Multi-Rate Processing and Polyphase Filtering with Plug-In Example

Will Pirkle

Oversampling theory is covered in detail in many DSP books and the math can be a little much to deal with at first. In Plug-Ins, we seek to use sample rate conversion in order to up-sample an input signal, process it at a higher sample rate, then down-sample it back to the original sample rate. This is often done with non-linear processing like overdrive and distortion effects. Up-sampling (oversampling) is done by interpolating points into the bitstream. Downsampling (decimation) is done by discarding points from the bitstream. These operations raise or lower the effective sample rate.

The easiest way to get into oversampling is to build a Plug-In to implement the basic interpolation and decimation processes. Polyphase filtering is a way to do sample rate conversion that is more efficient. When you try to optimize your original Plug-In code, you will actually be doing the essence of Polyphase filtering (without actually knowing?). Then, Polyphase will be easy to grasp; its one of those DSP things that is easier to implement than it is to talk about mathematically, at least at first.

Interpolation - increasing the sample rate

The up-sampling or interpolation block diagram shows how to up-sample by a factor of L (if L = 3, the new sample rate is 3X the original).

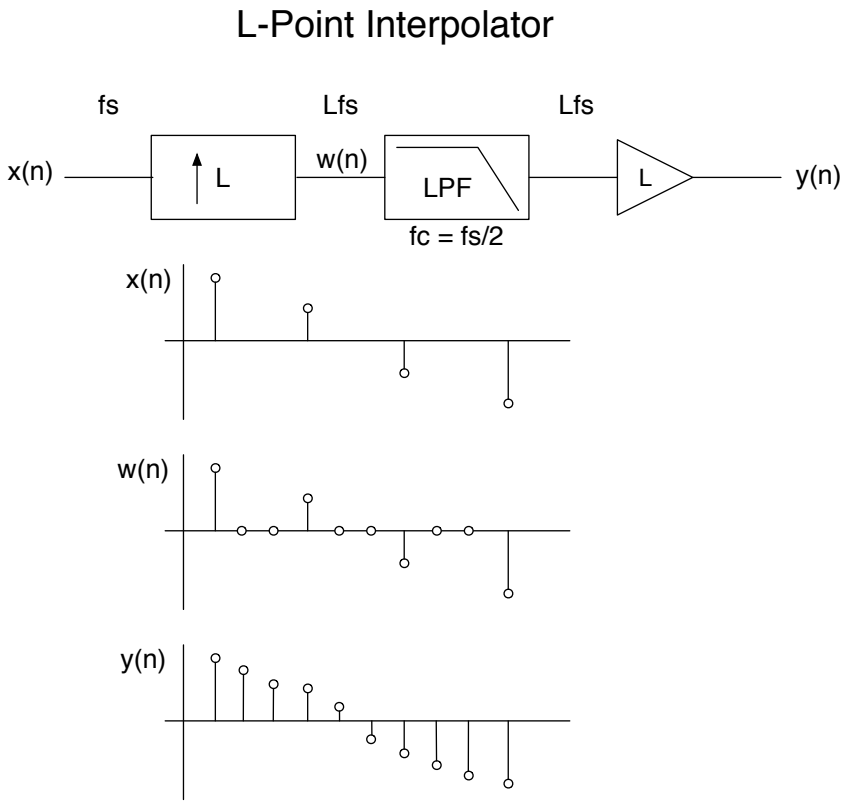


Figure 1.1: L-Point Interpolator block diagram and signals

The Interpolator consists of an up-sampler that inserts each input point ***plus*** L-1 zeros into the bitstream followed by a lowpass filter whose cutoff frequency is set to the original Nyquist frequency. However, this filter will be processing L times as many points so it is running at a sample rate of Lfs. This filter must be designed at that higher sample rate. The low-pass filter effectively interpolates the missing points during its convolution with the input. See my book, Chapter 1. The multiplier L is required at the end to gain the signal up by that factor; this is due to the fact that the convolution spreads the impulse response energy across L samples; the multiplier makes up that loss.

Interpolator:
- outputs L samples for every sample that enters
- uses a steep LPF to perform interpolation
- LPF can be designed with RackAFX FIR Design Tool, Optimal (Parks-McClellan) Method

Decimation - decreasing the sample rate

The down-sampling block diagram shows how to down-sample by a factor of M (if M = 3, the new sample rate is 1/3 the original).

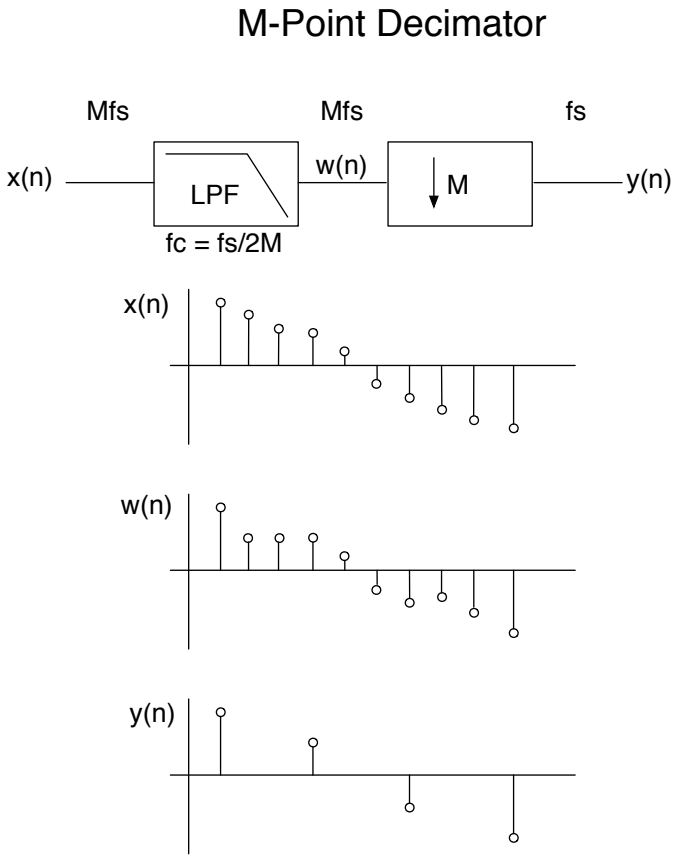


Figure 1.2: M-Point Decimator block diagram and signals

First, the signal is Low Pass Filtered with a cutoff of the target Nyquist frequency. This is to band-limit the input in just the same way you low pass filter the analog input. We need this to eliminate frequencies

above our Nyquist frequency. After filtering, M-1 samples are discarded resulting in the lower sample rate. The Low Pass Filter must be designed at that higher sample rate since it is running at the higher rate.

Decimator:
- outputs 1 sample for every M-samples that enter
- uses a steep LPF to prevent aliasing from the higher-rate input signal
- LPF can be designed with RackAFX FIR Design Tool, Optimal (Parks-McClellan) Method

Interpolator Operation - the details

To begin, we need to look specifically at the LPF's operation. It is usually implemented as a FIR filter using standard linear convolution, though this can be done with IIR filters. The LPF is shown as a combination of an input delay-line buffer and an impulse response (IR) buffer. The convolution produces the sum-of-products combination of the two buffers. We can label this filter simply H(z) in the frequency domain and represent it as shown at the bottom.

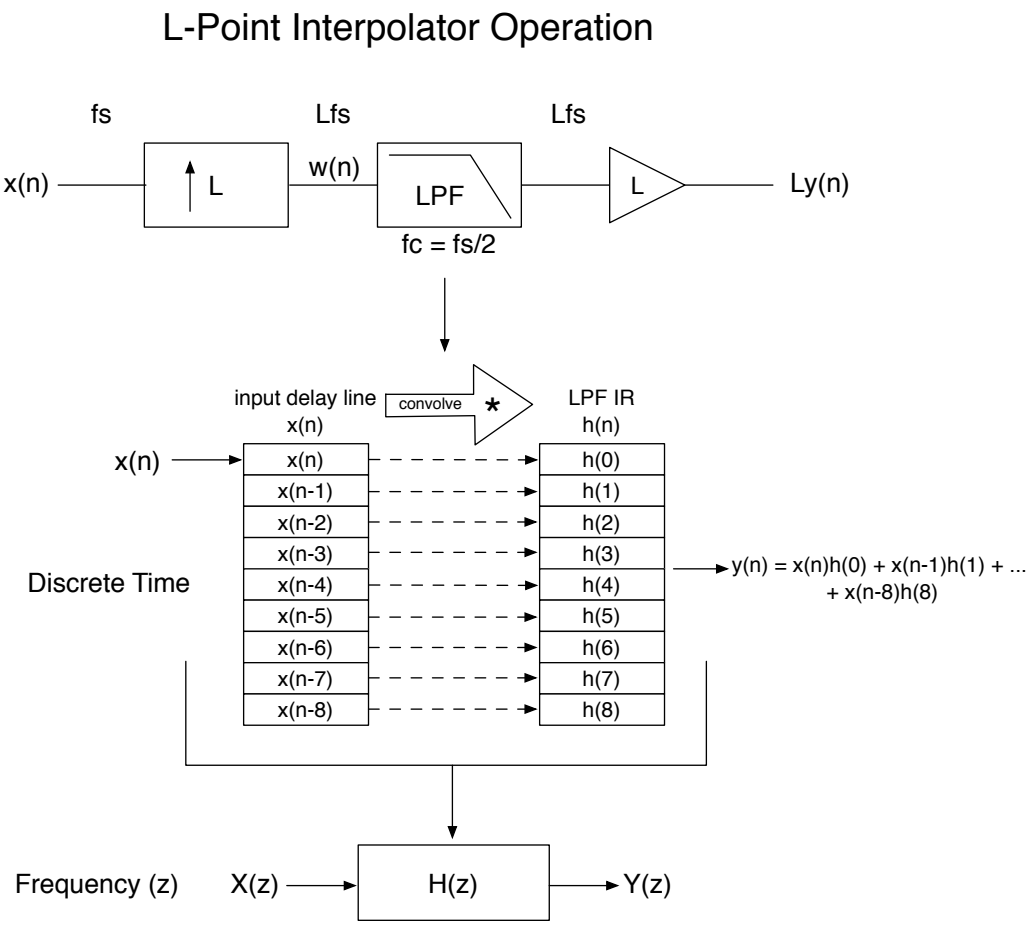


Figure 1.3: L-Point Interpolator details

The Flowchart for the L-Point Interpolator is:

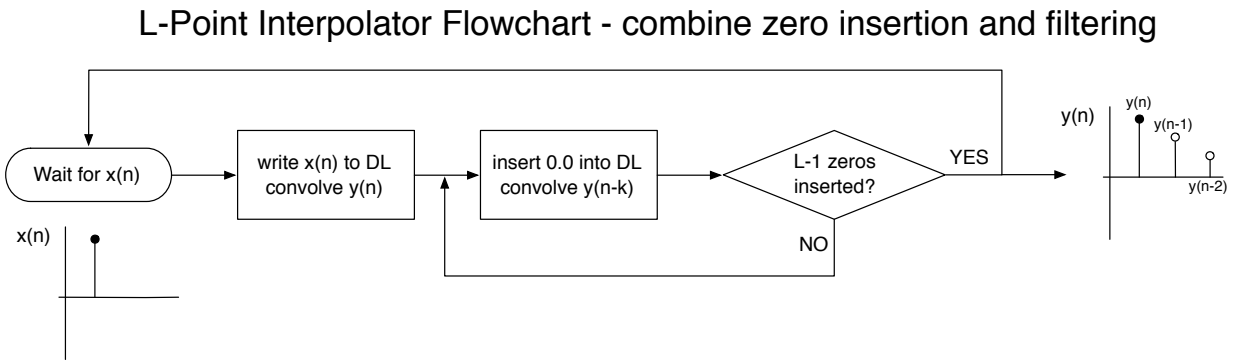


Figure 1.4: L-Point Interpolator flowchart

You can see that the two operations (zero insertion and convolution) are combined. One input sample produces 3 output samples, so this is a 3-Point Interpolator. The operation involves inserting either the input sample x(n) (the first time through the loop) or zeros into the delay line. A convolution is performed after each input value is applied which results in that output value y(n).

Example:
L = 3

For polyphase, FIR length and therefore input delay line is simplest if it is a factor of L, in this example I am using a 9-tap FIR.

Suppose the operation has been going on for some time and we have a new input x(n). The operations would look like this:

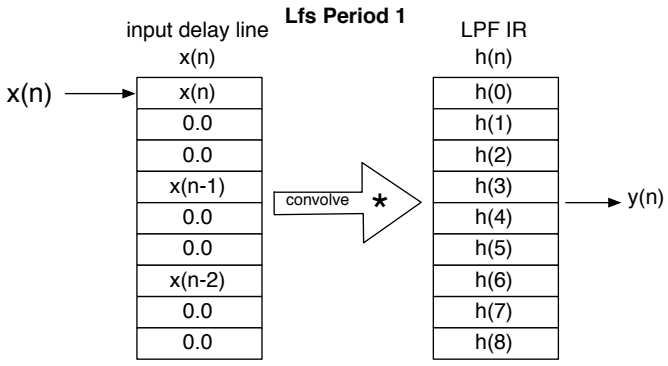


Figure 1.5: The input x(n) is applied and a convolution takes place producing y(n).

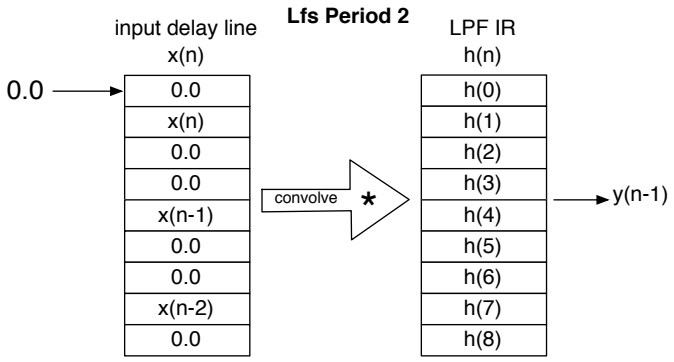


Figure 1.6: Next, a zero is inserted and another convolution produces the next output.

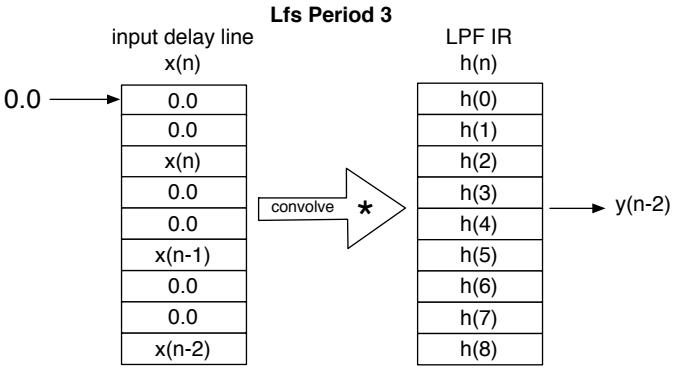


Figure 1.7: The last zero is inserted (up to L-1) and the final value computed.

That's all there is to it - the code is pretty simple and its really just a convolution that happens L times for each input. The most complicated part is designing a LPF that meets your system criteria.

But, you can see that the delay line is full of zeroes and convolving with all these zeros is inefficient; they have no effect on the final outputs sum. To optimize this in your plug-in, you go to the processing loop and figure out how to ignore the zero-value cells in the arrays during the loop.

Interpolator: decomposing the FIR into Polyphase Filters

Another way to think about making the system more efficient is on the algorithm side, not the code side. What we really want is to do the LPFiltering first (at the slower sample rate) and then insert the zeros. Its more efficient to run the filter at the slower rate; it won't have the zeros in it. But, how do we deal with inserting the zeros later without destroying the signal?

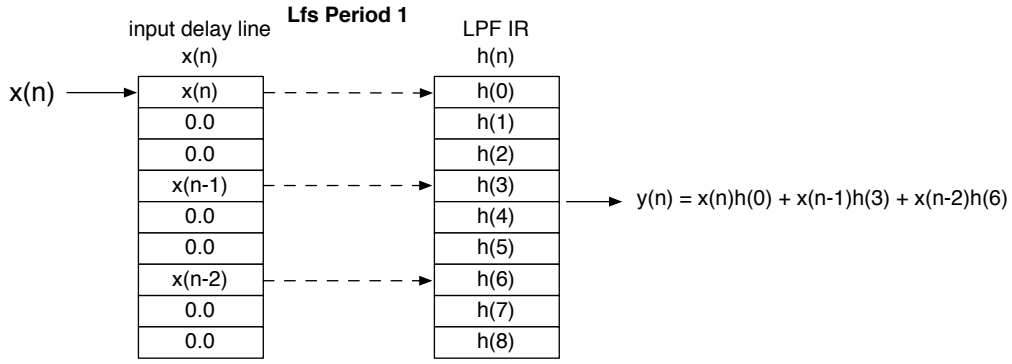


Figure 1.8: If you look a the first convolution and ignore the 0.0 values, you can see the output y(n) is a sum of L products. Its really a mini-convolution that only uses 1/L the number of coefficient/data pairs. Now look at the second Lfs period:

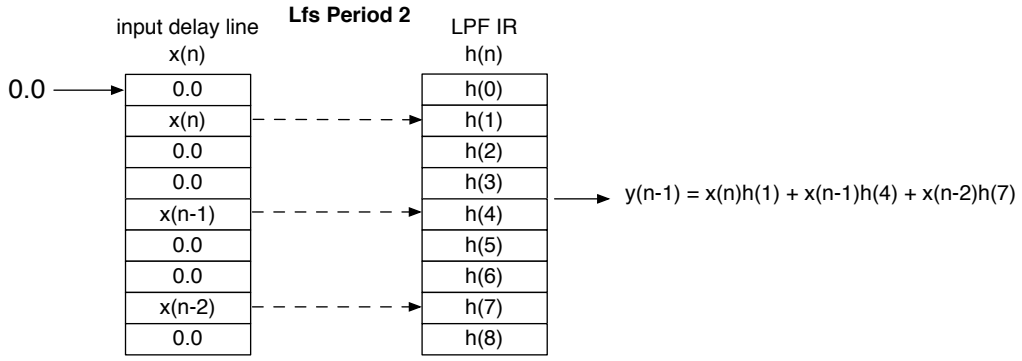


Figure 1.9: This results in another mini-convolution with the next set of IR coefficients. Finally, the last period implements this filter:

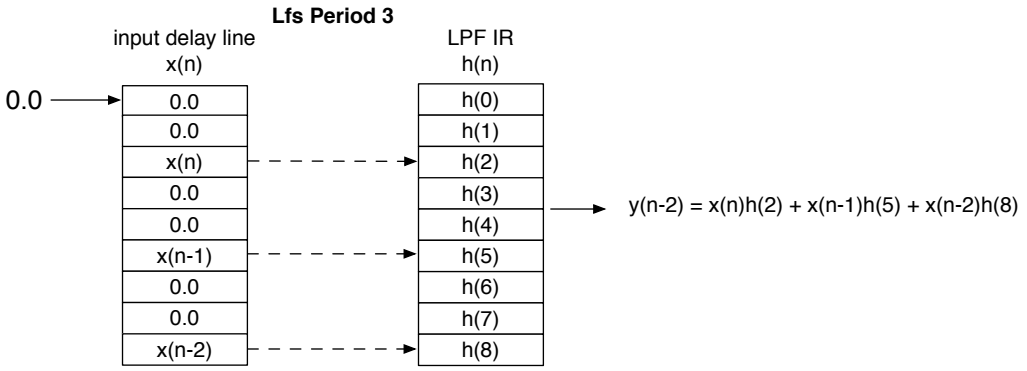


Figure 1.10: the last mini-convolution

NOTE: the same input sequence is applied three times to produce the outputs. Only the filter coefficients change!

Polyphase Decomposition

You can conceptually crack the original FIR filter into L pieces, each containing every L-th coefficient:

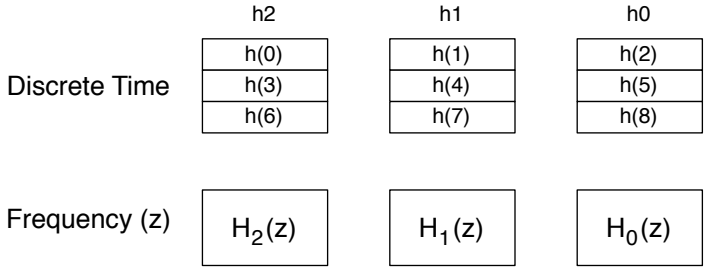


Figure 1.11: breaking the original filter into polyphase sub-band filters

This produces L (or 3 here) mini-FIR filters each with an impulse response h and a Frequency Transfer Function, H(z). Don't worry if the indexing of the filters looks "backwards." These little impulse responses in the mini-filters are made up of pieces of the original; none of them are, within themselves, a complete impulse response of a LPF. They are called sub-band filters or polyphase filters or sub-band polyphase filters. So you can see that you don't really design a polyphase filter; you take an existing filter and decompose it into the polyphase sub-filters. Decomposition follows a pretty simple rule about skipping through the original IR by L.

The decomposition of our 9-tap FIR filter results in three 3-tap polyphase (mini) filters. The last three steps to complete the algorithm are:

1) the input delay line is now shortened to the same length as the polyphase filters. It will not always be L as in this example as it depends on the length of the original FIR filter.
2) perform the filtering FIRST, then insert the zeros
3) use delays to offset the outputs of the three mini-filters so their samples line up properly. These "delays" are not actually storage locations or registers in our C/C++ code, nor is the zero-insertion; these are conceptual instead. How are they done in the code that you optimized?

Classic Polyphase Representation of Interpolation (L=3)

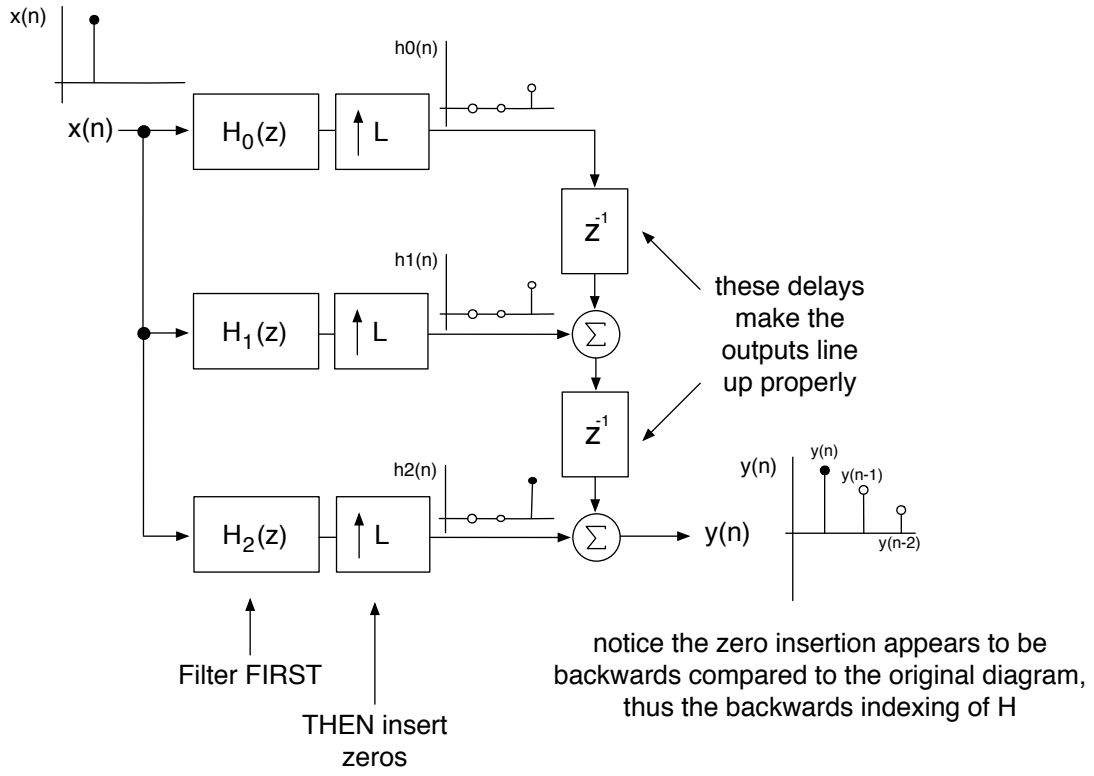


Figure 1.12: Classic representation of polyphase interpolation.

The same input is applied to each filter, just as we saw in the original sequence when we optimized it by skipping the zeros. The zeros are indeed inserted *after* the filtering operation but there are delays and summers in place that cause the output samples to line up properly; the first sample out is from H2,, followed by H1 and then H0's output two sample periods later. However, the delay elements are really implemented as iterations through our oversampling for-loop. The zeros aren't really inserted, they are ignored on each iteration too. From a programming point of view, we operate a for-loop that creates each output sample. An easier way to visualize the same thing is:

Alternate Polyphase Representation of Interpolation (L=3)

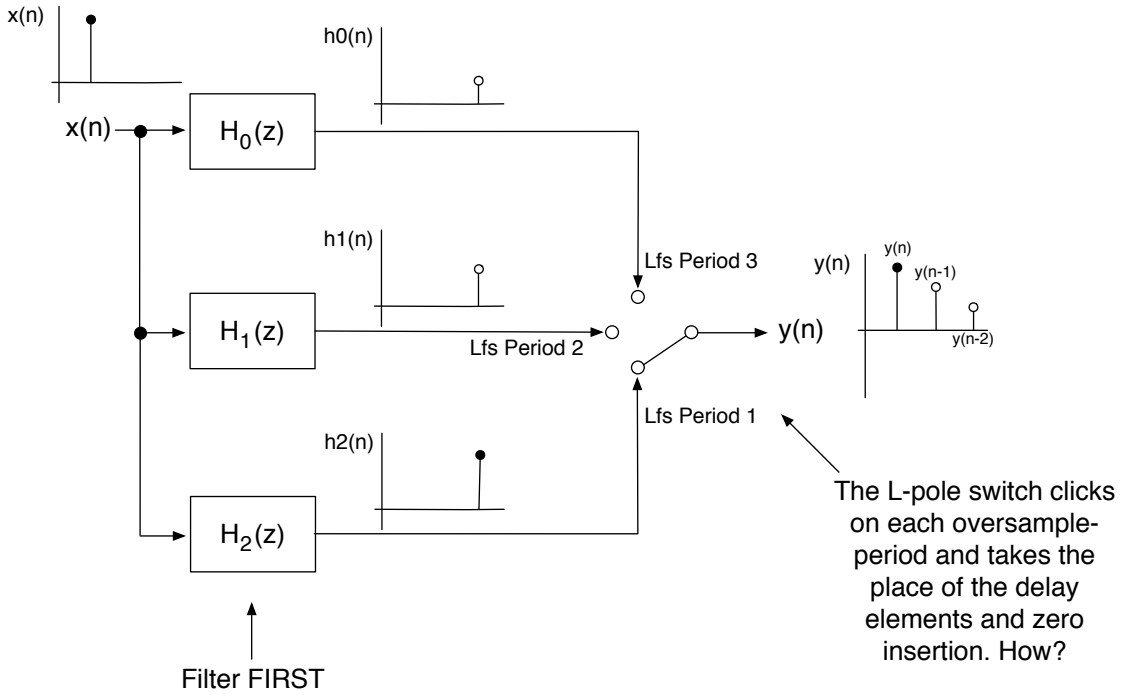


Figure 1.13: Alternate representation of polyphase interpolation.

In this case the L-Pole switch creates both the delays and the zero insertion at the same time. It represents each iteration (or “click”) through our for-loop. Mathematically, this figure is equivalent to the one above it.

Decimation Operation - the details

For Decimation or down-sampling, we also start with the LPF, shown as a combination of an input delay-line buffer and an impulse response (IR) buffer. The convolution produces the sum-of-products combination of the two buffers. We can label this filter simply H(z) in the frequency domain and represent it as shown at the bottom. This is identical to the Interpolation case.

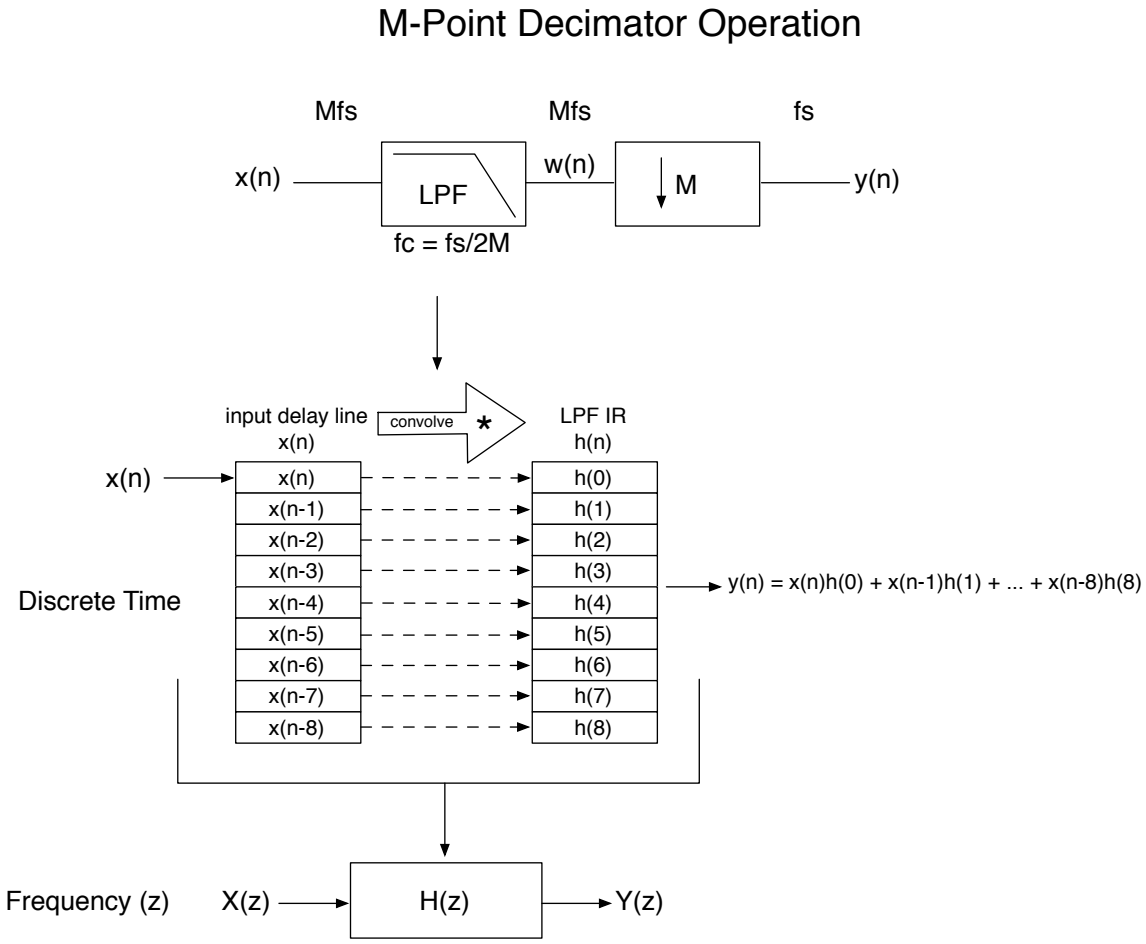


Figure 1.14: M-Point Decimator details

The flowchart is shown here (again, we'll use a value of 3 for M so we down-sample by M here). For every three points that go in, only one comes out.

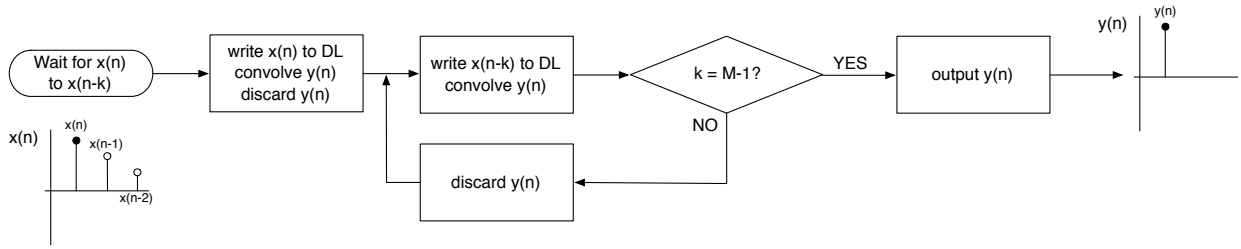
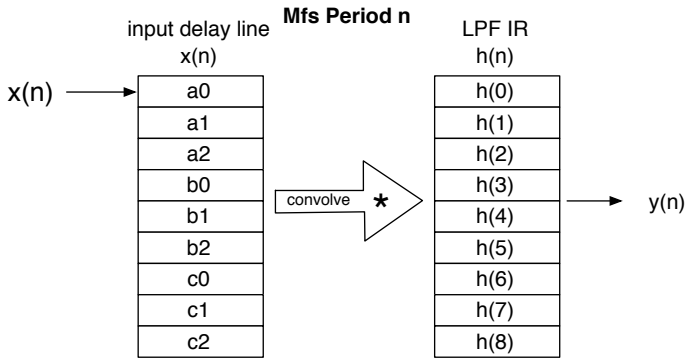


Figure 1.15: M-Point Decimator flowchart

You can see this is horribly inefficient since we convolve on each iteration of the loop, even though we discard M-1 of these convolutions. We can certainly fix that by only convolving once after M samples have been acquired. But can we decompose the filter the same way as in the interpolation case? The answer is sure we can and we get another polyphase decomposition/filtering algorithm. On the down-sampling side, the key is to identify that we are doing at least one full convolution and break that one apart. The result is that we will decimate before we filter, running our polyphase filters at the original (slower) sample rate.

For these examples, I am going to load up the input delay line a bit differently showing sets of L (3) samples labeled a,b,c…Each set of three samples will get converted into a single output.



y(n) = a0h(0) + a1h(1) + a2h(2) + b0h(3) + b1h(4) + b2h(5) + c0h(6) + c1h(7) + c2h(8)

Now decimate the input sequence (take every Mth input) and the filter (take every Mth coefficient)

y(n) = a0h(0) + b0h(3) + c0h(6)
+ a1h(1) + b1h(4) + c1h(7)
+ a2h(2) + b2h(5) + c2h(8)

or:

y(n) = y0(n) + y1(n) + y2(n)

where:
y0(n) = a0h(0) + b0h(3) + c0h(6)
y1(n) = a1h(1) + b1h(4) + c1h(7)
y2(n) = a2h(0) + b2h(5) + c2h(8)

Figure 1.16: The convolution can be broken into pieces by decimating both the input and filter coefficients

Look at the final difference equation for the filter and notice that we can re-arrange it just by moving the terms around and breaking the equation into 3 mini-outputs y0, y1 and y2. On the first Mfs sample period the three input samples are applied to the input delay line and then the convolution occurs. But, we fragment the convolution into three parts, using what is now a familiar method of skipping every Mth term in the IR

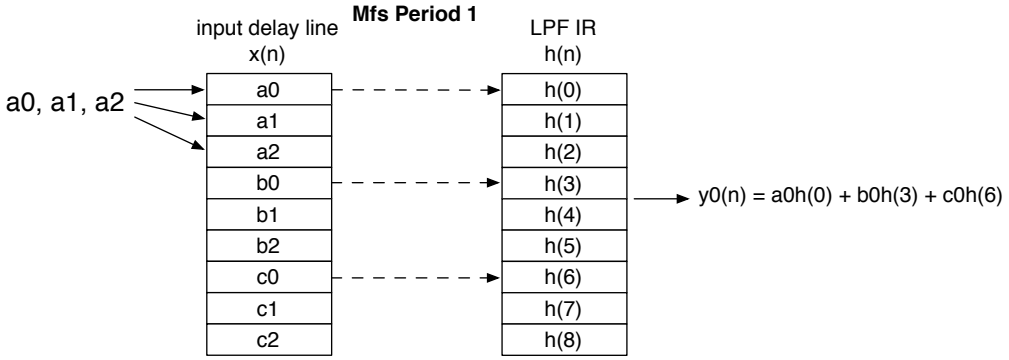
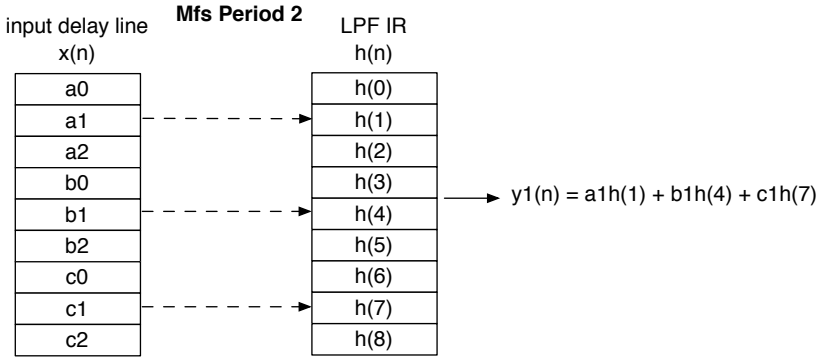


Figure 1.17: This forms the first mini-term for the output. The second two outputs are formed by shifting to the next mini-filter coefficient set:



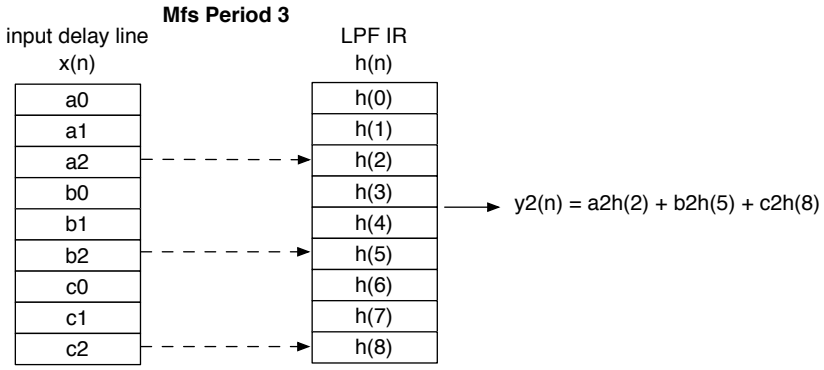


Figure 1.18: Does this look familiar when compared to the Interpolation? Kind-of. **Unlike the Interpolation version, the inputs to each coefficient group are not the same.**

Polyphase Decomposition

You can conceptually crack the original FIR filter into pieces, each containing every M-th (3rd) coefficient:

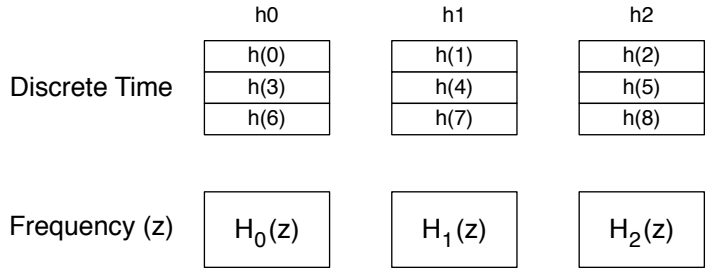


Figure 1.19: breaking the original filter into polyphase sub-band filters

This produces M (3) mini-FIR filters each with an impulse response h and a Frequency Transfer Function, H(z). In this case the indexing looks “normal” - again it is of no real consequence to our software operation. Now that we have three sub-band polyphase filters, we need to think about how we decimate the input before filtering. If you look at the figures above, you see that the inputs to each mini-filter are unique; in fact the inputs are decimated-by-3 versions of the original bitstream. The first is {a0, b0, c0} while the second is {a1, b1, c1} etc… Indeed the input as been decimated by 3 for each filter. And, each filter’s coefficients are identical to the Interpolation case as far as their indexes go.

Classic Polyphase Representation of Decimation (M=3)

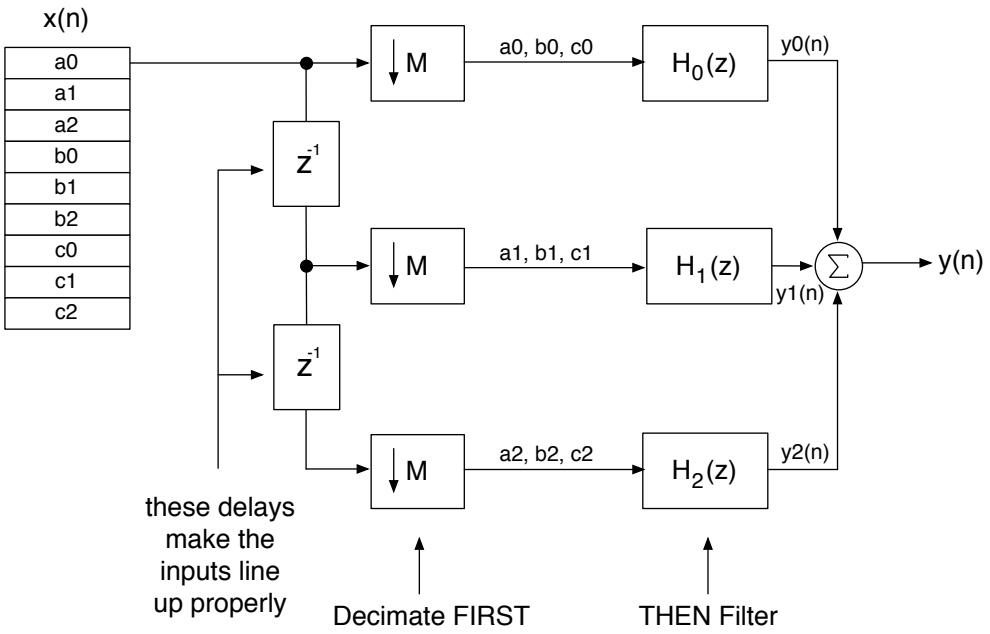


Figure 1.20: Classic representation of polyphase decimation

For the decimator, we also use conceptual delays to shift the signal before decimation. The top row has no delay, so it takes every Mth input value starting with a0. The second row has the input delayed by one sample period so it decimates M times starting with a1. The three branches produce the three mini-outputs y0, y1 and y2 then they are summed for the final output value. As with Interpolation, an alternate version of this block diagram is a bit easier to read.

Alternate Polyphase Representation of Decimation (M=3)

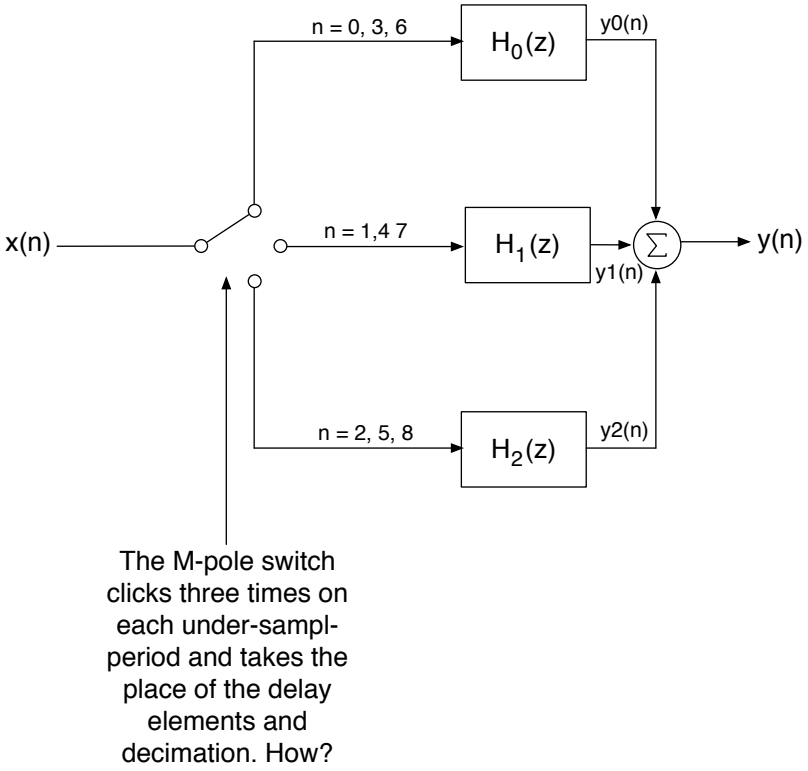


Figure 1.21: alternate representation of decimation with polyphase filters

Therefore, on one (normal) sample period, the switch clicks three times, applying x(0) to H0,, x(1) to H1 and x(2) to H2.

Polyphase Decomposition Math - some details

The polyphase math description has two parts:

1. Describing the decomposition of the original filter H(z) where we crack it into sub-filters
2. Describing the *interpolation and decimation processes* through these sub-filters

Type-1 Decomposition

Its easy to get confused, but the scope of this document is really the first item. Decomposing the polyphase filter is fairly easy to prove to yourself. The equation for the ***type-1 polyphase decomposition*** is:

$$H(z) = \sum_{p=0}^{L-1} H_p(z^L) z^{-p}$$

The original H(z) becomes a set of $H_p(z)$ filters. Each of these $H_p(z)$ filters is:

$$H_p(z) = \sum_{m=0}^{(K+1)/L} h[p + mL] z^{-m}$$

K is the order of the original filter

This produces the "*skipping through the impulse response by L*" that we saw visually above. For example if L = 3:

$$\begin{aligned}
H_0 &= h[0 + 0(3)]z^0 + h[0 + 1(3)]z^{-1} + h[0 + 2(3)]z^{-2} + ... \\
&= h[0] + h[3]z^{-1} + h[6]z^{-2} + ... \\
H_1 &= h[1 + 0(3)]z^0 + h[1 + 1(3)]z^{-1} + h[1 + 2(3)]z^{-2} + ... \\
&= h[1] + h[4]z^{-1} + h[7]z^{-2} + ... \\
H_2 &= h[2 + 0(3)]z^0 + h[2 + 1(3)]z^{-1} + h[2 + 2(3)]z^{-2} + ... \\
&= h[2] + h[5]z^{-1} + h[8]z^{-2} + ...
\end{aligned}$$

The consequence is that each filter's order becomes reduced by a factor of p. For example, suppose L = 3 and the original H(z) is an 8th order filter (a 9-tap FIR):

$$H(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3} + 5z^{-4} + 6z^{-5} + 7z^{-6} + 8z^{-7} + 9z^{-8}$$

The 3 sub-filters would then be:

$$\begin{aligned}
H_0(z) &= 1 + 4z^{-1} + 7z^{-2} \\
H_1(z) &= 2 + 5z^{-1} + 8z^{-2} \\
H_2(z) &= 3 + 6z^{-1} + 9z^{-2}
\end{aligned}$$

You can see that the original 8th order filter has been decomposed into three 2nd order filters. To get these transfer functions to combine back into the 8th order version is done by converting the filters from H(z) to H(z^L) then multiplying by z^{-p} like this:

$$H(z) = H_0(z^3)z^0 + H_1(z^3)z^{-1} + H_2(z^3)z^{-2}$$

Converting the filters from H(z) to H(z^L) is done by replacing z^q with z^{qL} like this:

$$H_0(z) = 1 + 4z^{-1} + 7z^{-2}$$
$$H_0(z^3) = 1 + 4z^{-3} + 7z^{-6}$$

$$H_1(z) = 2 + 5z^{-1} + 8z^{-2}$$
$$H_1(z^3) = 2 + 5z^{-3} + 8z^{-6}$$

$$H_2(z) = 3 + 6z^{-1} + 9z^{-2}$$
$$H_2(z^3) = 3 + 6z^{-3} + 9z^{-6}$$

Finally, multiply each filter by z^{-p} and reorder the terms in the proper sequence and get the final result:

$$\begin{aligned} H(z) &= H_0(z^3)z^0 + H_1(z^3)z^{-1} + H_2(z^3)z^{-2} \\ &= (1 + 4z^{-3} + 7z^{-6})z^0 + (2 + 5z^{-3} + 8z^{-6})z^{-1} + (3 + 6z^{-3} + 9z^{-6})z^{-2} \\ &= 1 + 4z^{-3} + 7z^{-6} + 2z^{-1} + 5z^{-4} + 8z^{-7} + 3z^{-2} + 6z^{-5} + 9z^{-8} \\ &= 1 + 2z^{-1} + 3z^{-2} + 4z^{-3} + 5z^{-4} + 6z^{-5} + 7z^{-6} + 8z^{-7} + 9z^{-8} \end{aligned}$$

Type-2 Decomposition

The polyphase decomposition equation can also be shown in a different form where the indexes run backwards; it is called the ***type-2 polyphase decomposition***:

$$H(z) = \sum_{p=0}^{L-1} H_p(z^L) z^{-(L-1-p)}$$

Now, each filter is defined by:

$$H_p(z) = \sum_{m=0}^{(K+1)/L} h[(L-1-p)+mL]z^{-m}$$

K is the order of the original filter

For example (L = 3):

$$\begin{aligned}
H_0 &= h[3-1-0+0(3)]z^0 + h[3-1-0+1(3)]z^{-1} + h[3-1-0+2(3)]z^{-2} + ... \\
&= h[2] + h[5]z^{-1} + h[8]z^{-2} + ... \\
H_1 &= h[3-1-1+0(3)]z^0 + h[3-1-1+1(3)]z^{-1} + h[3-1-1+2(3)]z^{-2} + ... \\
&= h[1] + h[4]z^{-1} + h[7]z^{-2} + ... \\
H_2 &= h[0+0(3)]z^0 + h[0+1(3)]z^{-1} + h[0+2(3)]z^{-2} + ... \\
&= h[0] + h[3]z^{-1} + h[6]z^{-2} + ...
\end{aligned}$$

The combination is now reversed:

$$H(z) = H_0(z^3)z^{-2} + H_1(z^3)z^{-1} + H_2(z^3)z^0$$

Using the original example above:

$$H(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3} + 5z^{-4} + 6z^{-5} + 7z^{-6} + 8z^{-7} + 9z^{-8}$$

The 3 sub-filters would then be:

$$\begin{aligned}
H_2(z) &= 1 + 4z^{-1} + 7z^{-2} \\
H_1(z) &= 2 + 5z^{-1} + 8z^{-2} \\
H_0(z) &= 3 + 6z^{-1} + 9z^{-2}
\end{aligned}$$

Notice that only the indexing of the filters has changed, compared to the first case. Finally:

$$\begin{aligned}
H(z) &= H_0(z^3)z^{-2} + H_1(z^3)z^{-1} + H_2(z^3)z^0 \\
&= (3 + 6z^{-3} + 9z^{-6})z^{-2} + (2 + 5z^{-3} + 8z^{-6})z^{-1} + (1 + 4z^{-3} + 7z^{-6})z^0
\end{aligned}$$

Which produces the same result.

The *mathematical descriptions* of the Interpolator and Decimator use the two different forms of decomposition in their proofs. The Decimator uses the type-1 decomposition while the Interpolator uses the type-2 decomposition. This is why the indexing on the H(z) filters looked backwards for the Interpolator but forwards for the decimator.

Why is it called "polyphase?"

The type-1 decomposition equation shows that each mini-filter is:

$$H_p(z) = \sum_{m=0}^{(K+1)/L} h[p + mL]z^{-m}$$

The original filter H(z) is decomposed into (K+1)/L mini-filters. The impulse response of each of those filters is a set of **h[n+L]** terms.

$$h[p + mL]$$

or more generally

$$h[n + L]$$

The 'p' value is a counter that produces the sequences and the L value skips through the original impulse response. So for example when p = 0 it produces an impulse response h = {h(0), h(3), h(6) etc…} and when p = 1 we get the next set h = {h(1), h(4), h(7)…}.

Manipulating the impulse response so that is shifted by L producing the **h[n+L]** version is called "time-shifting by L" in the Time Domain. In the Frequency Domain, this produces a Phase Shift.

$$h[n + L] \leftrightarrow e^{j\theta L}H(\theta)$$

$e^{j\theta L}$ represents a phase-shift of θL

The term "polyphase" comes from this phase shift component (the e term). Each sub-band filter is a phase shifted sub-component of the original impulse response. The interpolator and decimator filter branches use delays to line up the shifted/filtered components properly.

Designing an Oversampling Plug-In

We use oversampling plug-ins for several applications most notably for non-linear processing and waveform generation where we run into the problem that we can process or synthesize signals such that they contain aliasing components. Once the aliased components are present in the signal they can not be removed. In oversampling plug-ins the idea is to first up-sample, then perform the processing or synthesis at the new rate, then down-sample. Aliasing may still occur however the components that make it back into the original spectrum may be negligible.

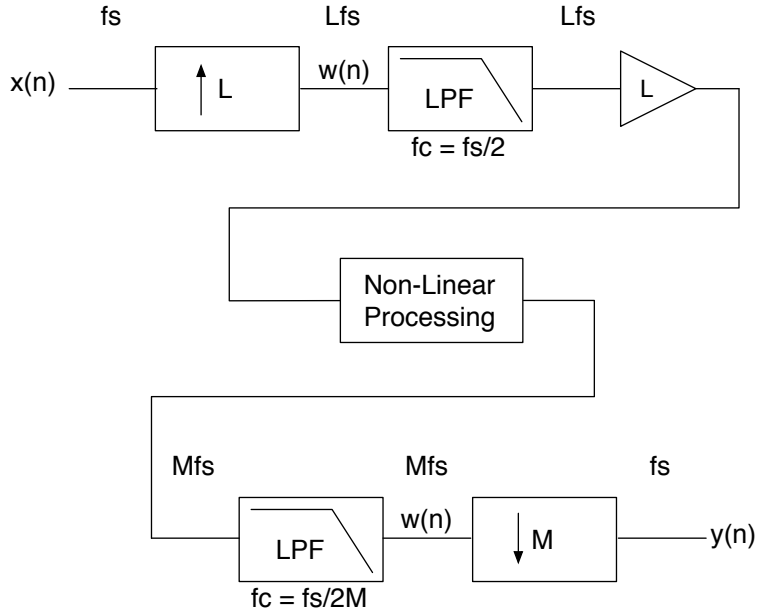


Figure 1.22: A non-linear processing plug-in with oversampling

We want to up and down-sample at the same rate so that L = M which means that the Low Pass Filters for both operations will be identical with a cutoff (theoretically) at fs/2 or Nyquist. Implementing the oversampling plug-in requires:

- designing this LPF as perfectly as possible knowing that it runs at the oversampled rate
- implementing the interpolator block which inserts L-1 zeros to perform L convolutions with the LPF
- implementing the decimator block which throws out L-1 convolutions with the LPF

The block flowcharts for the Interpolator and Decimator can be translated into C++ fairly easily. We observe that both the Interpolator and Decimator share common traits:

- they both convolve with a FIR Low Pass Filter
- their orders (L and M) are identical

So, they are both sort of like specialized convolvers. In order to implement these in C++ I created a base class that has the variables that are common to both. The base class is called CRateConverter.

```
class CRateConvertor
{
public:
	CRateConvertor(void);
	virtual ~CRateConvertor(void);

	// buffer for the impulse response h[n] of the FIR
	float *m_pIRBuffer;

	// buffers for the transversal delay lines, one for each input
	float* m_pLeftInputBuffer;
	float* m_pRightInputBuffer;

	// read index for delay lines (input x buffers)
	int m_nReadIndexDL;

	// read index for impulse response buffers
	int m_nReadIndexH;

	// write index for input x buffer
	int m_nWriteIndex;

	int m_nL; // OS value, 4 = 4X Oversampling

	// counters and index values for the convolutions
	int m_nCurrentL;
	int m_nIRLength;
	int m_nOSCounter;

	// initializer - creates the buffers and loads the FIR IR
	void init(int L, int FIRLen, float *pIRBuffer);

	// flush buffers
	void reset();

	// overrides for derived objects
	//
	// interpolateSamples: take one pair of L/R samples and produce L-length buffers of samples
	virtual void interpolateSamples(float xnL, float xnR, float* pLeftInterpBuffer, float* pRightInterpBuffer);

	// inner loop function that processes just one pair of inputs
	virtual void interpolateNextOutputSample(float xnL, float xnR, float& fLeftOutput, float& fRightOutput);

	// decimateSamples: take one pai rL-length buffers of samples and decimate down to just one pair of output samples
	virtual void decimateSamples(float* pLeftDeciBuffer, float* pRightDeciBuffer, float& ynL, float& ynR);

	// inner loop function that processes just one pair of inputs
	virtual bool decimateNextOutputSample(float xnL, float xnR, float& fLeftOutput, float& fRightOutput);

};
```

The Interpolator and Decimator are derived classes that only override the appropriate functions:

```
class CInterpolator : public CRateConvertor
{
public:
	CInterpolator(void);
	~CInterpolator(void);

	virtual void interpolateNextOutputSample(float xnL, float xnR, float& fLeftOutput, float& fRightOutput);
	virtual void interpolateSamples(float xnL, float xnR, float* pLeftInterpBuffer, float* pRightInterpBuffer);

};
```

```
class CDecimator : public CRateConvertor
{
public:
	CDecimator(void);
	~CDecimator(void);

	virtual bool decimateNextOutputSample(float xnL, float xnR, float& fLeftOutput, float& fRightOutput);
	virtual void decimateSamples(float* pLeftDeciBuffer, float* pRightDeciBuffer,float& ynL, float& ynR);
};
```

Using Interpolator and Decimator requires initializing them with a FIR Impulse Response and oversampling ratio then calling the appropriate function.

Our processAudioFrame() will:
- up-sample each input sample into a buffer of L interpolated samples
- go into a for-loop and process the buffers through the algorithm, producing buffers of processed data
- down-sample the buffers of processed data back to a single L/R pair of outputs

The **OverSampWaveShaper** plug-in uses 4X oversampling and implements one stage of the WaveShaper from my book. It declares the following variables in the .h file:

```
// the ratio = 4 in our case
int m_nOversamplingRatio;

// our objects to do the work
CInterpolator m_Interpolator;
CDecimator m_Decimator;

// buffers to hold the oversampled output of the Interpolator
float* m_pLeftInterpBuffer;
float* m_pRightInterpBuffer;

// buffers to hold the oversampled input to the Decimator
float* m_pLeftDecipBuffer;
float* m_pRightDeciBuffer;
```

We will use the CPlugIn's built-in impulse response array to hold the IR for the Low Pass Filters. The filters will be identical right and left so we only need one of the arrays, I chose the left. We will design the FIR Low Pass filter after discussing the operation.

```
// impulse response buffers (plugin.h file)
float m_h_Left[1024];
float m_h_Right[1024]; // don't need
```

The code details can be found in the downloadable project. The highlights and code snippets are as follows:

Constructor:
- initialize the FIR Array using the copy-to-clipboard functionality in the FIR designer
- initialize the Interpolator and Decimator objects
- create and flush our buffers

```
	// now init the units
	m_Interpolator.init(m_nOversamplingRatio, m_nIRLength, &m_h_Left[0]);
	m_Decimator.init(m_nOversamplingRatio, m_nIRLength, &m_h_Left[0]);
```

```
// dynamically allocate the input x buffers and save the pointers
m_pLeftInterpBuffer = new float[m_nOversamplingRatio];
m_pRightInterpBuffer = new float[m_nOversamplingRatio];

// flush interp buffers
memset(m_pLeftInterpBuffer, 0, m_nOversamplingRatio*sizeof(float));
memset(m_pRightInterpBuffer, 0, m_nOversamplingRatio*sizeof(float));

// dynamically allocate the input x buffers and save the pointers
m_pLeftDecipBuffer = new float[m_nOversamplingRatio];
m_pRightDeciBuffer = new float[m_nOversamplingRatio];

// flush deci buffers
memset(m_pLeftDecipBuffer, 0, m_nOversamplingRatio*sizeof(float));
memset(m_pRightDeciBuffer, 0, m_nOversamplingRatio*sizeof(float));
```

Destructor:
- delete the stuff we allocated in the constructor

```
if(m_pLeftInterpBuffer) delete [] m_pLeftInterpBuffer;
if(m_pRightInterpBuffer) delete [] m_pRightInterpBuffer;
if(m_pLeftDecipBuffer) delete [] m_pLeftDecipBuffer;
if(m_pRightDeciBuffer) delete [] m_pRightDeciBuffer;
```

prepareForPlay():
- reset the Interpolator and Decimator objects

```
m_Interpolator.reset();
m_Decimator.reset();
```

processAudioFrame():
- interpolate the inputs from L and R into buffers; each buffer has 4 interpolated samples in it

```
m_Interpolator.interpolateSamples(xnL, xnR, m_pLeftInterpBuffer, m_pRightInterpBuffer);
```

- loop through the buffers processing each sample; this is the high speed part
- the output of each iteration through the loop produces a sample to load into the decimation buffer

```
for(int i=0; i<m_nOversamplingRatio; i++)
{
    // interp next sample
    interpL = m_pLeftInterpBuffer[i];
    interpR = m_pRightInterpBuffer[i];

    // DO THE PROCESSING - the results are in processedL and processedR

    // load decimation buffers with processed audio
    m_pLeftDecipBuffer[i] = processedL;
    m_pRightDeciBuffer[i] = processedR;
}
```

- decimate the buffers to produce a single output L/R pair

```
// decimate them
m_Decimator.decimateSamples(m_pLeftDecipBuffer, m_pRightDeciBuffer, ynL, ynR);
```

- ynL and ynR are then delivered to the RackAFX output buffers

Designing the LPF

Use RackAFX's FIR Designer Tool to create the plug-in. We will use the Optimal Method to create the filter (this is one of several popular FIR filters for Rate Conversion and is not the only option, but it's easy for use because it's built-in). We will specify the filter as follows:

- Passband Frequency: 20kHz
- Stopband Frequency: 22kHz
- fs: 176.4kHz (4 x 44.1kHz)
- Passband Ripple: 0.1dB
- Stopband Attenuation: 96dB

Open the controls to set the proper variables. Use the slider to find the first filter that meets our criteria.

We also want the FIR to be a multiple of our oversampling ratio so you can turn it into a Polyphase Oversampling Plug-In as your homework.

I found that a filter of order 300 will barely get us by. Here's a screenshot of the filter designer:

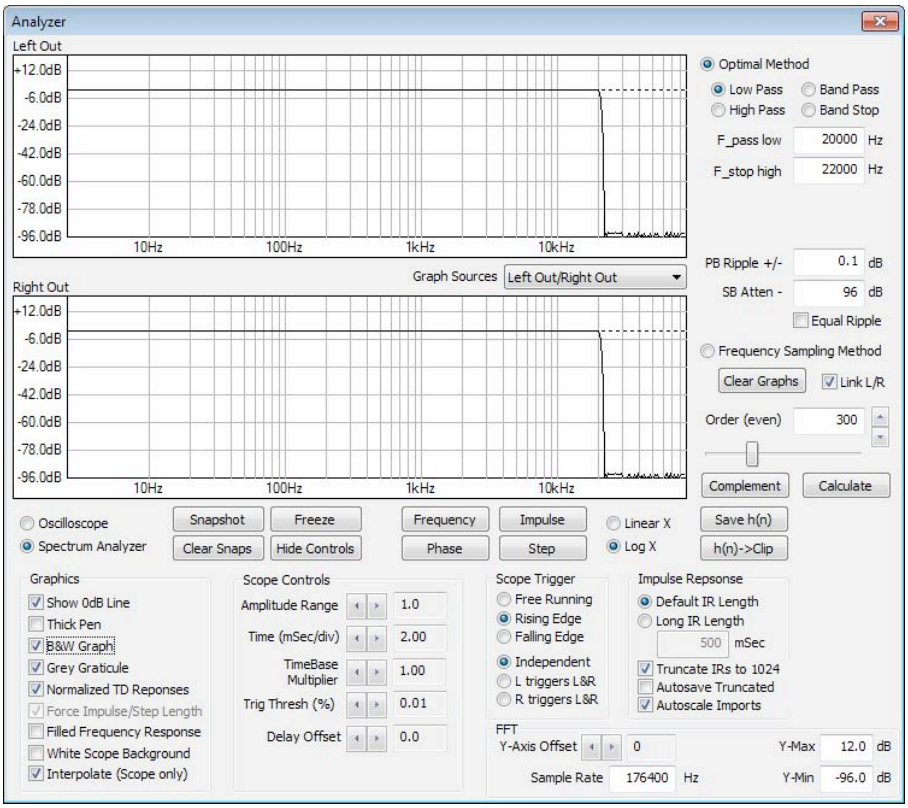


Figure 1.23: The FIR Filter designer in RackAFX is used for the LPF design

You can see this is a wicked filter! It's very steep - note that with the new Sample Rate (you enter it at the bottom) the graph stretches out so that Nyquist is 88.2kHz and you can also just barely see the stopband attenuation humps just above -96dB. The filter has 300 taps which is a multiple of our oversampled rate of 4.

Use the h(n) -> Clip button to copy the IR into the clipboard, then paste it into your constructor. You can delete the right channel IR array plus the zero-valued array items. It will look like this:

(In the constructor):

```
// Finish initializations here
m_nOversamplingRatio = 4;

// Optimal Method Filters designed at 176400Hz; fpass = 20kHz, fstop = 22kHz
// pb ripple = 0.1 dB
// sd  < -96dB
// adjusted till just met specs AND length was a multiple of OS Ratio (4) for polyphase later
// used RackAFX FIR Designer
// the Left IR array is used for both filters

// 4X Coeffs
m_nIRLength = 300;

m_h_Left[0] = -0.0000064687028498;
m_h_Left[1] = 0.0000422707853431;
m_h_Left[2] = 0.0001517860073363;
m_h_Left[3] = 0.0003613336593844;
m_h_Left[4] = 0.0006766511942260;
m_h_Left[5] = 0.0010650000767782;
m_h_Left[6] = 0.0014487076550722;
m_h_Left[7] = 0.0017182164592668;
```

< SNIP SNIP SNIP >

```
m_h_Left[289] = 0.0010077664628625;
m_h_Left[290] = 0.0015226191608235;
m_h_Left[291] = 0.0017646064516157;
m_h_Left[292] = 0.0017182164592668;
m_h_Left[293] = 0.0014487076550722;
m_h_Left[294] = 0.0010650000767782;
m_h_Left[295] = 0.0006766511942260;
m_h_Left[296] = 0.0003613336593844;
m_h_Left[297] = 0.0001517860073363;
m_h_Left[298] = 0.0000422707853431;
m_h_Left[299] = -0.0000064687028498;

// now init the units
m_Interpolator.init(m_nOversamplingRatio, m_nIRLength, &m_h_Left[0]);
m_Decimator.init(m_nOversamplingRatio, m_nIRLength, &m_h_Left[0]);

// dynamically allocate the input x buffers and save the pointers
m_pLeftInterpBuffer = new float[m_nOversamplingRatio];
m_pRightInterpBuffer = new float[m_nOversamplingRatio];
```

```
// flush interp buffers
memset(m_pLeftInterpBuffer, 0, m_nOversamplingRatio*sizeof(float));
memset(m_pRightInterpBuffer, 0, m_nOversamplingRatio*sizeof(float));

// dynamically allocate the input x buffers and save the pointers
m_pLeftDecipBuffer = new float[m_nOversamplingRatio];
m_pRightDeciBuffer = new float[m_nOversamplingRatio];

// flush deci buffers
memset(m_pLeftDecipBuffer, 0, m_nOversamplingRatio*sizeof(float));
memset(m_pRightDeciBuffer, 0, m_nOversamplingRatio*sizeof(float));
```

Build and test your plug-in. Use the oscillator to verify that the oversampling is indeed preventing aliasing (see my video if you need to).

Now, you can begin the process of converting it to a Polyphase design. It's easier to start by optimizing my code as follows:

Interpolator:
- the interpolator is inefficient because the convolutions have lots of zero-valued inputs. Fix this by altering the loop (don't forge the loop and array indexing!!) so that the convolutions only happen with non-zero valued input values

Decimator:
- the decimator is inefficient because it convolves even when it is going to throw the result away! Fix this in the convolution setup so that only the last convolution is actually performed

When you do these optimizations, you will be performing the polyphase decomposition almost directly for the Interpolator. The difference is that instead of decomposing and declaring mini-filters you will be keeping 4 interleaved filters in the original Impulse Response Array.

On the decimator side your fix will speed things up but the convolution will still be a single one. You would need to split this into 4 smaller convolutions each with its own IR array to really be doing Polyphase to-the-letter.

Challenge:
Create a derivative plug-in project (File -> Save Project As) and implement true polyphase filtering; you will need to alter the init() method on the CRateConverter object so that it creates sets of input buffers and polyphase sub-band filter IR arrays. You will need to alter the Interpolator and Decimator loops to perform the new convolutions. Use the figures above labeled "***Alternate Polyphase Representation of…***" to help you figure this out.

References:

Ifeachor, Emmanuel C. and Jervis, Barrie W. 1993. *Digital Signal Processing, A Practical Approach*, Chap. 4,6. Menlo Park: Addison Wesley.

Pirkle, Will. 2012. *Designing Audio Effect Plug-Ins in C++*, Burlington: Focal Press.

Porat, Boaz. 1997. *A Course in Digital Signal Processing*, New York: John Wiley and Sons.

http://www.rle.mit.edu/dspg/documents/main.pdf
https://ccrma.stanford.edu/~jos/sasp/N_Channel_Polyphase_Decomposition.html

http://ws2.binghamton.edu/fowler//fowler%20personal%20page/EE521_files/IV-05%20Polyphase%20FIlters%20Revised.pdf