Filter Analysis and Solutions

Rich Ormiston

January 2020

Note About the Analysis

The filter coefficients found in this analysis will be those which minimize the mean square error of each of the filters under consideration (as opposed to L1 norm, root mean square etc.)

These filters are being constructed in a repository called **The Refinery** which can be found on GitLab `here`

1 Wiener Filter

Assume that we have the system signal $d[k]$ and we have an estimate of the noise to the system given by

$$y[k] = \sum_{m=0}^{M-1} a_k[m]x[k-m] \tag{1}$$

where $a_k[m]$ is the filter coefficient at the k^{th} time step and at tap m and the signal $x[k-m]$ is the witness channel to the noise in the system signal at time step k and delay tap m.
Since the system output signal $d[k]$ and the witness channel of the noise we wish to regress, $x[k]$ are known, then we just need to solve for the optimal filter coefficients. To do so, we minimize the expectation value of the mean square error with respect to the filter coefficient tap.

$$\frac{\partial}{\partial a_k[n]}\big\langle d[k] - y[k]\big\rangle^2 = \frac{\partial}{\partial a_k[n]}\big\langle -2d[k]y[k] + y^2[k]\big\rangle \tag{2}$$

$$= \Big\langle 2d[k]x[k-n]\Big\rangle + \left\langle \sum_{m=0}^{M-1} a_k[m]x[k-m]x[k-n]\right\rangle \tag{3}$$

$$= 0 \tag{4}$$

Letting the cross-correlation vector be represented as $\langle 2d[k]x[k-n]\rangle = p[k]$ and letting the auto-correlation matrix be $\left\langle \sum_{m=0}^{M-1} a_k[m]x[k-m]x[k-n]\right\rangle = \sum_{m=0}^{M-1} a_k[m]R[m-n]$, then we may write this in matrix form as follows

$$\begin{pmatrix} p[0] \\ p[1] \\ \vdots \\ p[M-1] \end{pmatrix} = \begin{pmatrix} R[0] & R[1] & \cdots & R[M-1] \\ R[1] & R[0] & \cdots & R[M-2] \\ \vdots & & \ddots & \vdots \\ R[M-1] & R[M-2] & \cdots & R[0] \end{pmatrix} \begin{pmatrix} a[0] \\ a[1] \\ \vdots \\ a[M-1] \end{pmatrix} \tag{5}$$

Or equivalently, since

$$p[k] = \sum_{m=0}^{M-1} a_k[m]R[m-n] \rightarrow \mathbf{p} = \mathbf{a}\hat{\mathbf{R}} \tag{6}$$

therefore we find that the optimal filter coefficients for the Wiener filter are given by

$$\mathbf{a} = \hat{\mathbf{R}}^{-1}\mathbf{p} \tag{7}$$

2 LMS Adaptive Filter

One of the drawbacks of the Wiener filter is that the analysis occurs once over the dataset meaning that non-stationary noise cannot be removed (the Wiener filter is "wide sense stationary"). The LMS adaptive filter addresses this by creating a new set of filter coefficients at every time step. The filter coefficients are updated based upon the error signal and the step size:

$$\mathbf{a}[k+1] = \mathbf{a}[k] - \mu\vec{\nabla}e^2[k] \tag{8}$$

where the error signal is given by $e[k] = d[k] - y[k]$. Assuming that the filter may be given as in Equation 1, we then find for the *instantaneous* update

$$\frac{\partial e^2[k]}{\partial a_k[n]} = 2e[k]\frac{\partial e[k]}{\partial a_k[n]} = -2e[k]\frac{\partial y[k]}{\partial a_k[n]} = -2e[k]x[k-n] \tag{9}$$

where $x[k]$ is the input noise signal vector given as

$$\mathbf{x}[k] = \begin{pmatrix} x[k] \\ x[k-1] \\ \vdots \\ x[k-M+1] \end{pmatrix} \tag{10}$$

Thus we find that the filter coefficients update as

$$\mathbf{a}[k+1] = \mathbf{a}[k] + 2\mu e[k]\mathbf{x}[k] \tag{11}$$

3 Slow Second Order Volterra Filter

We now consider a bilinear filter (input is the product of two channels) wherein one of the channels changes slowly relative to the other. The general 2nd order filter noise estimate with witness channels x_1 and x_2 and filter coefficient matrix $a_k[m, m']$ is given by

$$y[k] = \sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} a_k[m, m'] x_1[k-m] x_2[k-m'] \tag{12}$$

Before solving the full second order coefficients, we first assume that one of the witness channels, say x_2 changes slowly relative to x_1 across each batch of $M-1$ taps. For example, we could imagine that x_1 encodes the 60 Hz line and x_2 is an ASC channel resonating at 0.5 Hz. If we had data sampled at 512 Hz and $M = 10$, then in those 10 taps, we will see more than a full cycle of the 60 Hz signal and only about one percent of the 0.5 Hz signal. Therefore, over the duration of those $M-1$ taps, it is reasonable to assume that x_2 is in fact a single, stationary value. Therefore we may write the slow second order filter as

$$y_s[k] = x_2[k] \sum_{m=0}^{M-1} a_k[m] x_1[k-m] \tag{13}$$

We can now recognize this as a generic linear filter, but multiplied (modulated) by a secondary channel. We can either feed this "new" filter into the Wiener filter analysis or the LMS adaptive filter depending upon the stationarity of the data under consideration. For the adaptive filter, the coefficients update as

$$\mathbf{a}[k+1] = \mathbf{a}[k] + 2\mu e[k] x_2[k] \mathbf{x_1}[k] \tag{14}$$

This is precisely how Gabriele's NonSens subtraction works (except that he does it in frequency space).

4 General Second Order Volterra Filter

In the event that both input channels shown in Equation 12 change on the same timescale, then we may not simplify the filter any further. We must however decide if the filter coefficients are to be updated with each time step or if there is a single set of coefficients (data is wide-sense stationary). Let us solve both of those methods now.

4.1 Adaptive Second Order Volterra Filter

To find the filter coefficients here we simply insert the noise estimate (Eq. 12) into the coefficient update algorithm (Eq 8). Doing so we find

$$a_{k+1}[m, m'] = a_k[m, m'] + 2\mu e[k] x_1[k-m] x_2[k-m'] \tag{15}$$

This may be cast into a matrix form as

$$\hat{\mathbf{a}}_{k+1} = \hat{\mathbf{a}}_k + 2\mu e[k](\mathbf{x_1}[k] \cdot \mathbf{x_2}^T[k]) \tag{16}$$

$$= \hat{\mathbf{a}}_k + 2\mu e[k] \begin{pmatrix} x_1[k] \\ x_1[k-1] \\ \vdots \\ x_1[k-M+1] \end{pmatrix} \begin{pmatrix} x_2[k] & x_2[k-1] & \cdots & x_2[k-M+1] \end{pmatrix} \tag{17}$$

4.2 Complete Second Order Volterra Filter

Let us now assume that the noise we wish to regress is stationary and bilinear. We may proceed then as we did in the case of the Wiener filter, though we will see that the solution is a bit more complicated even if it is conceptually identical. Minimizing the MSE ($\partial \left\langle e^2[k] \right\rangle / \partial a_k[i,j]$) leads to

$$\left\langle d[k]x_1[k-i]x_2[k-j] \right\rangle = \sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} a_k[m,m'] \left\langle x_1[k-m]x_2[k-m']x_1[k-i]x_2[k-j] \right\rangle \tag{18}$$

The term on the LHS is just a cross-correlation matrix of the input signals multiplied by the system signal. The 4-point correlation is more difficult to compute, however it is made more simple if we may break it up as the sum of the permutations of the products of the 2-point correlations (i.e., if it is a multivariate normal distribution). This means that we could write the RHS as

$$\begin{aligned} \left\langle x_1[k-m]x_2[k-m']x_1[k-i]x_2[k-j] \right\rangle &= \left\langle x_1[k-m]x_2[k-m'] \right\rangle \left\langle x_1[k-i]x_2[k-j] \right\rangle \\ &+ \left\langle x_1[k-m]x_1[k-i] \right\rangle \left\langle x_2[k-m']x_2[k-j] \right\rangle \\ &+ \left\langle x_1[k-m]x_2[k-j] \right\rangle \left\langle x_2[k-m']x_1[k-i] \right\rangle \\ &= R^{(1,2)}[m-m']R^{(1,2)}[i-j] \\ &+ R^{(1,1)}[m-i]R^{(2,2)}[m'-j] \\ &+ R^{(1,2)}[m-j]R^{(1,2)}[m'-i] \\ &\equiv C^{(i,j)}[m,m'] \end{aligned} \tag{19}$$

But this is **not true** for LIGO data! We must do the full 4-point correlation and solve for the rank-4 tensor. So instead let

$$C^{(i,j)}[m,m'] \equiv \left\langle x_1[k-m]x_2[k-m']x_1[k-i]x_2[k-j] \right\rangle$$

Now, let's look at the term on the LHS of Equation 18. Letting $\left\langle d[k]x_1[k-i]x_2[k-j]\right\rangle \equiv P^{(1,2)}[i-j]$ then putting it together we may write

$$P^{(1,2)}[i-j] = \sum_{m=0}^{M-1}\sum_{m'=0}^{M-1} a_k[m,m']C^{(i,j)}[m,m'] \tag{20}$$

The LHS is a matrix of size $M \times M$ and therefore the RHS must be as well. Look however at $C^{(i,j)}[m,m']$. For each (i,j) pair that we pick, we are still left with an $M \times M$ matrix from the m and m' terms. The filter coefficients are also a matrix of size $M \times M$. In order to get a scalar then, it seems like we need to "dot" these matrices. This is hard to understand, though the direct product is easy to code (albeit expensive to run). There is another way to visualize the algebra here; instead of a matrix of matrices dotting yet another matrix, we flatten the $C^{(i,j)}[m,m']$ matrix to a vector of size M^2 and we do the same to the coefficients. For example, if i,j,m and m' all run from 0 to 1, then

$$\hat{\mathbf{a}} = \begin{pmatrix} a[0,0] & a[0,1] \\ a[1,0] & a[1,1] \end{pmatrix} \rightarrow \begin{pmatrix} a[0,0] \\ a[0,1] \\ a[1,0] \\ a[1,1] \end{pmatrix} \tag{21}$$

Looking at Equation 21 we can start running through i and j and do the sum. The $i = 0 = j$ term would be

$$\begin{aligned} P^{(1,2)}[0] = {} & C^{(0,0)}[0,0]a[0,0] + C^{(0,0)}[0,1]a[0,1] \\ & + C^{(0,0)}[1,0]a[1,0] + C^{(0,0)}[1,1]a[1,1] \end{aligned} \tag{22}$$

Similarly, the $i = 1$, $j = 0$ term would be

$$\begin{aligned} P^{(1,2)}[1] = {} & C^{(1,0)}[0,0]a[0,0] + C^{(1,0)}[0,1]a[0,1] \\ & + C^{(1,0)}[1,0]a[1,0] + C^{(1,0)}[1,1]a[1,1] \end{aligned} \tag{23}$$

Collecting everything we see that we can write this as a matrix multiplication problem

$$\begin{pmatrix} P^{(1,2)}[0] \\ P^{(1,2)}[1] \\ P^{(1,2)}[1] \\ P^{(1,2)}[0] \end{pmatrix} = \begin{pmatrix} C^{(0,0)}[0,0] & C^{(0,0)}[0,1] & C^{(0,0)}[1,0] & C^{(0,0)}[1,1] \\ C^{(0,1)}[0,0] & C^{(0,1)}[0,1] & C^{(0,1)}[1,0] & C^{(0,1)}[1,1] \\ C^{(1,0)}[0,0] & C^{(1,0)}[0,1] & C^{(1,0)}[1,0] & C^{(1,0)}[1,1] \\ C^{(1,1)}[0,0] & C^{(1,1)}[0,1] & C^{(1,1)}[1,0] & C^{(1,1)}[1,1] \end{pmatrix} \begin{pmatrix} a[0,0] \\ a[0,1] \\ a[1,0] \\ a[1,1] \end{pmatrix} \tag{24}$$

Lastly, we can put this all in matrix notation to get

$$\vec{\mathbf{P}} = \hat{\mathbf{C}}\vec{\mathbf{a}}$$

$$\rightarrow \vec{\mathbf{a}} = \hat{\mathbf{C}}^{-1}\vec{\mathbf{P}} \tag{25}$$

The $\hat{\mathbf{C}}$ matrix still has a lot of components to it, so getting the final form of that matrix is tedious. For example, by using Equation 19 we can find the $i = j = m = m' = 0$ term explicitly

$$\begin{aligned} C^{(0,0)}[0,0] &= R^{(1,2)}[0]R^{(1,2)}[0] + R^{(1,1)}[0]R^{(2,2)}[0] + R^{(1,2)}[0]R^{(1,2)}[0] \\ &= 2\left(R^{(1,2)}[0]\right)^2 + R^{(1,1)}[0]R^{(2,2)}[0] \end{aligned} \tag{26}$$

Since each term is composed of 6 terms, then there are a total of $(4 \times 4) \times 6 = 96$ terms in the $\hat{\mathbf{C}}$ matrix in the 2-tap case. More generally, there will be $(M^2 \times M^2) \times 6 = 6M^4$. We can see an immediate limitation to this bilinear filter, the size of the matrix grows with the tap length as M^4 as opposed to the Wiener filter which grows as M^2. A tap length of 32 would lead to $\sim 10^6$ matrix elements which is about as big as we can reasonably invert in a usable amount of time.

5 Non-linear Input to Linear Filters

There is another way to regress bilinear or $\mathcal{O}(2)$ noise from a system signal; we feed in some pre-coupled data and pretend as if it is a genuine linear input. For example, to regress non-stationary bilinear noise with an LMS filter we can use the following as input

$$\mathbf{x}_{(1,2)}[n] = \begin{pmatrix} x_1[n]x_2[n] \\ x_1[n]x_2[n-1] \\ \vdots \\ x_1[n]x_2[n-N] \\ \vdots \\ x_1[n-N+1]x_2[n-N] \\ x_1[n-N]x_2[n-N] \end{pmatrix} \tag{27}$$

This would result in the coefficient update equation

$$\mathbf{a}_{(1,2)}[n+1] = \mathbf{a}_{(1,2)}[n] - 2\mu e[n]\mathbf{x}_{(1,2)}[n] \tag{28}$$

To make sure that loud transients do not change the filter coefficients for many iterations down the line, we can introduce a "leak" coefficient, α. Additionally, we can normalize the step-size by the power in the M-tap chunk of data we are considering in order to curb the slow converge due to a large difference in eigenvalues (consequence of a large power difference in the spectrum). This leads to (including a stabilization factor ψ)

$$\mathbf{a}_{(1,2)}[n+1] = (1-\alpha)\mathbf{a}_{(1,2)}[n] - \frac{2\mu}{\mathbf{x}^T_{(1,2)}[n]\mathbf{x}_{(1,2)}[n] + \psi} e[n]\mathbf{x}_{(1,2)}[n] \tag{29}$$

We could apply this same logic to a Wiener filter. That is, we assume some coupling between channels and feed this coupled channel into the filter as if it is a single channel and we attempt to remove the "linear" noise as represented by the input channel. This obviously requires nothing more than a Wiener filter or a single-channel LMS filter (and is consequently very fast and lightweight), so why should we go through the trouble of calculating the full second order Volterra kernel?

The reason is that there is only one set of weights for a linear filter. Consider an M-tap Wiener filter into which we are feeding a direct product of channels, $x[n] \equiv x_1[n] \cdot x_2[n]$. We will end up with M coefficients which encode how $x[n]$ couples into the system signal. This means that the contribution of $x_1[n]$ and $x_2[n]$ must be the same within the system. This is not necessarily true. Contrast this with the coefficient matrix given in Equation 21. Here, each channel can change independently of the other or be coupled in various, changing ways. In other words, we would have to either know ahead of time what the coupling of the noise is, or be very lucky in order to regress non-linear noise with a linear filter and nonlinear input.

If we *do* know the form of the noise, then this can be a good way to go. Consider the modulation of the power lines (@ 60 Hz) due to the ASC noise (@ 0.5 Hz) resulting in power at 60 ± 0.5 Hz. Let us call the channels x_{ASC} and x_{MAINS}. We can then take the product of the channels giving $x = x_{ASC} \cdot x_{MAINS}$ and then bandpass the input x. This will remove the side bands around the power line quickly and safely (if this is done in the time domain it may require a phase shift). An ordinary linear filter will then remove the 60 Hz line itself. In this way, we can in principle demodulate any channels which allow it.
While we cannot recover the non-linear-input-to-a-linear-filter results directly from the complete second order filter, we can determine where the differences between the two reside. First, from Equation 18, we note that there will only two terms to cross-correlate instead of four. Because of this, we must have that $m = m'$ and $i = j$. Setting $x_1[n] \cdot x_2[n] = x_{12}[n]$, then Equation 18 reduces to

$$\left\langle d[k]x_{12}[k-j] \right\rangle = \sum_{m=0}^{M-1} a_k[m] \left\langle x_{12}[k-m]x_{12}[k-j] \right\rangle \tag{30}$$

Although the form and procedures are similar, the mathematical outcomes are not. In the full second order result, we cross-correlate the individual witness channels and then multiply the resulting matrices. In the non-linear input case the situation is reversed: we multiply the witness channels and cross-correlate the results. And since the product of cross-correlations is not the same as the cross-correlation of a product in general, these filters behave rather differently under most circumstances.

5.1 The Extended Wiener Filter

The extended Wiener filter subtracts stationary, poly-coherent noise. We now generalize the results of the Wiener Filter section to account for a noise estimate which is comprised of B input signals of tap length M:

$$y[k] = \sum_{b=0}^{B-1} \sum_{m=0}^{M-1} a_k^{(b)}[m] x^{(b)}[k-m] \tag{31}$$

Minimizing the MSE gives

$$\frac{\partial \left\langle e^2[k] \right\rangle}{\partial a_k^{(b')}[j]} = -2 \left\langle d[k] \begin{pmatrix} x^0[k-j] \\ x^1[k-j] \\ \vdots \\ x^{B-1}[k-j] \end{pmatrix} \right\rangle + 2 \left\langle y[k-j] \begin{pmatrix} x^0[k-j] \\ x^1[k-j] \\ \vdots \\ x^{B-1}[k-j] \end{pmatrix} \right\rangle \tag{32}$$

More generally, we may write

$$\begin{aligned} \frac{\partial \left\langle e^2[k] \right\rangle}{\partial a_k^{(b')}[j]} &= -2 \left(p^{(0)}[j] \delta^{b',0} + p^{(1)}[j] \delta^{b',1} + \cdots \right) \\ &= \left\{ \sum_{m=0}^{M-1} a_k^{(0)}[m] R^{(0,0)}[m-j] \delta^{b',0} + \sum_{m=0}^{M-1} a_k^{(1)}[m] R^{(1,0)}[m-j] \delta^{b',0} + \cdots \right. \\ &= \sum_{m=0}^{M-1} a_k^{(0)}[m] R^{(0,1)}[m-j] \delta^{b',1} + \sum_{m=0}^{M-1} a_k^{(1)}[m] R^{(1,0)}[m-j] \delta^{b',1} + \cdots \\ &= + \cdots \\ &= \left. + \sum_{m=0}^{M-1} a_k^{(0)}[m] R^{(0,B-1)}[m-j] \delta^{b',B-1} + \sum_{m=0}^{M-1} a_k^{(1)}[m] R^{(0,B-1)}[m-j] \delta^{b',B-1} + \cdots \right\} \end{aligned} \tag{33}$$

In matrix form this becomes

$$\begin{pmatrix} \mathbf{p}^{(0)} \\ \mathbf{p}^{(0)} \\ \vdots \\ \mathbf{p}^{(B-1)} \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{R}}^{(0,0)} & \hat{\mathbf{R}}^{(1,0)} & \cdots & \hat{\mathbf{R}}^{(B-1,0)} \\ \hat{\mathbf{R}}^{(0,1)} & \hat{\mathbf{R}}^{(1,1)} & \cdots & \hat{\mathbf{R}}^{(B-1,1)} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{R}}^{(0,B-1)} & \hat{\mathbf{R}}^{(1,B-1)} & \cdots & \hat{\mathbf{R}}^{(B-1,B-1)} \end{pmatrix} \begin{pmatrix} \mathbf{a}^{(0)} \\ \mathbf{a}^{(1)} \\ \vdots \\ \mathbf{a}^{(B-1)} \end{pmatrix} \tag{34}$$

The cross-correlation and filter coefficient 'vectors' have dimensions $B \times M$ and the correlation matrix (only auto-correlation along the diagonal) has dimensions $B \times B$ where each entry $\hat{\mathbf{R}}$ is an $M \times M$ matrix.

6 Correlation: Visual Analysis

Given the arrays

$$\begin{aligned}\mathbf{d} &= [1, 2, 3, 4] \\ \mathbf{x} &= [5, 6, 7, 8]\end{aligned} \tag{35}$$

then the correlation is calculated by dotting the arrays, permutating $\mathbf{x}$ and then repeating the process for $2N+1$ terms where N is the length of the input arrays. Pictorially,

$$\begin{gathered}[0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0] \\ [5, 6, 7, 8, 0, 0, 0, 0, 0, 0, 0, 0] \\ = 0\end{gathered}$$

$$\begin{gathered}[0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0] \\ [0, 5, 6, 7, 8, 0, 0, 0, 0, 0, 0, 0] \\ = 8\end{gathered}$$

$$\vdots$$

$$\begin{gathered}[0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0] \\ [0, 0, 0, 0, 0, 0, 0, 5, 6, 7, 8, 0] \\ = 20\end{gathered}$$

$$\begin{gathered}[0, 0, 0, 0, 1, 2, 3, 4, 0, 0, 0, 0] \\ [0, 0, 0, 0, 0, 0, 0, 0, 5, 6, 7, 8] \\ = 0\end{gathered}$$

$$\therefore \operatorname{corr}(\mathbf{d}, \mathbf{x}) = [0, 8, 23, 44, 70, 56, 39, 20, 0] \tag{36}$$

7 Three-Point Static Correlation: Visual Analysis

Given the arrays

$$\begin{aligned}\mathbf{d} &= [1, 2, 3, 4] \\ \mathbf{x} &= [5, 6, 7, 8] \\ \mathbf{y} &= [9, 10, 11, 12]\end{aligned} \tag{37}$$

then the static 3-point correlation

$$\langle d[k]\, x[k-i]\, y[k-j] \rangle \tag{38}$$

is calculated by dotting the arrays, permutating **x** and **y** and then repeating the process for $(2N+1)^2$ terms where N is the length of the input arrays. Note, the **d** array does not cycle, it just sits there while the other arrays move around it. Pictorially,

$$\begin{gathered}
[0,0,0,0,1,2,3,4,0,0,0,0] \\
[5,6,7,8,0,0,0,0,0,0,0,0] \\
[9,10,11,12,0,0,0,0,0,0,0,0] \\
= 0 \\
\\
[0,0,0,0,1,2,3,4,0,0,0,0] \\
[0,5,6,7,8,0,0,0,0,0,0,0] \\
[9,10,11,12,0,0,0,0,0,0,0,0] \\
= 8 \\
\vdots \\
[0,0,0,0,1,2,3,4,0,0,0,0] \\
[0,0,0,0,0,0,0,5,6,7,8,0] \\
[9,10,11,12,0,0,0,0,0,0,0,0] \\
= 20 \\
\\
[0,0,0,0,1,2,3,4,0,0,0,0] \\
[0,0,0,0,0,0,0,0,5,6,7,8] \\
[9,10,11,12,0,0,0,0,0,0,0,0] \\
= 0 \\
\vdots \\
[0,0,0,0,1,2,3,4,0,0,0,0] \\
[0,5,6,7,8,0,0,0,0,0,0,0] \\
[0,9,10,11,12,0,0,0,0,0,0,0] \\
= 88 \\
\vdots \\
[0,0,0,0,1,2,3,4,0,0,0,0] \\
[0,0,0,0,0,0,0,0,0,5,6,7,8] \\
[0,0,0,0,0,0,0,0,9,10,11,12]
\end{gathered} \tag{39}$$

8 Beyond the Thee Point-Static Correlation

The story for the three and four-point correlations are much the same as above. For the three-point correlation we need to calculate

$$\langle d[k-m]\, x[k-i]\, y[k-j] \rangle \tag{40}$$

To do this, we 'roll' each term one time step, multiply all of the arrays and sum, then rinse and repeat until every permutation of the three channels has been made.
The four-point correlation is conceptually no different. We calculate

$$\left\langle x_1[k-m] x_2[k-m'] x_1[k-i] x_2[k-j] \right\rangle \tag{41}$$

This adds another loop, but with the use of padding, it boils down to $(2N+1)^4$ multiplications where N is the length of the arrays. This can get pretty slow pretty fast. So, we can instead realize that we only need to "roll" the arrays over the $(2M+1)$ taps we care about, and therefore we are down to $(2M+1)^4$ multiplications which is plenty fast enough to be able to run something like this online. The code to calculate this is on the Refinery git repository linked at the top of the document (a function called `four_point_corr()` in `library.py` and `volterra.py`)

9 Recap of Filter Expansions So Far

Up to second order, the filter expansion can be written as

$$y[k] = b_0 + \sum_{m=0}^{M-1} a_k[m] x[k-m] + \sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} a_k[m,m'] x_1[k-m] x_2[k-m'] \tag{42}$$

or using the generalized Wiener filter

$$y[k] = b_0 + \sum_{b=0}^{B-1} \sum_{m=0}^{M-1} a_k^{(b)}[m] x^{(b)}[k-m] + \sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} a_k[m,m'] x_1[k-m] x_2[k-m'] \tag{43}$$

I am not sure if the second order term extends the same way, but perhaps we can generalize further to the (probably computationally ridiculous) extension

$$\begin{aligned} y[k] = b_0 &+ \sum_{b=0}^{B-1} \sum_{m=0}^{M-1} a_k^{(b)}[m] x^{(b)}[k-m] \\ &+ \sum_{b=0}^{B-1} \sum_{b'=0}^{B-1} \sum_{m=0}^{M-1} \sum_{m'=0}^{M-1} a_k^{(b,b')}[m,m'] x_1^{(b)}[k-m] x_2^{(b')}[k-m'] \end{aligned} \tag{44}$$

Liberally adopting an Einstein summation convention and dropping the k dependence, Equation 44 may be succinctly rewritten as

$$y = b_0 + a_m^b x_m^b + a_{m,m'}^{b,b'} x_{1,m}^b x_{2,m'}^{b'} \tag{45}$$

Equation 45 is the next to be implemented into `The Refinery`. Further nonlinear adaptive filters will be made and the jobs will be parallelized on condor in order to combat the combinatorics issue with the channel selection process.