Notes on tree dedispersion (very incomplete)

Kendrick Smith

July 4, 2026

Contents

1 Tree gridding kernel

Our starting point for tree dedispersion is a shape $(N_{\rm freq}, N_{\rm time})$ array representing channelized intensity time series. We do not assume that frequency channels are evenly spaced (in CHORD we use unequal spacing). We want to search this array for FRBs with delay $\propto \nu^{-2}$.

The first step is a "tree gridding" kernel which changes variables so that the delay is linear (i.e. the FRB dispersion sweep looks like a straight line). The input of the tree gridding kernel is a shape $(N_{\rm freq}, N_{\rm time})$ array, and the output is a shape $(N_{\rm tree}, N_{\rm time})$ array, where $N_{\rm tree} = 2^r$. The parameter r is the tree dedispersion "rank" and is specified in a config file.

The leading axis of the input array represents frequency channels, and will be indexed by $0 \le F < N_{\rm freq}$. The leading axis of the output array represents a reparameterized frequency channel ("tree-freq" channel for short), and will be indexed by $0 \le f < 2^r$. Conceptually, the tree gridding kernel just reparameterizes the frequency axis (i.e. the time axis is a spectator.)

The tree gridding kernel also takes an auxiliary array `channel_map`, which describes the mapping between frequency channels $0 \le F < N_{\rm freq}$ and tree-freq channels $0 \le f < N_{\rm tree}$. The array `channel_map` has length $N_{\rm tree}+1$, and its entries are (in general non-integer) frequency-channel indices in the range $[0, N_{\rm freq}]$. Writing c_f for the f-th entry, tree-freq channel f is defined to occupy the half-open interval of frequency channels

$$I_f = \big[\, c_{f+1},\ c_f \,\big), \qquad 0 \le f < N_{\rm tree}, \tag{1}$$

i.e. consecutive entries give the lower and upper edges of each tree-freq channel. The entries are fixed by the dispersion geometry: the tree-freq channels are equally spaced in the dispersion variable $x \equiv \nu^{-2}$ (this is the change of variables that linearizes the delay), so the f-th edge sits at

$$x_f = \nu_{\rm hi}^{-2} + \frac{f}{N_{\rm tree}} \left(\nu_{\rm lo}^{-2} - \nu_{\rm hi}^{-2}\right), \tag{2}$$

and c_f is the frequency-channel index of the frequency $\nu_f = x_f^{-1/2}$, where the index increases with frequency, running from 0 at the band bottom $\nu_{\rm lo}$ to $N_{\rm freq}$ at the band top $\nu_{\rm hi}$. In particular $c_0 = N_{\rm freq}$ and $c_{N_{\rm tree}} = 0$, so `channel_map` is monotonically *decreasing* ($c_{f+1} < c_f$). We store it in double precision, since the gridding weights below are obtained by differencing adjacent entries, which loses relative precision.

The details of the tree gridding kernel are as follows. Each output tree-freq channel f is a weighted sum of the input frequency channels whose bins overlap the interval $I_f = [c_{f+1}, c_f)$:

$$\mathrm{out}[f] = \sum_{F=0}^{N_{\rm freq}-1} w_{fF}\, \mathrm{in}[F], \tag{3}$$

where $\mathrm{in}[F]$ and $\mathrm{out}[f]$ are length-$N_{\rm time}$ time series, and the same weights are applied at every time sample (the time axis is untouched). The weight w_{fF} is the length of the overlap between the tree-freq channel interval I_f and the unit-width frequency bin $[F, F+1)$:

$$w_{fF} = \max\big(0,\ \min(c_f,\, F+1) - \max(c_{f+1},\, F)\big). \tag{4}$$

A frequency channel lying entirely within I_f contributes with weight 1, while a channel straddling an edge of I_f contributes a fractional weight. Because the intervals I_f tile $[0, N_{\rm freq}]$, the weights of any fixed input channel sum to one, $\sum_f w_{fF} = 1$; the gridding kernel is thus a delay-linearizing rebinning of the frequency axis that conserves total intensity, $\sum_f \mathrm{out}[f] = \sum_F \mathrm{in}[F]$.

Note that the tree gridding kernel is a monotone *decreasing* reindexing operation. Thus, to search for FRBs with positive dispersion (lowest frequencies arise last), we want to search over lines in the tree array with positive slope (highest tree-freq channels arrive last).

2 Tree dedispersion

After the tree gridding kernel, the next step is "tree dedispersion", which searches for straight lines with $0 \leq \text{slope} \leq 1$. (The extension to slope > 1 will be discussed in a future section, along with more details such as subband searches, early triggers, and peak-finding.)

The dedispersion input array has shape $(2^r, N_{\text{time}})$, where the leading axis is "tree-freq" (see above). Straight lines will be parameterized by an integer $0 \leq d < 2^r$, representing the delay (in time samples) between the lowest and highest tree-freq channel, and a time index t representing the arrival time in the *highest* tree-freq channel (i.e. the *latest* arrival time). Thus, the dedispersion output array also has shape $(2^r, N_{\text{time}})$, where the first axis represents a delay (or DM).[1]

Tree dedispersion is a composition of functions:

$$\big(\text{Input array}\big) \overset{\text{DD}(0)}{\longrightarrow} \big(\text{Intermediate array}\big) \overset{\text{DD}(1)}{\longrightarrow} \cdots \overset{\text{DD}(r-1)}{\longrightarrow} \big(\text{Output array}\big) \tag{5}$$

where the function $\text{DD}(k)$ is described as follows:

- The $\text{DD}(k)$ input array is a shape $(2^{r-k}, 2^k, N_{\text{time}})$ array. The first axis represents a coarse frequency channel. Each coarse-freq channel f corresponds to a block of 2^k tree-freq channels in the original tree input array:
$$2^k f \leq (\text{tree-freq index}) < 2^k(f+1) \tag{6}$$
The second axis represents a delay $0 \leq d < 2^k$ within the coarse-freq channel (6). As k increases, the coarse-freq channels double in width, and we need twice as many delay indices to cover the range $0 \leq (\text{slope}) \leq 1$.
The time index represents an arrival time at the largest tree-freq channel in the coarse-freq channel (6), i.e. the latest arrival time within the coarse-freq channel.

- The $\text{DD}(k)$ operation is defined by:

$$\begin{aligned} \texttt{out}[f, 2d, t] &= \frac{1}{\sqrt{2}}\big(\texttt{in}[2f, d, t-d] + \texttt{in}[2f+1, d, t]\big) \\ \texttt{out}[f, 2d+1, t] &= \frac{1}{\sqrt{2}}\big(\texttt{in}[2f, d, t-d-1] + \texttt{in}[2f+1, d, t]\big) \end{aligned} \tag{7}$$

- As formulated above, tree dedispersion can lead to negative time indices. By default, `in[...,t]` with $t < 0$ is defined to be zero. (In the real-time system, we run tree dedispersion in an "incremental" context where a negative time index should be interpreted as a reference to data received in a previous time chunk.)

- The $\text{DD}(0)$ input array is sometimes reshaped from $(2^r, 1, N_{\text{time}})$ to $(2^r, N_{\text{time}})$, and the $\text{DD}(r-1)$ output array is sometimes reshaped from $(1, 2^r, N_{\text{time}})$ to $(2^r, N_{\text{time}})$. Thus, we think of the entire tree dedispersion operation (5) as having input and output shapes $(2^r, N_{\text{time}})$.

3 Subband search

So far, we have described a search for FRBs which cover the full tree-freq range $0 \leq f < 2^r$. This would be suboptimal since real FRBs do not always cover the full frequency range. Next, we introduce a "sub-band" search for searching a configurable set of subbands.

[1] Frequently in the code (but not in these notes), it's convenient to use a *bit-reversed* delay index $0 \leq d_{\text{rev}} < 2^r$. (This turns out to allow some array operations to operate "in place", facilitating parallelization.)

3.1 Parameterization of subbands

Our dedispersion config files contain a length-$(R+1)$ integer-valued array of subband counts C_l.[2] Here $0 \le R \le r$, where r is the tree rank. The counts C_l parameterize a set of subbands as follows:

- Frequency subbands are indexed by a pair (l, s) where $0 \le l \le R$ and $0 \le s < C_l$. We will sometimes "flatten" the pair (l, s) into a single subband index $0 \le n < N$, where

$$N \equiv \sum_{l=0}^{R} C_l \tag{8}$$

  is the total number of subbands. (In the code, n and N are denoted by the same symbols, e.g. N is `FrequencySubbands::N`.)

- Level $l = 0$ is special: subband $(0, s)$ corresponds to tree-freq range

$$2^{r-R}s \le (\text{tree-freq index}) < 2^{r-R}(s+1) \tag{9}$$

- At levels $l > 0$, subband (l, s) corresponds to tree-freq range

$$2^{r-R+l-1}s \le (\text{tree-freq index}) < 2^{r-R+l-1}(s+2) \tag{10}$$

  Thus, level 0 consists of non-overlapping bands whose fractional width is $1/2^R$. Levels $l > 0$ consist of bands whose fractional width is $1/2^{R-l}$, and overlap by a factor 2.

- To avoid "overflowing" the band, C_l must satisfy

$$0 \le C_l \le \begin{cases} 2^R & \text{for } l = 0 \\ 2^{R-l+1} - 1 & \text{for } l > 0 \end{cases} \tag{11}$$

  A level with $C_l = 0$ is allowed, and simply contributes no subbands. We require $C_R = 1$, i.e. we always search the full frequency band. The case $C_l = \{1\}$ (with $R = 0$) corresponds to disabling subbands, and only searching the full band.

- The details of this scheme, including the "specialness" of level $l = 0$, are dictated by convenience in the GPU kernel.

- To *interpret* the C_l already in a config file, run `pirate_frb show_dedisperser -cv`. For example, for $C_l = [5, 9, 7, 3, 1]$ this prints:

```
# In this config, there are 25 top-level frequency subband(s):
#   pf_level=0: [948.7,1500.0], [750.0,948.7], [639.6,750.0], [566.9,639.6], [514.5,566.9]
#   pf_level=1: [750.0,1500.0], [639.6,948.7], [566.9,750.0], [514.5,639.6], [474.3,566.9],
#               [442.3,514.5], [416.0,474.3], [393.9,442.3], [375.0,416.0]
#   pf_level=2: [566.9,1500.0], [474.3,750.0], [416.0,566.9], [375.0,474.3], [344.1,416.0],
#               [319.8,375.0], [300.0,344.1]
#   pf_level=3: [416.0,1500.0], [344.1,566.9], [300.0,416.0]
#   pf_level=4: [300.0,1500.0]
```

- To *create* a C_l array with a given max $\Delta(\log \nu)$, run `pirate_frb make_subbands`. For example, `pirate_frb make_subbands 300 1500 0.1 -r 4` gives $C_l = [5, 9, 7, 3, 1]$. These subbands are shown visually in Figure 1.

[2] In the code, R is denoted `pf_rank`, l is denoted `pf_level`, and C_l is denoted `frequency_subband_counts`.

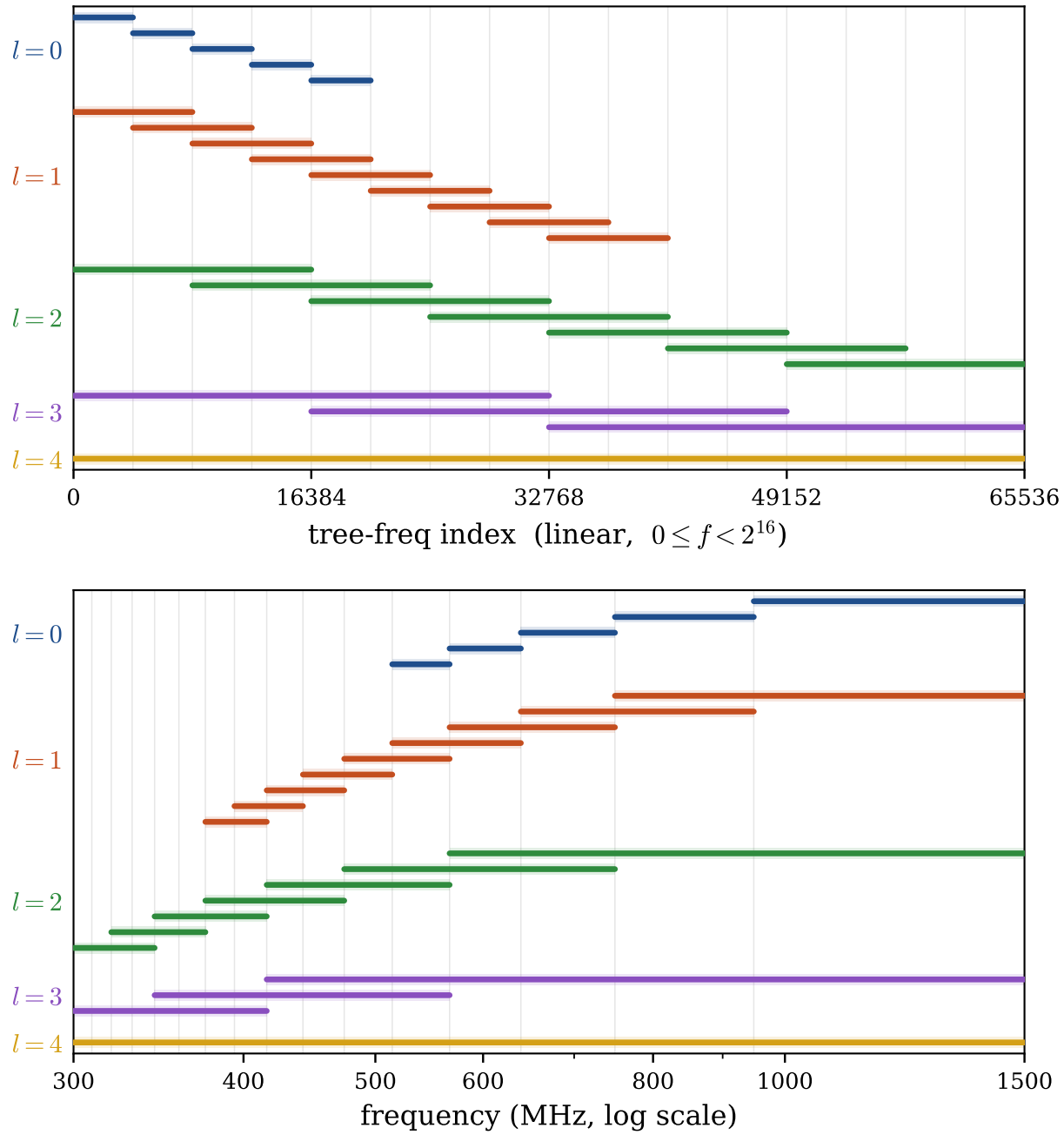

Figure 1: Frequency subbands in a CHORD config with `frequency_subband_counts=[5,9,7,3,1]`). The top panel shows the subbands in tree-freqs (where they are "naturally" defined) and the bottom panel shows the conversion to physical frequency ν in MHz. Note that the conversion between tree-freqs and ν is monotone decreasing.

3.2 Subbanded dedispersion

Subbanded dedispersion is an operation whose input array is the same as ordinary (non-subbanded) tree dedispersion (2), i.e. a shape $(2^r, N_{\rm time})$ array where the first index is tree-freq. The output array has shape $(2^{r-R}, M, N_{\rm time})$, where M is the number of "multiplets":

$$M \equiv \sum_{l=0}^{R} 2^l C_l \tag{12}$$

Each subband (l,s) "owns" a shape $(2^{r-R}, 2^l, N_{\rm time})$ subarray of the output. The first two axes correspond to 2^{r-R+l} dispersion delays across the subband, split into a "coarse" index $0 \le d_{\rm hi} < 2^{r-R}$ and a "fine" index $0 \le d_{\rm lo} < 2^l$. (This index-splitting ensures that the combined output of all subbands is a regular $(2^{r-R}, M, N_{\rm time})$ array, rather than a "ragged" object.)

Conceptually, subband dedispersion works as follows. For each subband (l,s), we apply tree dedispersion to the appropriate tree-freq range (either Eq. (9) or (10)). We also apply a time lag so that the time index corresponds to the extrapolated arrival time at the highest tree-freq in the full band, not the highest tree-freq in the subband. For an efficient implementation, we run tree dedispersion as usual (Eq. 5), and extract per-subband dedispersion outputs from intermediate arrays "on the fly", rather than dedispersing each subband "from scratch".

Formally, the details are as follows:

- **Case 1: $l = 0$ or even s.** In this case, the subband corresponds to an "aligned" tree-freq range of the form:

  $$2^{r-R+l} f \le (\text{tree-freq index}) < 2^{r-R+l}(f+1) \qquad (\text{where } 0 \le f < 2^{R-l}) \tag{13}$$

  where the coarse-freq index f is given by:

  $$f = \begin{cases} s & \text{if } l = 0 \\ s/2 & \text{if } l > 0 \end{cases} \tag{14}$$

  The subband owns a shape $(2^{r-R}, 2^l, N_{\rm time})$ subarray of the output (denote by `out`).

  In this case, the per-subband dedispersion outputs already arise as intermediates in tree dedispersion (Eq. 5); we just need to apply a time lag. Let `tmp` be the output array of DD$(k-1)$, where $k = r-R+l$. Then `tmp` has shape $(2^{R-l}, 2^{r-R+l}, N_{\rm time})$. To populate the `out` array, we slice/reshape `tmp` and apply time lags:

  $$\texttt{out}\big[d_{\rm hi}, d_{\rm lo}, t\big] = \texttt{tmp}\big[f, d, t - T\big] \tag{15}$$

  where on the RHS we define:

  $$d \equiv d_{\rm hi} 2^l + d_{\rm lo} \qquad\qquad T \equiv d_{\rm hi} 2^l (2^{R-l} - 1 - f) \tag{16}$$

- **Case 2: $l > 0$ and odd s.** In this case, the subband corresponds to a "half-aligned" tree-freq range of the form:

  $$2^{r-R+l-1}(2f+1) \le (\text{tree-freq index}) < 2^{r-R+l-1}(2f+3) \qquad (\text{where } 0 \le f < 2^{R-l} - 1) \tag{17}$$

  where the coarse-freq index f is given by $f \equiv (s-1)/2$. The subband owns a shape $(2^{r-R}, 2^l, N_{\rm time})$ subarray of the output (denote by `out`).

  This case is a little more complicated, since the per-subband dedispersion outputs are "one DD-step away" from the intermediate arrays in tree dedispersion. Let `tmp` be the output array of DD$(k-1)$,

where $k = r - R + l - 1$. Then `tmp` has shape $(2^{R-l+1}, 2^{r-R+l-1}, N_{\rm time})$.

$$\begin{aligned} \texttt{out}\big[d_{\rm hi}, 2d_{\rm lo}, t\big] &= \frac{1}{\sqrt{2}}\Big(\texttt{tmp}\big[2f+1, d, t - T_0 - d\big] + \texttt{tmp}\big[2f+2, d, t - T_0\big]\Big) \\ \texttt{out}\big[d_{\rm hi}, 2d_{\rm lo}+1, t\big] &= \frac{1}{\sqrt{2}}\Big(\texttt{tmp}\big[2f+1, d, t - T_1 - d - 1\big] + \texttt{tmp}\big[2f+2, d, t - T_1\big]\Big) \end{aligned} \tag{18}$$

where on the RHS we define:

$$d \equiv d_{\rm hi} 2^{l-1} + d_{\rm lo} \qquad\qquad T_0 = T_1 = d_{\rm hi} 2^{l-1} (2^{R-l+1} - 2f - 3) \tag{19}$$

(Warning: the quantities $f, k, d_{\rm lo}$ have slightly different meanings in cases 1 and 2!)

Note that subband dedispersion includes the full band (via $C_R = 1$), but the full-band output array is "reshaped" to $(2^{r-R}, 2^R, N_{\rm time})$, whereas in the previous section it had shape $(2^r, N_{\rm time})$.

Remark (time-lag convention). The lags T (Case 1) and $T_0 = T_1$ (Case 2) above depend only on the *coarse* delay $d_{\rm hi}$, not on the fine delay $d_{\rm lo}$: both equal $g\, d_{\rm hi}$, where g is the number of coarse channels (each 2^{r-R} tree-freq channels wide) lying above the subband. Equivalently, the 2^l fine-DM multiplets that share a given subband and coarse delay $d_{\rm hi}$ are assigned a *common* time origin, extrapolated to the full-band top from the coarse delay alone. This is the convention used by the GPU kernel and `ReferenceTree` (their `ff*dd`), and we adopt it here only to match the current code.

Arguably it would make more sense to reference each multiplet to the full-band-top arrival extrapolated from its *own* (full) subband delay, so that a given burst peaks at the same output time index for every multiplet of its DM. That choice replaces the lags above by

$$\begin{aligned} &\text{Case 1:} \quad T = d\,(2^{R-l} - 1 - f), \qquad d = d_{\rm hi} 2^l + d_{\rm lo}, \\ &\text{Case 2:} \quad T_0 = d\,(2^{R-l+1} - 2f - 3), \qquad T_1 = T_0 + (2^{R-l} - f - 2), \end{aligned} \tag{20}$$

i.e. the (floored) half-integer extrapolation, for which $T_0 \neq T_1$. We may switch the code to this convention in the future.

4 Peak-finding

The final stage of the dedispersion transform is *peak-finding*. For now, we give a partial description – a future version of the notes will describe peak-finding in more detail. The input is the shape-$(2^{r-R}, M, N_{\rm time})$ array from the previous section. The output is a shape-$(2^{r-R}, N_{\rm time}/T_{\rm ds})$ array, where $T_{\rm ds}$ is a time downsampling factor. Conceptually, peak-finding is 3 steps:

- For each $(d_{\rm hi}, m)$, the input array is a 1-d time series. We convolve the time series with P peak-finding kernels, described below, to obtain a shape $(2^{r-R}, M, N_{\rm time}, P)$ array.
- We apply weights that depend on $(d_{\rm hi}, m, p)$, to normalize the array so that each array element has variance 1. (Thus, array elements are normalized to "sigmas".)
- We downsample the array to shape $(2^{r-R}, N_{\rm time}/T_{\rm ds})$. Downsampling is done with "max" rather than addition, so each array element corresponds to max detection significance over all downsampled parameters.

In implementation, these 3 steps are coalesced into one GPU kernel for speed (and also coalesced with the final stages of tree dedispersion), e.g. the shape $(2^{r-R}, M, N_{\rm time}, P)$ is never "materialized" in GPU memory.

4.1 Peak-finding kernels

The kernels are organized into $\Lambda \equiv \max(\log_2 W_{\max}, 1)$ *levels* $0 \le \lambda < \Lambda$,[3] where $W_{\max}$ (a power of two) is the config parameter `max_kernel_width`. The total number of profiles is

$$P = 3 \log_2 W_{\max} + 1. \tag{21}$$

The building block at level λ is the *unnormalized running boxcar sum* of width 2^λ,

$$b_\lambda[t] \equiv \sum_{i=0}^{2^\lambda - 1} x[t-i], \tag{22}$$

computed by the recursion

$$b_0[t] = x[t], \qquad b_{\lambda+1}[t] = b_\lambda[t] + b_\lambda[t - 2^\lambda]. \tag{23}$$

Note that (23) is a *plain sum*: there is no $1/\sqrt{2}$ in the boxcar downsampling. (All normalization is deferred to the weights.)

At each level λ we form up to four kernels, denoted $h_{\lambda,q}$ where $0 \le q \le 3$:

$$\begin{aligned}
h_{\lambda,0} &= [\underbrace{1, \ldots, 1}_{2^\lambda}] \\
h_{\lambda,1} &= [\underbrace{1, \ldots, \ldots, 1}_{2^{\lambda+1}}] \\
h_{\lambda,2} &= [\underbrace{\tfrac{1}{2}, \ldots,}_{2^\lambda}\ \underbrace{1, \ldots,}_{2^\lambda}\ \underbrace{\tfrac{1}{2}, \ldots}_{2^\lambda}] \\
h_{\lambda,3} &= [\underbrace{\tfrac{1}{2}, \ldots,}_{2^\lambda}\ \underbrace{1, \ldots, \ldots, 1,}_{2^{\lambda+1}}\ \underbrace{\tfrac{1}{2}, \ldots}_{2^\lambda}]
\end{aligned} \tag{24}$$

The profile index $0 \le p < P$ is related to (λ, q) by $p = 3\lambda + q$, with the special value $p = 0$ for $(\lambda, q) = (0, 0)$. Thus level 0 contributes profiles $p = 0, 1, 2, 3$, and each level $\lambda > 0$ contributes $p = 3\lambda + 1, 3\lambda + 2, 3\lambda + 3$ — the single-boxcar kernel $h_{\lambda,0}$ is dropped for $\lambda > 0$, since $h_{\lambda,0} = b_\lambda = h_{\lambda-1,1}$ would be redundant.

5 Downsampled trees and early triggers

So far, we have described a single `DedispersionTree` which searches up to slope 1 (i.e. full-band dispersion delay equal to 2^r time samples), and "triggers" when the pulse arrives at the lowest frequency (highest tree-freq). It will be useful to define a larger `Dedisperser` object, consisting of multiple trees, as follows:

- **Downsampled trees** in order to search to higher DM. If we time-downsample the data by a factor $2^{\texttt{ids}}$, and run rank-r dedispersion as usual, then we can search to a max dispersion delay which is larger by a factor $2^{\texttt{ids}}$. When we time-downsample by a factor $2^{\texttt{ids}}$, we add time samples and multiply by $2^{-\texttt{ids}/2}$, in order to leave the variance unchanged (assuming uncorrelated samples).

  The integer `ids` is called the *downsampling level*, and is denoted in the code by either `ids` or `ds_level`. If $\texttt{ids} > 0$, then we only keep the upper half (in delay) of the dedispersion output, since the lower half was already searched at downsampling level $(\texttt{ids} - 1)$.

- **Early triggers** in order to reduce triggering latency, especially at high DM. By default, we only trigger when the pulse arrives at the highest tree-freq (i.e. lowest radio frequency). To trigger earlier for some bursts, we also run parallel dedispersion trees over the lowest $2^{r-\delta}$ tree-freqs, where $\delta = 1, 2, \cdots$ is

[3] In the code these are denoted `l` and `L`; we use λ and Λ here to avoid ambiguity with the subband level l in §3.

denoted `delta_rank` in the code. These trees trigger at lower tree-freq (by a factor 2^δ), i.e. higher radio frequency.

A `Dedisperser` consists of one or more `DedispersionTrees`, each identified by a $(\texttt{ids}, \delta)$ pair. The "base" tree $(\texttt{ids}, \delta) = (0, 0)$ does not downsample or trigger early. Conceptually, each `DedispersionTree` just runs "vanilla" rank-$(r - \delta)$ tree dedispersion on the first $2^{r-\delta}$ tree-freqs, downsampled in time by $2^{\texttt{ids}}$, and then drops the lower DM-half of the outputs if $\texttt{ids} > 0$.[4]

As a concrete example, Table 1 and Figures 2–3 show the ten-tree family of the CHORD config `chord_sb2_et.yml` ($r = 16$, with four downsampling levels and six early triggers).

`ids`	δ	Full-band delay range (s)	DM range ($\mathrm{pc\,cm^{-3}}$)	trigger (MHz)	latency range (s)
0	0	0.0–65.5	0–1481	300	0.0–65.5
1	0	65.5–131.1	1481–2962	300	65.5–131.1
	1			416	32.8–65.5
2	0	131.1–262.1	2962–5924	300	131.1–262.1
	1			416	65.5–131.1
	2			567	32.8–65.5
3	0	262.1–524.3	5924–11847	300	262.1–524.3
	1			416	131.1–262.1
	2			567	65.5–131.1
	3			750	32.8–65.5

Table 1: The ten `DedispersionTree`s of the CHORD config `chord_sb2_et.yml` (rank $r = 16$; four downsampling levels; six early triggers). Each tree is identified by an $(\texttt{ids}, \delta)$ pair. The DM range and *full-band* dispersion delay range depend only on `ids`. The trigger frequency depends only on δ, and the latency range is $2^{-\delta} \times$ (delay range). (Generated with `pirate_frb show_dedisperser -v configs/dedispersion/chord_sb2_et.yml`.)

[4] This description suggests that each tree is an independent computation run "from scratch". There are a lot of tricks in the code that reuse computations between trees, i.e. the `Dedisperser` is faster than running the constituent `DedispersionTrees` independently.

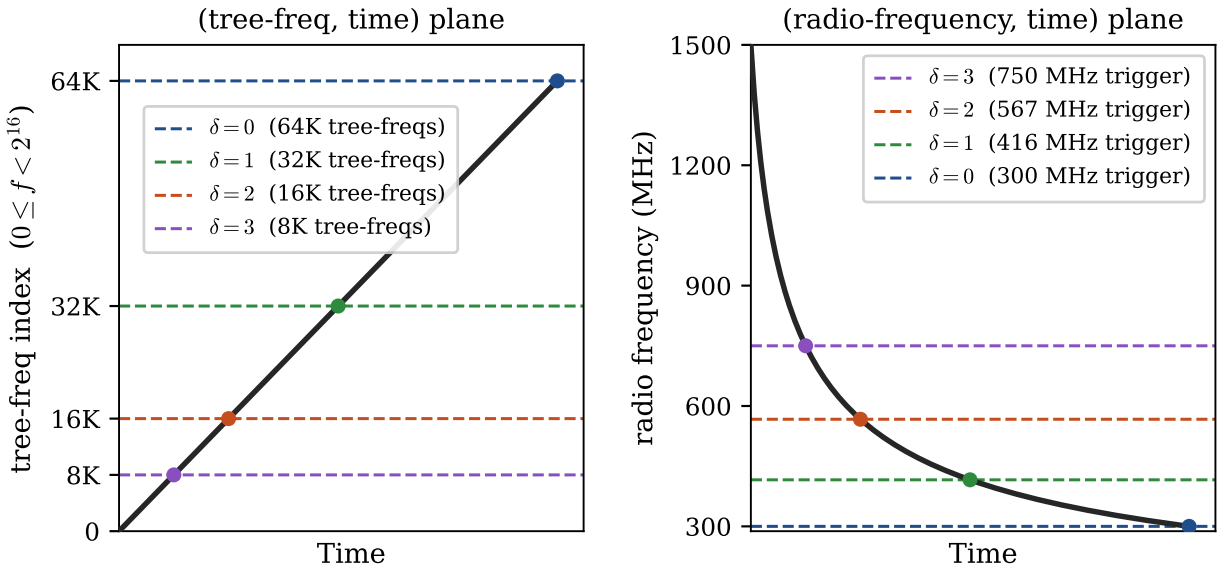

Figure 2: A visualization of early triggers, in radio frequency (right panel) or the "tree-freq" reparametrization used in the code (left panel). Trees with early triggers search the lower part of the tree-freq range, i.e. the high part of the radio frequency range. A tree with $\delta > 0$ triggers earlier (by a factor 2^δ) than the main tree. The plot assumes CHORD params (300–1500 MHz, $r = 16$, $0 \le \delta \le 3$).

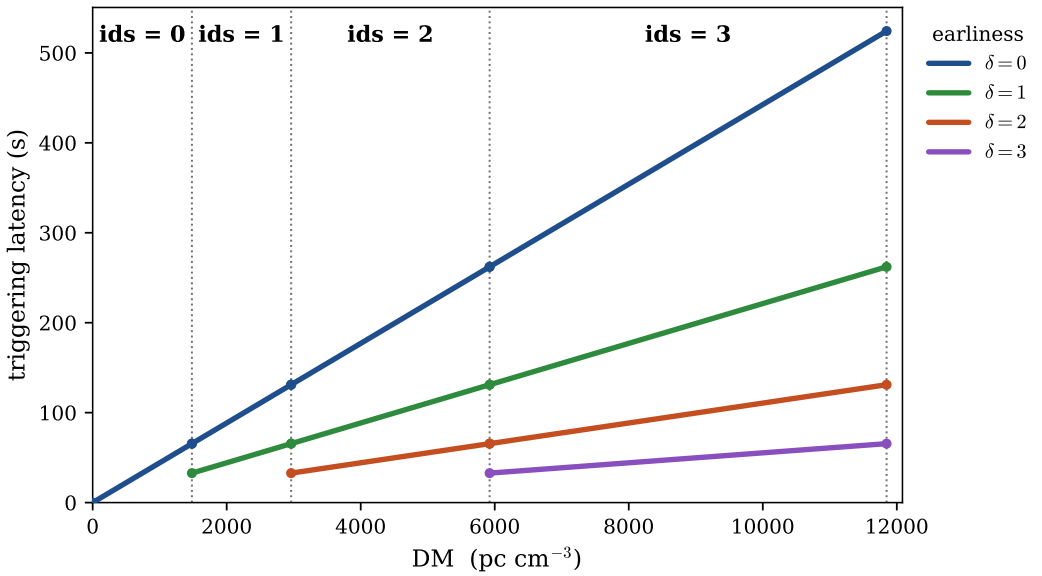

Figure 3: The ten `DedispersionTrees` from Table 1, shown as line segments in the (DM, triggering latency) plane. For each tree $(\texttt{ids}, \delta)$, the value of `ids` determines the DM range, and the value of δ determines the slope.

6 Dedispersion output arrays

6.1 Parameter definitions

The pirate `DedispersionPlan` defines the following global (i.e. not per-tree) parameters:

$T_{\rm in} =$ `nt_in` (input time samples per time chunk)

$r_{\rm top} =$ `toplevel_rank` (tree rank specified in toplevel config file; denoted r in earlier sections)

and the following per-tree parameters:

`ids` = `ds_level` (downsampling level of tree)

$\delta =$ `delta_rank` ("earliness" of trigger)

$T_{\rm ds} =$ `time_downsampling` (time coarse-graining factor in peak-finding kernel)

$D_{\rm ds} =$ `dm_downsampling` (dm coarse-graining factor in peak-finding kernel)

These parameters can be inspected with `pirate_frb show_dedisperser -v`, and are also passed from pirate to the grouper via the `dedispersion_plan` yaml metadata.

6.2 Details of output arrays

The dedispersion output consists of a pair of 2-d arrays (`out_max`, `out_argmax`) for each tree, indexed by (dm,time). (In the code, there is also a beam axis, omitted in the notes for simplicity). The per-tree array shape (`ndm_out`, `nt_out`) is given by:

$$\texttt{ndm\_out} = \begin{cases} 2^{r_{\rm top}-\delta}/D_{\rm ds} & \text{if } \texttt{ids} = 0 \\ 2^{r_{\rm top}-\delta-1}/D_{\rm ds} & \text{if } \texttt{ids} \geq 1 \end{cases} \tag{25}$$

$$\texttt{nt\_out} = \frac{T_{\rm in}}{2^{\texttt{ids}} T_{\rm ds}} \tag{26}$$

The *full-band* delay range searched by each tree (expressed as an integer number samples, before any per-tree time downsampling has been done) is:

$$d_{\rm lo} = \begin{cases} 0 & \text{if } \texttt{ids} = 0 \\ 2^{r_{\rm top}+\texttt{ids}-1} & \text{if } \texttt{ids} \geq 1 \end{cases} \qquad d_{\rm hi} = 2^{r_{\rm top}+\texttt{ids}} \tag{27}$$

To derive Eqs. (25)–(27), we walk through steps in the code:

- On the time axis, we start with a length-$T_{\rm in}$ input array, downsample by a factor $2^{\texttt{ids}}$, run tree dedispersion, and downsample the output (max-ing, not averaging!) by $T_{\rm ds}$. This implies Eq. (26).
- The DM axis is a bit more complicated. First consider the base tree (no downsampling or early triggers), and assume $D_{\rm ds} = 1$ for simplicity. We have:

  $$d_{\rm lo} = 0 \qquad d_{\rm hi} = 2^{r_{\rm top}} \qquad \texttt{ndm\_out} = 2^{r_{\rm top}} \tag{28}$$

  Then we include additional complications one at a time. If we downsample the input by $2^{\texttt{ids}}$, then $d_{\rm hi}$ gets multiplied by $2^{\texttt{ids}}$. Recall that for $\texttt{ids} > 0$, we also discard the lower half of the output DMs (which was already searched at downsampling level $\texttt{ids} - 1$). Therefore:

  $$d_{\rm lo} = \begin{cases} 0 & \text{if } \texttt{ids} = 0 \\ 2^{r_{\rm top}+\texttt{ids}-1} & \text{if } \texttt{ids} > 0 \end{cases} \qquad d_{\rm hi} = 2^{r_{\rm top}+\texttt{ids}} \qquad \texttt{ndm\_out} = \begin{cases} 2^{r_{\rm top}} & \text{if } \texttt{ids} = 0 \\ 2^{r_{\rm top}-1} & \text{if } \texttt{ids} > 0 \end{cases} \tag{29}$$

If this is an early-triggered tree with $\delta > 0$, then the *full-band* delay range is unchanged, but the spacing between trial DMs increases by 2^δ, so `ndm_out` decreases by the same factor:

$$d_{\rm lo} = \begin{cases} 0 & \text{if } \texttt{ids} = 0 \\ 2^{r_{\rm top}+\texttt{ids}-1} & \text{if } \texttt{ids} > 0 \end{cases} \qquad d_{\rm hi} = 2^{r_{\rm top}+\texttt{ids}} \qquad \texttt{ndm\_out} = \begin{cases} 2^{r_{\rm top}-\delta} & \text{if } \texttt{ids} = 0 \\ 2^{r_{\rm top}-\delta-1} & \text{if } \texttt{ids} > 0 \end{cases} \tag{30}$$

Finally, we downsample the output (max-ing, not averaging!) by $D_{\rm ds}$. This decreases `ndm_out` by a factor $D_{\rm ds}$, leaving $d_{\rm lo}$, $d_{\rm hi}$ unchanged. This implies Eqs. (25), (27).

6.3 Converting output indices to DM and arrival time

In the grouper, a peak in the (dm,time) plane is identified by a pair of indices $0 \le \texttt{idm} < \texttt{ndm\_out}$ and $0 \le \texttt{it} < \texttt{nt\_out}$. Before sending the event to the sifter, we must convert to a physical DM and arrival time.

For a tree without an early trigger (i.e. $\delta = 0$), we convert `idm` and `it` to d and t as follows:

$$d = d_{\rm lo} + \left(d_{\rm hi} - d_{\rm lo}\right) \frac{\texttt{idm} + 0.5}{\texttt{ndm\_out}} \qquad\qquad t = T_{\rm in} \frac{\texttt{it} + 0.5}{\texttt{nt\_out}} \tag{31}$$

where the quantities $d_{\rm lo}$, $d_{\rm hi}$, `ndm_out`, $T_{\rm in}$, `nt_out` were defined above (in the previous subsections). Then we convert d (full-band delay, in "units" of the input time sample length) to DM, and convert t (arrival time relative to the start of the time chunk, in "units" of the input time sample length) to a physical time.

If the tree has an early trigger (i.e. $\delta > 0$), then there is an extra subtlety. The value of d in Eq. (31) is the *full-band* dispersion delay, but the time t in Eq. (31) is the arrival time *at the trigger frequency*. The sifter expects the arrival time *at the lowest frequency in the full band*, which is given by:[5]

$$t' = t + \left(1 - 2^{-\delta}\right) d \tag{32}$$

[5] In the code, Eqs. (31), (32) appear in `FrbGrouper.create_events()`, which converts array indices to DMs and arrival times. Note that for now, we place each (dm,time) at the center of its coarse-grained output array element. In the future, we'll add code to recover the fine-grained position, using the `out_argmax` array. (This is not as trivial as it sounds, since the `out_argmax` array uses an "encoding" that's convenient in the GPU kernel, and I need to write code to decode it.)