# Inverting a Polyphase Filter Bank

## 1 Introduction

Polyphase filter banks (PFBs) are widely used in digital signal processing to provide flat response within frequency channels and excellent rejection of out-of-band signals. Several current and upcoming radio arrays (CHIME, HIRAX, ALBATROS, CHORD) will record baseband data that has gone through a PFB, which will later be post-processed. A regular feature of the post-processing is to rechannelize data since finer frequency resolution than can be provided by the original (usually FPGA-based) PFBs is often required. The natural way to do this is to invert the PFB to return to the original electric field samples. Unfortunately, quanitzation effects generally will induce spurious noise into the inverse-PFB, so in practice, a carefully constructed pseudo-inverse will be preferred. To see how such a pseudo-inverse should be made, we first show how the forward PFB is constructed using the notation of linear algebra.

## 2 Forward PFB

The goal of the PFB is to spit a stream of electric field data up into frequency channels, where the response inside a channel is flat, and the response outside the channel dies as quickly as possible. If we had a very long electric field timestream, the ideal thing to do would be to Fourier transform the electric field, then cut out a boxcar in Fourier space. The PFB tries to do this as well as possible and as efficiently as possible given the real-world constraints of digital signal processing.

If we have the Fourier transform of a long data stretch and wish to split up the data into frequency bins, and then sum the signal in the bins, we can do this by convolving the Fourier transform with a boxcar, then sampling the ensuing convolved data with a spacing set by the boxcar width. Invoking the convolution theorem, convolving a Fourier transform with a boxcar is equivalent to multiplying the real space by the transform of a boxcar, or a sinc function. This is the first key step in a PFB. For additional out-of-band rejection, a further window (*e.g.* Hamming, Hanning...) is often used. The second key insight is to note that if we only want every $m^{th}$ frequency bin, we can split the timestream into $m$ chunks, add them together, and Fourier transform that. Effectively, the fact that we want poorer frequency resolution than the long timestream provides means we are aliasing pieces of the timestream onto each other. With these steps, we can now carry out a PFB. If we want $n_{chan}$ channels, and we want to use a boxcar of width $n_{tap}$, then to get the PFB of a single chunk of data, we need to:
1) Take $2n_{chan}n_{tap}$ entries of the original timestream.
2) Apply a window to those samples consisting of a sinc times an (optional) window.

3) Split the windowed timestream into $n_{tap}$ segments and add those together.
4) Carry out the discrete Fourier transform of that stacked, windowed timestream. This gives us $n_{chan}$ complex numbers, as desired. We then repeat the process, shifting over in the original timestream by $2n_{chan}$ samples.

To undo the PFB, it is helpful to write down the forward PFB as a series of matrix operators. We can write the individual chunk PFB as follows:

$$\mathrm{PFB} = \mathrm{FSW}d$$

Where $d$ are the raw data, W is the $2n_{chan}n_{tap}$ diagonal matrix that applies the sinc/window function to the raw timestream, S is the $2n_{chan}$ by $2n_{chan}n_{tap}$ matrix that splits the windowed timetream into $n_{tap}$ chunks and adds them together, and F is the (complex) $n_{chan}$ by $2n_{chan}$ real-to-complex Fourier transform operator. Note that S is just a set of $n_{tap}$ identity matrices of size $2n_{chan}$ stacked horizontally. Consequently, SW just splits the window function into $n_{tap}$ pieces, turns then into diagonal matrices, and horizontally stacks those blocks. To carry out the PFB of a long chunk of data, we just take FSW, shift it to the right by $2n_{chan}$ elements (corresponding to shifting the piece of $d$ we PFB), and stack that shifted matrix below.

## 3 Inverting the PFB

To undo the effects of the PFB, note first that we can multiply each block of the output by $\mathrm{F}^{-1}$, which is just the inverse DFT. This leaves us with real data, and we are left needing to undo the effects of SW. The key fact about the nature of this matrix is that its form as a stack of diagonal matrices only mixes samples separated by $2n_{chan}$. After we take the IDFT, then, we are left with $2n_{chan}$ decoupled problems. When we extract the bit of SW corresponding to each decoupled chunk, we are left with a band-diagonal Toeplitz matrix with $n_{tap}$ diagonals given by taking every $2n_{chan}^{th}$ entry of the window function.

Formally, this is not an soluble system, since we always have more input data points than output PFB points, and so we will need to *something* to fill in the missing data. Due to missing data, we will never be able to accurately reconstruct the ends of the inverse PFB, so we might as well pick a scheme that simplifies life and lets us understand how the inverse procedure is working. We will, of course, need to verify that the effects of whatever choice we make leave us with a practically useful solution.

In this note, we advocate introducing circulant boundary conditions. In particular, we take the first few samples of the IFT'd PFB blocks and repeating them at the end, and extending the blocks of SW to make them square and circulant. Why? Because the eigenvectors of a circulant matrix are sine waves, and the eigenvalues are just the entries of the Fourier transform of a row. We can apply the inverse by convolving with the the IFT of the inverse of the DFT. Since the natural way to do that convolution is via Fourier transfoms, so we can get back to the original samples by Fourier transforming the IDFT'd PFB slice, dividing that by the Fourier transform of the first row of the matching block of

SW (which will consist of $n_{tap}$ non-zero entries, followed by zeros), and taking the inverse Fourier transform of that. The original timestream will then be these $2n_{chan}$ blocks stiched together, and we're done. Since the inverse requires a second round of Fourier transforms, we expect it will be a constant factor of a few slower than the forward PFB (although for problem sizes chosen to keep the FFT sizes the product of small primes, out current CPU implementation of the inverse is actually slightly *faster* than the forward PFB). Additionally, we note that these sorts of operations are extremely well suited to GPUs, with the computationally intensive parts relying on pre-existing, optimized libraries.

The FFT-based PFB inversion can be coded quite compactly in Python. We provide an implementation here that mirrors the call in Richard Shaw's PFB inversion utility. The entire routine, including optional Wiener filtering, is only a dozen lines of Python.

```
def inverse_pfb_fft_filt(dat,ntap,window=pfb.sinc_hamming,thresh=0.0):
    dd=np.fft.irfft(dat,axis=1)
    win=window(ntap,dd.shape[1])
    win=np.reshape(win,[ntap,len(win)//ntap])
    mat=np.zeros(dd.shape,dtype=dd.dtype)
    mat[:ntap,:]=win
    matft=np.fft.rfft(mat,axis=0)
    ddft=np.fft.rfft(dd,axis=0)
    if thresh>0:
        filt=np.abs(matft)**2/(thresh**2+np.abs(matft)**2)*(1+thresh**2)
        ddft=ddft*filt
    return np.fft.irfft(ddft/np.conj(matft),axis=0)
```

## 4 Noise-filtering the PFB Inverse

Anyone inverting PFBs will quickly notice that the restored timestream is frequently poorly behaved. If the PFB has been quantized (as it almost certainly has been due to data volume concerns), then the restored timestream has regularly spaced noise spikes, which also give rise to a "picked fence" set of corrupted frequencies in rechannelizations. The circulant approximation lets us 1) understand where the noise spikes come from, and 2) provides options in how to deterministically filter them. One might naively expect that the noise spikes come from zeroes in the sinc used in the window function. This is wrong. Instead, note that if we ever had our entries in a block of SW be of the form [$abba$0000...] (for a 4-tap PFB), then any sine wave that was odd across the first four entries would give zero in the DFT. For a 4-tap PFB, the entries never quite reach perfect symmetry, but they get close and in the worst case, the smallest eigenvalue goes to about 0.2168 divided by the number of channels. This wouldn't matter if we were working in arbitrary precision (since the forward PFB would damp power in these modes), but quantization effectively introduces noise that gets amplified in the inverse. The eigenvalue spectra for a 4-tap, 1024-channel PFB are shown in Figure 1.

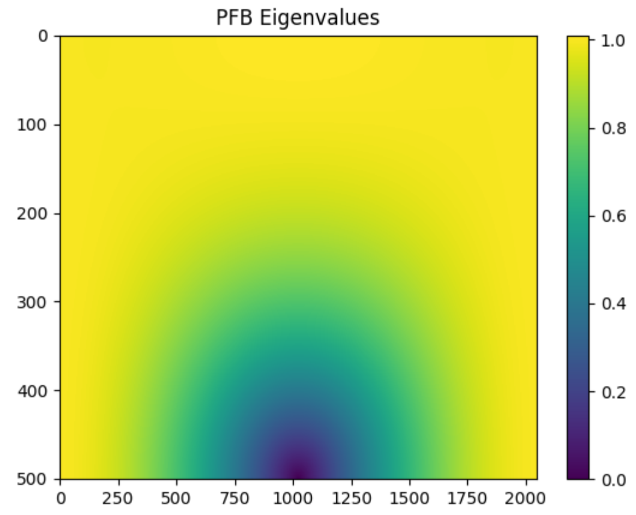

Figure 1: Eigenvalues of a 4-tap, 1024-channel PFB, assuming stationarity. The poorly conditioned region arises from the window function sampled by the sliced of the PFB being nearly symmetric.

The circulant formulation gives us a powerful tool to filter out this noise. The usual formulation is to use a Wiener filter $\mathrm{Sig}\,(\mathrm{Sig}+\mathrm{Noise})^{-1}$. We can of course do this, but because the circulant formulation gives us the explicit eigendecomposition of the transform matrix, we can choose *any* eigenvalue filtering scheme we desire. In Figure 2, we show the reconstruction errors induced by quantizing the PFB to 4 bits, and the reconstruction errors when using a Wiener filter. The original (unquantized) points are in blue, the quantized errors are in yellow, and the filtered quantized errors are in green. Figure 3 shows the (square root of the) binned power spectrum of the errors in the reconstructed timestreams without (blue) and with (yellow) a Wiener filter. There is no avoiding some level of stripes here since the quantization fundamentally loses information, but the Wiener filter quite substantially reduced the power in the noisy frequencies. (Simon will investigate various filtering schemes and we may update these figures. We expect the unfiltered to get worse as the number of channels goes up, but the filtered reconstruction seems to stay about the same, since the same fraction of data is lost. It's just lost even more thoroughly in the higher $n_{chan}$ PFBs.)

Finally, we note that our diagnosis of the source of the quantization noise suggests obvious paths towards reducing it. Ill-conditioning is introduced by symmetry in the sub-sampled window function, so a well-chosen skewed window function might be able to substantially reduce the quantization noise. Of course,

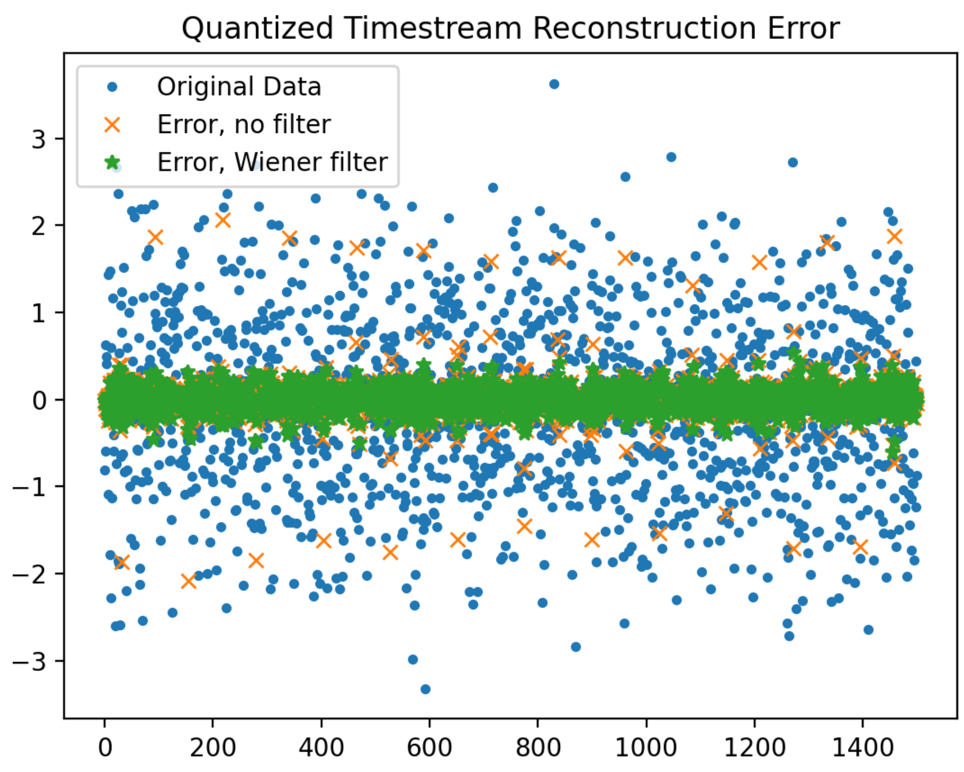

Figure 2: Reconstruction errors showing the effects of quantization on a 32-channel PFB. Original (unquantized) points are blue, reconstruction errors are in yellow, and Wiener-filtered reconstruction errors are in green. The worst errors are substantially reduced by using the Wiener filter.

such a window function would also need to avoid anti-symmetry as well (since then sine waves that are even over the first $n_{tap}$ entries would give zero) while also maintaining the beneficial properties of the PFB. While it may be tricky to find such a window, it is fortunately very easy to test - just Fourier transform the sub-sampled window and look for minimal dynamic range. If a suitable window were found, it should be easy to implement in practice since the only change required to FPGA firmware would be to update the window coefficients, with everything else proceeding exactly as before.

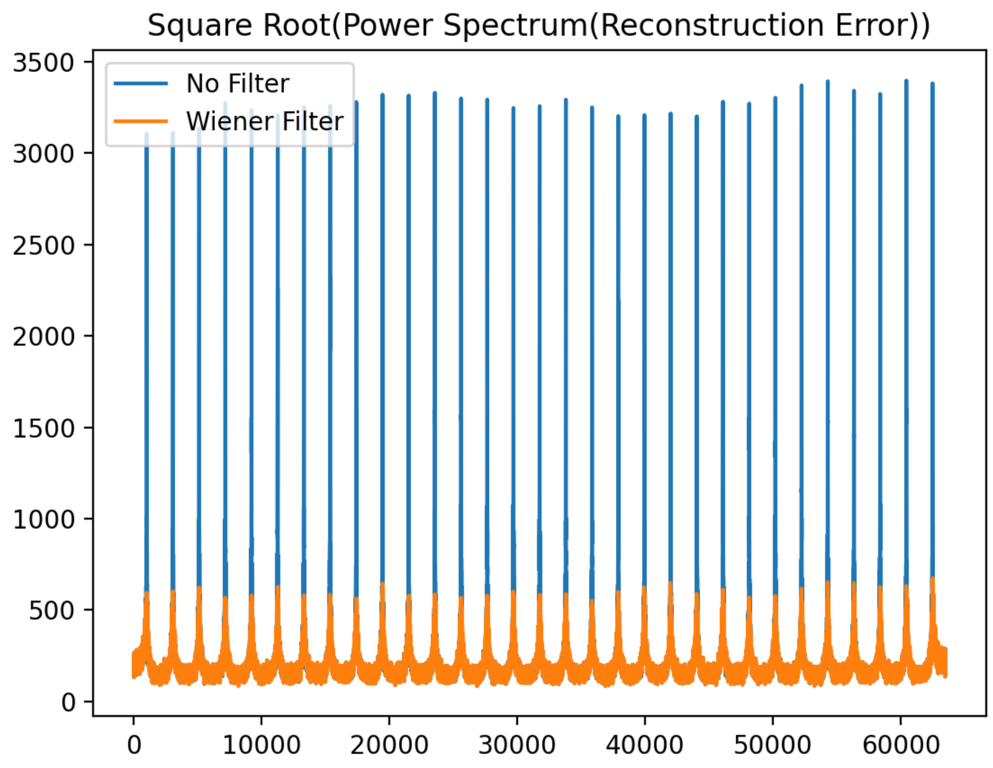

Figure 3: (Square root) of the binned power spectrum errors from the simulation in Figure 2. The noisy frequencies are still noisy, but the power in the reconstruction errors in those bad channels is reduced by about an order of mangitude when using the Wiener filter.