dilation iteration, the blob is enlarged or 'dilated' by also including the set of pixels adjacent to all current blob pixels.

Again presenting the *SciPy* definition, "the binary erosion of an image by a structuring element is the locus of the points where a superimposition of the structuring element centered on the point is entirely contained in the set of non-zero elements of the image." Mathematically, if A is a binary image located on an integer grid E, and B is a structuring element, then the erosion of A by B is given by

$$A \ominus B = \{z \in E \mid B_z \subseteq A\} = \bigcap_{b \in B} A_{-b}, \text{ where } B_z = \{b + z \mid b \in B\}, \ \forall z \in E. \tag{27}$$

In words, it is basically the inverse of binary dilation; using a fully connected structuring element, it removes pixels from the blob which are not bordered on all side by other blob pixels.

The way these two operations can be used in conjunction is by dilating a blob mask repeatedly, until the interior holes have all been filled, and then eroding as many times as it was dilated. During the dilations, the blob interior becomes solid and filled as null pixels adjacent to blob pixels are admitted blob membership; during the subsequent erosions, the interior is consequently unaffected, as all interior points are now bordered on all sides by other blob pixels.

In its implementation, I require the user to specify two input parameters: the user must first specify the maximum number of dilation iterations, after which the procedure terminates; and the user must specify the iteration increment, e.g. how many dilation iterations are performed at a time. The function basically executes a *while* loop, applying the specified number of additional dilations during each pass, and terminating only when the entire interior is filled or the maximum number of iterations has been reached. To check whether the interior has been filled, I actually reference the OpenCV function described in the previous section, and determine whether or not it returns a single contour (indicating the presence of a single, filled object).

At this point you may be wondering, why go through all this trouble when OpenCV does the exact same thing more simply? My original motivation for creating this function was to vastly reduce the number of pixels associated with a blob without losing information about its 'spatial' extent. At the time I was computing the closest approach of each pair of blobs in the frequency-time plane, and since this can be done using the boundary alone, and I couldn't find a better way to do it than a pairwise pixel distance minimization, I needed to reduce the number of operations to make it computationally feasible (since this is an $\mathcal{O}(n^2)$ algorithm). Because using OpenCV provides you with the *exact* blob contour, it actually contains many more pixels than the contour derived using these math morphology operations (as a quick test will confirm).

6 Sample Output

To better illustrate some of the features I've discussed, I'll present an example obtained using the B0523+11 data set. For the 1st second of data in the data set, I smoothed by 2 frequency channels and 16 time bins. I then applied a first-pass threshold of 0.25σ, and a combined S/N threshold of 175σ. 15 blobs were identified, and I'll use one of them as my prototype. The raw blob, in the frequency-time plane, is shown in figure 1.

Various scalar and boolean attributes of the blob are shown below:

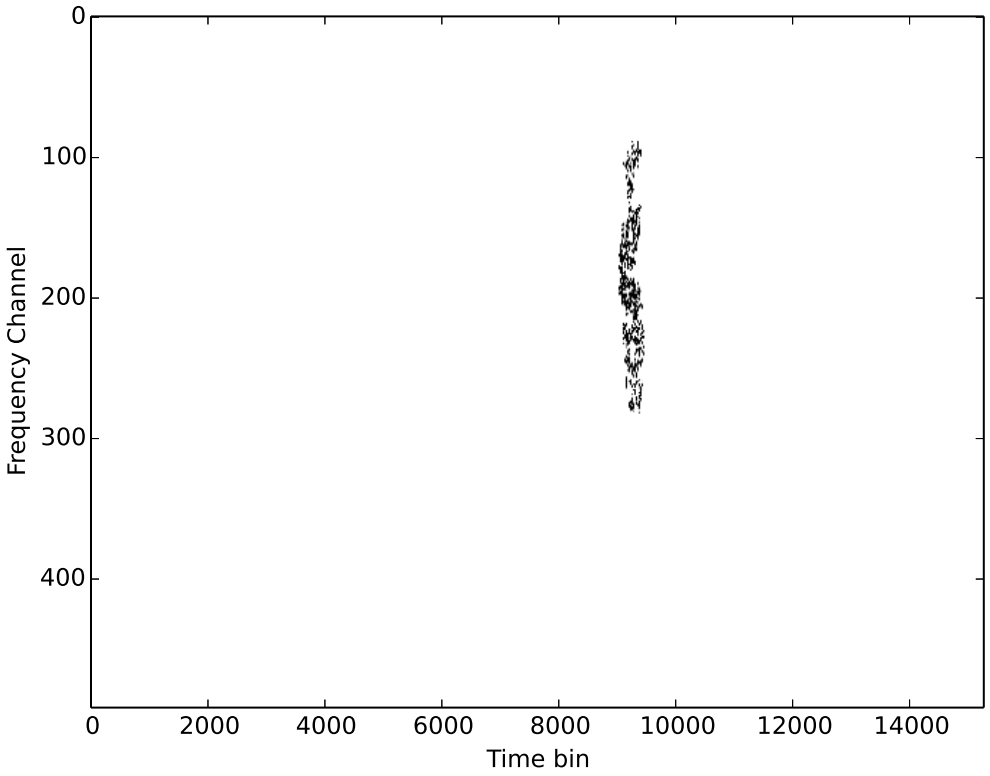

Figure 1: The raw blob mask in the frequency-time plane.

```
In [51]: blobs[6].pprint()
```

```
 S/N Ratio:             345.33
 Frequency Channels:    [85,282]
 Time Bins:             [8995,9441]
 Centroid:              [192.03,9221.70]
 Slope, intercept:      2.86, -26222.74
 Slope converged:       True
 Slope res_var:         9.84
 Curvature, intercept:  1.56e-04, -13063.85
 Curvature converged:   True
 Curvature res_var:     9.84
 Roundness:             0.86
 Filledness:            0.81
 RFI Slope Flag:        False
 Widths:                (278.85,97.35)
```

```
Fill Type:              N/A
```

Additionally, the linear and quadratic extrapolation masks are illustrated in figures 2 and 3 (using $n = 3$). I know they look indiscernible, but they are distinct.

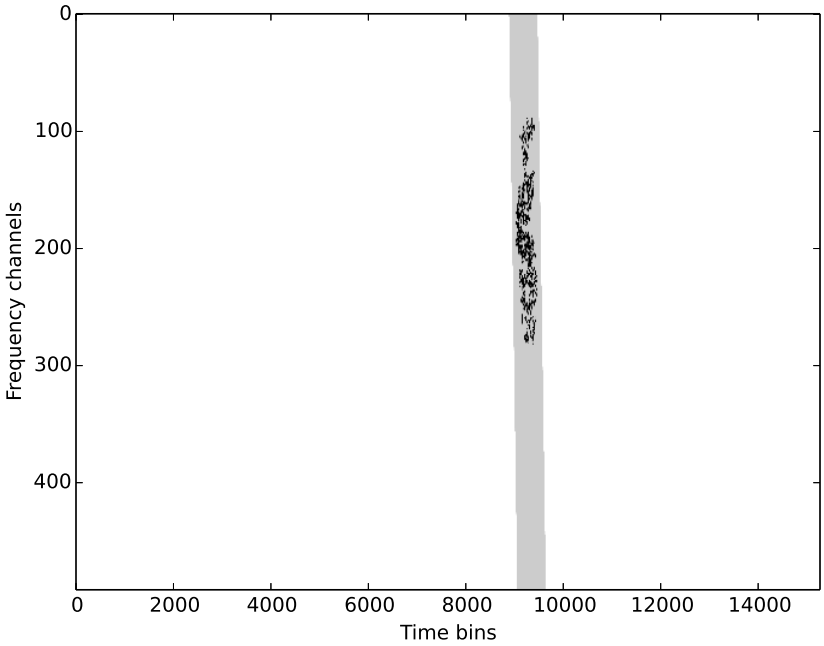

Figure 2: The blob plotted over its linear extrapolation window.

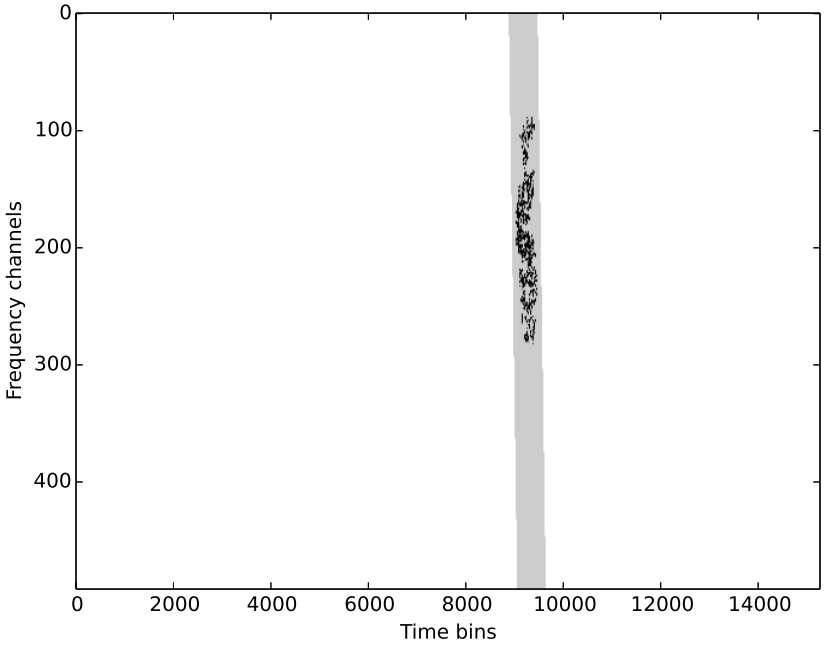

Figure 3: The blob plotted over its quadratic extrapolation window.