Digital Filters with MATLAB®*

Ricardo A. Losada†
The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

December 15, 2009

* MATLAB® and Simulink® are registered trademarks of The MathWorks, Inc.
† Ricardo Losada is with the Signal Processing and Communications development team at The MathWorks, Inc.

Contents

Preface

This document* constitutes a tutorial on design and implementation of digital filters in MATLAB. The tutorial is based on functionality from the Filter Design Toolbox™.

The document covers the design of FIR and IIR single-rate and multirate filters. It also discusses advanced design techniques such as multirate/multistage decimation/interpolation and the use of special multirate filters such as allpass-based polyphase IIR filters, CIC filters, and Farrow filters.

The tutorial focuses on practical aspects of filter design and implementation, and on the advantages and disadvantages of the different design algorithms. The theory behind the design algorithms is kept to a minimal.

* This document may be updated from time to time. The latest version can be found at www.mathworks.com/matlabcentral

Part I

Filter Design

Chapter 1

Basic FIR Filter Design

Overview

In this chapter we discuss the basic principles of FIR filter design. We concentrate mostly on lowpass filters, but most of the results apply to other response types as well. We discuss the basic trade offs and the degrees of freedom available for FIR filter design. We motivate the use of optimal designs and introduce both optimal equiripple and optimal least-squares designs. We then discuss optimal minimum-phase designs as a way of surpassing in some sense comparable optimal linear-phase designs. We introduce sloped equiripple designs as a compromise to obtain equiripple passband yet non-equiripple stopband. We also mention a few caveats with equiripple filter design. We end with an introductory discussion of different filter structures that can be used to implement an FIR filter in hardware.

The material presented here is based on [1] and [2]. However, it has been expanded and includes newer syntax and features from the Filter Design Toolbox.

1.1 Why FIR filters?

There are many reasons why FIR filters are very attractive for digital filter design. Some of them are:

- Simple robust way of obtaining digital filters

- Inherently stable when implemented non recursively
- Free of limit cycles when implemented non recursively
- Easy to attain linear phase
- Simple extensions to multirate and adaptive filters
- Relatively straight-forward to obtain designs to match custom magnitude responses
- Some vendors and specialized hardware only support FIR
- Low sensitivity to quantization effects compared to many IIR filters

FIR filters have some drawbacks however. The most important is that they can be computationally expensive to implement. Another is that they have a long transient response. It is commonly thought that IIR filters must be used when computational power is at a premium. This is certainly true in some cases. However, in many cases, the use of multistage/multirate techniques can yield FIR implementations that can compete (and even surpass) IIR implementations while retaining the nice characteristics of FIR filters such as linear-phase, stability, and robustness to quantization effects.* However, these efficient multistage/multirate designs tend to have very large transient responses, so depending on the requirements of the filter, IIR designs may still be the way to go.

In terms of the long transient response, we will show in Chapter 2 that minimum-phase FIR filters can have a shorter transient response than comparable IIR filters.

1.2 Lowpass filters

The ideal lowpass filter is one that allows through all frequency components of a signal below a designated cutoff frequency ω_c, and rejects all frequency components of a signal above ω_c.

* That being said, there are also modern IIR design techniques that lend themselves to efficient multirate implementations and are extremely computationally efficient. We are referring here to more traditional IIR designs implemented using direct-form I or II structures, possibly in cascaded second-order section form.

Its frequency response satisfies

$$H_{\mathrm{LP}}(e^{j\omega}) = \begin{cases} 1, & 0 \le \omega \le \omega_c \\ 0, & \omega_c < \omega \le \pi \end{cases} \tag{1.1}$$

The impulse response of the ideal lowpass filter (1.1) can easily be found to be [3]

$$h_{\mathrm{LP}}[n] = \frac{\sin(\omega_c n)}{\pi n}, \quad -\infty < n < \infty. \tag{1.2}$$

1.2.1 FIR lowpass filters

Because the impulse response required to implement the ideal lowpass filter is infinitely long, it is impossible to design an ideal FIR lowpass filter.

Finite length approximations to the ideal impulse response lead to the presence of ripples in both the passband ($\omega < \omega_c$) and the stopband ($\omega > \omega_c$) of the filter, as well as to a nonzero transition width between the passband and stopband of the filter (see Figure 1.1).

1.2.2 FIR filter design specifications

Both the passband/stopband ripples and the transition width are undesirable but unavoidable deviations from the response of an ideal lowpass filter when approximating with a finite impulse response. Practical FIR designs typically consist of filters that meet certain design specifications, i.e., that have a transition width and maximum passband/stopband ripples that do not exceed allowable values.

In addition, one must select the filter order, or equivalently, the length of the truncated impulse response.

A useful metaphor for the design specifications in FIR design is to think of each specification as one of the angles in a triangle as in Figure 1.2 *.

The metaphor is used to understand the degrees of freedom available when designating design specifications. Because the sum of the angles is fixed, one can at most select the values of two of the specifications. The third specification will be determined by the design algorithm utilized.

* For the ripples we should more generally speak of some measure or norm of them. The peak ripple corresponding to the $\mathcal{L}_\infty$-norm is the most commonly used measure, but other norms are possible.

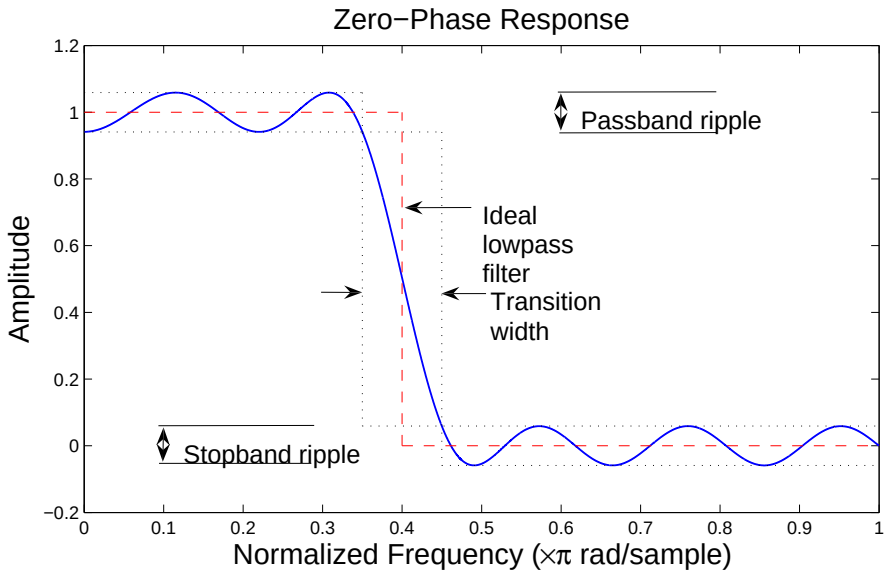


Figure 1.1: *Illustration of the typical deviations from the ideal lowpass filter when approximating with an FIR filter,* $\omega_c = 0.4\pi$.

Moreover, as with the angles in a triangle, if we make one of the specifications larger/smaller, it will impact one or both of the other specifications.

Example 1 *As an example, consider the design of an FIR filter that meets the following specifications:*

Specifications Set 1

1. *Cutoff frequency:* 0.4π *rad/sample*
2. *Transition width:* 0.06π *rad/sample*
3. *Maximum passband/stopband ripple: 0.05*

The filter can easily be designed with the truncated-and-windowed impulse response algorithm (a.k.a. the "window method") if we use a Kaiser window[*]*:*

[*] Notice that when specifying frequency values in MATLAB, the factor of π should be omitted.

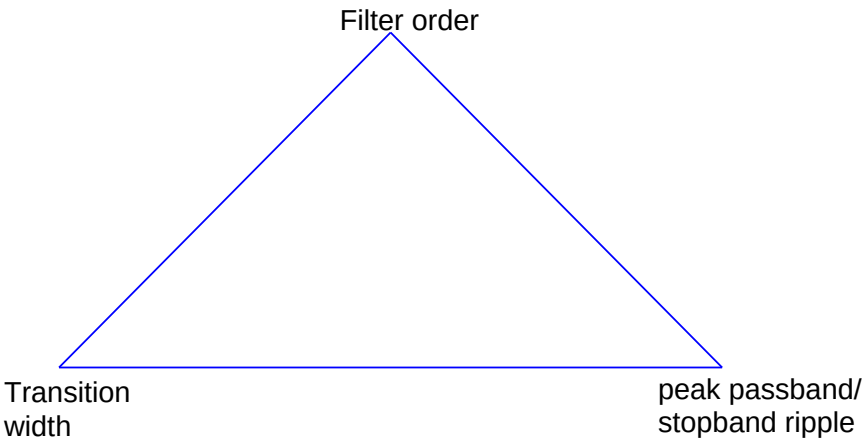


Figure 1.2: *FIR design specifications represented as a triangle.*

```
Fp = 0.4 - 0.06/2; Fst = 0.4 + 0.06/2;
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,0.05,0.05,'linear');
design(Hf,'kaiserwin');
```

The zero-phase response of the filter is shown in Figure 1.3. Note that since we have fixed the allowable transition width and peak ripples, the order is determined for us.

Close examination at the passband-edge frequency[*]*, $\omega_p = 0.37\pi$, and at the stopband-edge frequency, $\omega_s = 0.43\pi$, shows that the peak passband/stopband ripples are indeed within the allowable specifications. Usually the specifications are exceeded because the order is rounded to the next integer greater than the actual value required.*

[*] The passband-edge frequency is the boundary between the passband and the transition band. If the transition width is T_w, the passband-edge frequency ω_p is given in terms of the cutoff frequency ω_c by $\omega_p = \omega_c - T_w/2$. Similarly, the stopband-edge frequency is given by $\omega_s = \omega_c + T_w/2$.

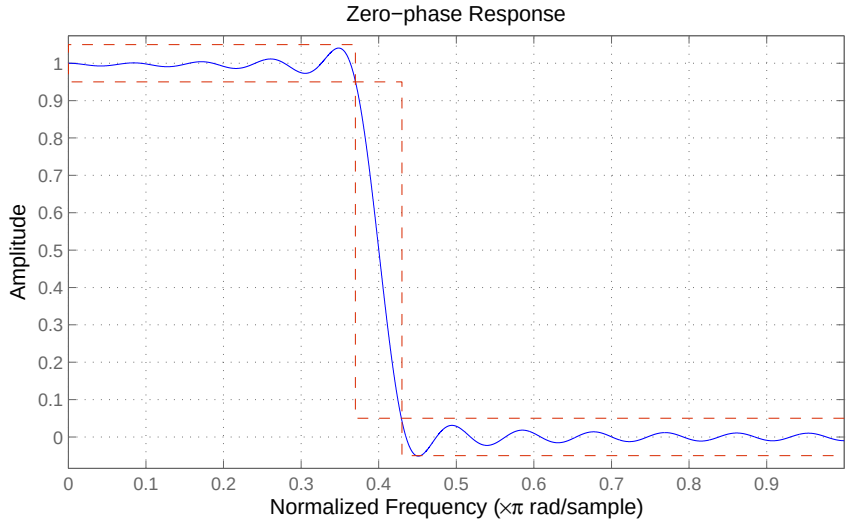


Figure 1.3: *Kaiser window design meeting prescribed specifications.*

1.2.3 Working with Hertz rather than normalized frequency

In many applications the specifications are given in terms of absolute frequency in Hertz rather than in terms of normalized frequency. Conversion between one and the other is straightforward. Recall that normalized frequency is related to absolute frequency by

$$\omega = \frac{2\pi f}{f_s}$$

where f is absolute frequency in cycles/second, f_s is the sampling frequency in samples/second, and ω is normalized frequency in radians/sample.

Suppose the specifications for a filter design problem include a passband frequency of 250 Hz, a stopband frequency of 300 Hz, and a sampling frequency of 1 kHz. The corresponding normalized passband and stopband frequencies are 0.5π and 0.6π respectively.

However, it is not necessary to perform such conversion manually. It is possible to specify design parameters directly in Hertz.

For example, the following two filters `H1` and `H2` are identical.

```
Hf  = fdesign.lowpass('Fp,Fst,Ap,Ast',.5,.6,1,80);
```

```
H1  = design(Hf);
Hf2 = fdesign.lowpass('Fp,Fst,Ap,Ast',250,300,1,80,1000);
H2  = design(Hf2);
```

Notice that we don't add `'Fs'` to the string `'Fp,Fst,Ap,Ast'` (or any other specification string) when we specify parameters in Hertz. Simply appending the sampling frequency to the other design parameters indicates that all frequencies specified are given in Hertz.

1.3 Optimal FIR filter design

While the truncated-and-windowed impulse response design algorithm is very simple and reliable, it is not optimal in any sense. The designs it produces are generally inferior to those produced by algorithms that employ some optimization criteria in that it will have greater order, greater transition width or greater passband/stopband ripples. Any of these is typically undesirable in practice, therefore more sophisticated algorithms come in handy.

1.3.1 Optimal FIR designs with fixed transition width and filter order

Optimal designs are computed by minimizing some measure of the deviation between the filter to be designed and the ideal filter. The most common optimal FIR design algorithms are based on fixing the transition width and the order of the filter. The deviation from the ideal response is measured only by the passband/stopband ripples. This deviation or error can be expressed mathematically as [4]

$$E(\omega) = H_a(\omega) - H_{\mathrm{LP}}(e^{j\omega}), \quad \omega \in \Omega$$

where $H_a(\omega)$ is the zero-phase response of the designed filter and $\Omega = [0, \omega_p] \cup [\omega_s, \pi]$. It is still necessary to define a measure to determine "the size" of $E(\omega)$ - the quantity we want to minimize as a result of the optimization. The most often used measures are the $\mathcal{L}_\infty$-norm ($\|E(\omega)\|_\infty$ - minimax designs) and the $\mathcal{L}_2$-norm ($\|E(\omega)\|_2$ - least-squares designs).

In order to allow for different peak ripples in the passband and stopband, a weighting function, $W(\omega)$ is usually introduced,

$$E_W(\omega) = W(\omega)[H_a(\omega) - H_{\mathrm{LP}}(e^{j\omega})], \quad \omega \in \Omega$$

Linear-phase designs

A filter with linear-phase response is desirable in many applications, notably image processing and data transmission. One of the desirable characteristics of FIR filters is that they can be designed very easily to have linear phase. It is well known [5] that linear-phase FIR filters will have impulse responses that are either symmetric or antisymmetric. For these types of filters, the zero-phase response can be determined analytically [5], and the filter design problem becomes a well behaved mathematical approximation problem [6]: Determine the best approximation to a given function (the ideal lowpass filter's frequency response) by means of a polynomial (the FIR filter) of a given order. By "best" it is meant the one which minimizes the difference between them - $E_W(\omega)$ - according to a given measure.

The `equiripple` design implements an algorithm developed in [7] that computes a solution to the design problem for linear-phase FIR filters in the $\mathcal{L}_\infty$-norm case. The design problem is essentially to find a filter that minimizes the *maximum* error between the ideal and actual filters. This type of design leads to so-called equiripple filters, i.e. filters in which the peak deviations from the ideal response are all equal.

The `firls` design implements an algorithm to compute solution for linear-phase FIR filters in the $\mathcal{L}_2$-norm case. The design problem is to find a filter that minimizes the energy of the error between ideal and actual filters.

Equiripple filters

Linear-phase equiripple filters are desirable because they have the smallest maximum deviation from the ideal filter when compared to all other linear-phase FIR filters of the same order. Equiripple filters are ideally suited for applications in which a specific tolerance must be met. For example, if it is necessary to design a filter with a given minimum stopband attenuation or a given maximum passband ripple.

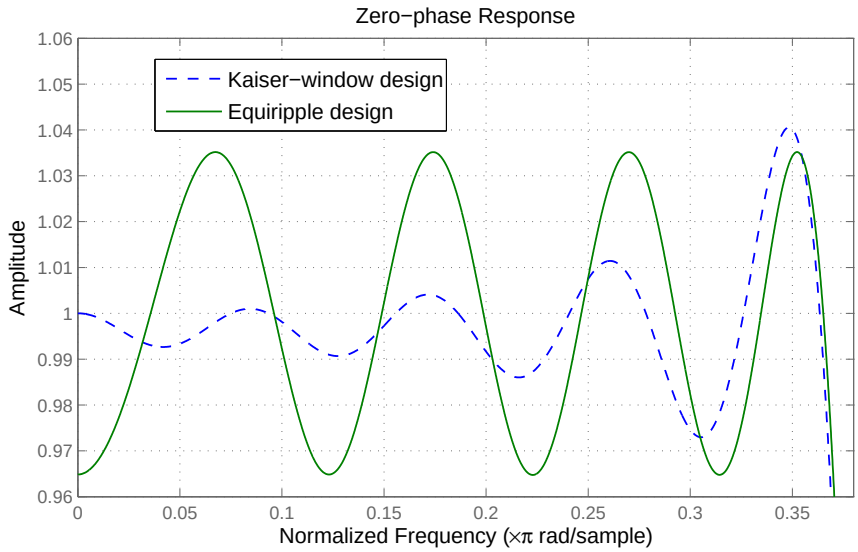


Figure 1.4: *Passband ripple for of both the Kaiser-window-designed FIR filter and the equiripple-designed FIR filter.*

Example 2 *The Kaiser-window design of Example 1 was of 42nd order. With this same order, an equiripple filter (with fixed transition width) can be designed that is superior to the Kaiser-window design:*

```
Fp  = 0.4 - 0.06/2; Fst = 0.4 + 0.06/2;
Hf  = fdesign.lowpass('N,Fp,Fst',42,Fp,Fst);
Heq = design(Hf,'equiripple');
```

Figure 1.4 shows the superposition of the passband details for the filters designed with the Kaiser window and with the `equiripple` *design. Clearly the maximum deviation is smaller for the* `equiripple` *design. In fact, since the filter is designed to minimize the maximum ripple (minimax design), we are guaranteed that no other linear-phase FIR filter of 42nd order will have a smaller peak ripple for the same transition width.*

We can measure the passband ripple and stopband attenuation in dB units using the `measure` *command,*

```
Meq = measure(Heq);
```

If we compare the measurements of the equiripple design to those of the Kaiser-window design, we can verify for instance that the equiripple design provides a minimum stopband attenuation of 29.0495 dB compared to 25.8084 dB for the Kaiser-window design.

Least-squares filters

Equiripple designs may not be desirable if we want to minimize the energy of the error (between ideal and actual filter) in the passband/stopband. Consequently, if we want to reduce the energy of a signal as much as possible in a certain frequency band, least-squares designs are preferable.

Example 3 *For the same specifications,* `Hf`*, as the equiripple design of Example 2, a least-squares FIR design can be computed from*

```
Hls = design(Hf,'firls');
```

The stopband energy for this case is given by

$$E_{sb} = \frac{2}{2\pi} \int_{0.43\pi}^{\pi} \left| H(e^{j\omega}) \right|^2 d\omega$$

where $H(e^{j\omega})$ is the frequency response of the filter.

In this case, the stopband energy for the equiripple filter is approximately 3.5214-004 while the stopband energy for the least-squares filter is 6.6213e-005. (As a reference, the stopband energy for the Kaiser-window design for this order and transition width is 1.2329e-004).

The stopband details for both equiripple design and the least-squares design are shown in Figure 1.5.

So while the equiripple design has less peak error, it has more "total" error, measured in terms of its energy. However, although the least-squares design minimizes the energy in the ripples in both the passband and stopband, the resulting peak passband ripple is always larger than that of a comparable equiripple design. Therefore there is a larger disturbance on the signal to be filtered for a portion of the frequencies that the filter should allow to pass (ideally undisturbed). This is a drawback of least-squares designs. We will see in Section 1.3.5 that a possible compromise is to design equiripple filters in such a way that the maximum ripple in the passband is minimized, but with a sloped stopband that can reduce the stopband energy in a manner comparable to a least-squares design.

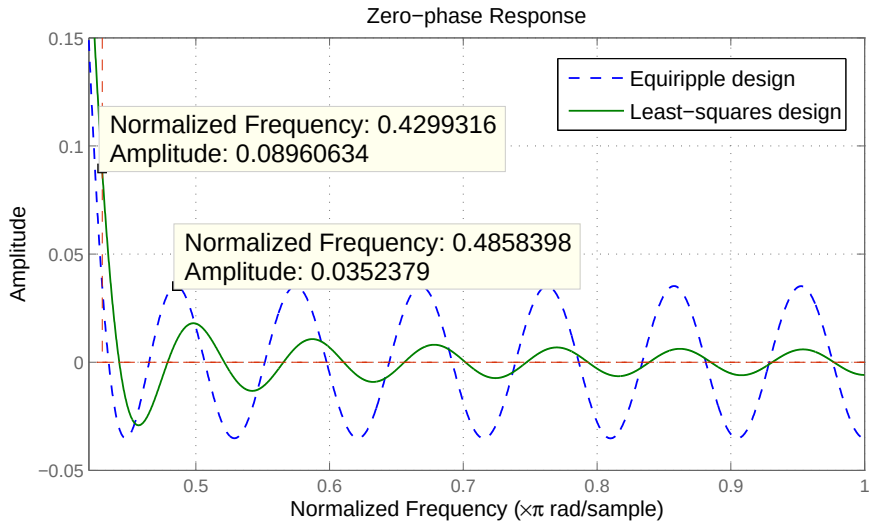


Figure 1.5: *Comparison of an optimal equiripple FIR design and an optimal least-squares FIR design. The equiripple filter has a smaller peak error, but larger overall error.*

Using weights

Both equiripple and least-squares designs can be further controlled by using weights to instruct the algorithm to provide a better approximation to the ideal filter in certain bands. This is useful if it is desired to have less ripple in one band than in another.

Example 4 *In Example 2 above, the filter that was designed had the same ripples in the passband and in the stopband. This is because we implicitly were using a weight of one for each band. If it is desired to have a stopband ripple that is say ten times* smaller *than the passband ripple, we must give a weight that is ten times* larger:

```
Heq2 = design(Hf,'equiripple','Wpass',1,'Wstop',10);
```

The result is plotted in Figure 1.6.

It would be desirable to have an analytic relation between the maximum ripples in a band and the weight in such band. Unfortunately no such relation exists. If the design specifications require a specific maximum ripple amount, say δ_p in the passband and δ_s in the stopband (both in linear units, not decibels), for a lowpass filter we can proceed as follows:

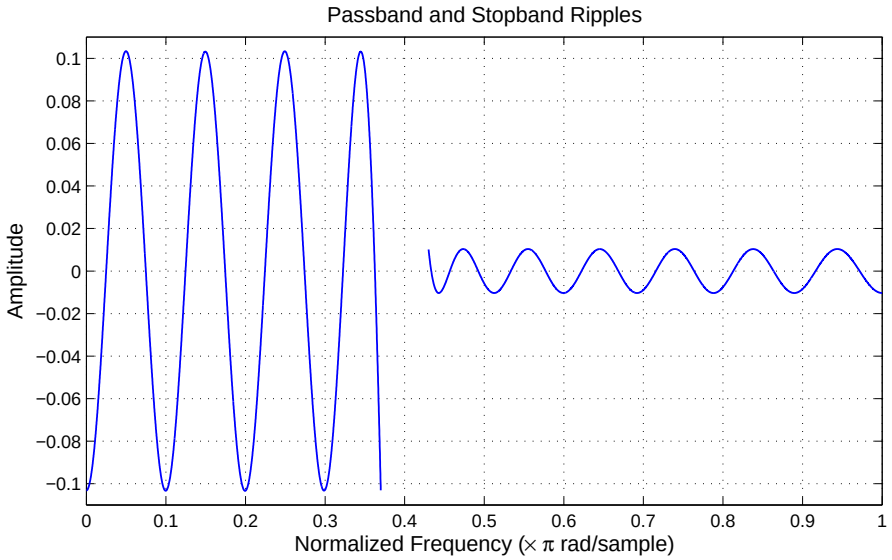


Figure 1.6: *Passband and stopband ripples obtained from weighing the stopband 10 times higher than the passband.*

1. Set the passband weight to one.

2. Set the stopband weight to $\frac{\delta_p}{\delta_s}$.

since both the filter order and the transition width are assumed to be fixed, this will *not* result in the desired ripples unless we are very lucky. However, the relative amplitude of the passband ripple relative to the stopband ripple will be correct. In order to obtain a ripple of δ_p in the passband and δ_s in the stopband we need to vary either the filter order or the transition width.

The procedure we have just described requires trial-and-error since either the filter order or the transition width may need to be adjusted many times until the desired ripples are obtained. Instead of proceeding in such manner, later we will describe ways of designing filters for given passband/stopband ripples and either a fixed transition width or a fixed filter order.

For least-squares designs, the relative weights control not the amplitude of the ripple but its energy relative to the bandwidth it occupies. This

means that if we weigh the stopband ten times higher than the passband, the energy in the stopband relative to the stopband bandwidth will be 10 times smaller than the energy of the ripples in the passband relative to the passband bandwidth. For the case of lowpass filters these means that

$$\frac{E_{\text{sb}}}{\pi - \omega_s} = \frac{2}{2\pi(\pi - \omega_s)} \int_{\omega_s}^{\pi} \left| H(e^{j\omega}) \right|^2 d\omega$$

will be ten times smaller than

$$\frac{E_{\text{pb}}}{\omega_p} = \frac{2}{2\pi(\omega_p)} \int_{0}^{\omega_p} \left| H(e^{j\omega}) \right|^2 d\omega.$$

Minimum-phase designs

One of the advantages of FIR filters, when compared to IIR filters, is the ability to attain exact linear phase in a straightforward manner. As we have already mentioned, the linear phase characteristic implies a symmetry or antisymmetry property for the filter coefficients. Nevertheless, this symmetry of the coefficients constraints the possible designs that are attainable. This should be obvious since for a filter with $N+1$ coefficients, only $N/2+1$ of these coefficients are freely assignable (assuming N is even). The remaining $N/2$ coefficients are immediately determined by the linear phase constraint.

If one is able to relax the linear phase constraint (i.e. if the application at hand does not require a linear phase characteristic), it is possible to design minimum-phase equiripple filters that are superior to optimal equiripple linear-phase designs based on a technique described in [8].

Example 5 *For the same specification set of Example 2 the following minimum-phase design has both smaller peak passband ripple and smaller peak stopband ripple* than the linear-phase equiripple design of that example:*

```
Hmin = design(Hf,'equiripple','Wpass',1,'Wstop',10,...
  'minphase',true);
```

It is important to note that this is not a totally unconstrained design. The minimum-phase requirement restricts the resulting filter to have all

* This can easily be verified using the measure command.

its zeros on or inside the unit circle.* However, the design is optimal in the sense that it satisfies the minimum-phase alternation theorem [9].

Having smaller ripples for the same filter order and transition width is not the only reason to use a minimum-phase design. The minimum-phase characteristic means that the filter introduces the lowest possible phase offset (that is, the smallest possible transient delay) to a signal being filtered.

Example 6 *Compare the delay introduced by the linear-phase filter of Example 2 to that introduced by the minimum-phase filter designed above. The signal to be filtered is a sinusoid with frequency* 0.1π *rad/sample.*

```
n    = 0:500;
x    = sin(0.1*pi*n');
yeq  = filter(Heq,x);
ymin = filter(Hmin,x);
```

The output from both filters are plotted overlaid in Figure 1.7. The delay introduced is equal to the group delay of the filter at that frequency. Since group-delay is the negative of the derivative of phase with respect to frequency, the group-delay of a linear-phase filter is a constant equal to half the filter order. This means that all frequencies are delayed by the same amount. On the other hand, minimum-phase filters do not have constant group-delay since their phase response is not linear. The group-delays of both filters can be visualized using `fvtool(Heq,Hmin,'Analysis','Grpdelay');`*. The plot of the group-delays is shown in Figure 1.8.*

1.3.2 Optimal equiripple designs with fixed transition width and peak passband/stopband ripple

We have seen that the optimal equiripple designs outperform Kaiser-window designs for the same order and transition width. The differences are even more dramatic when the passband ripple and stopband ripple specifications are different. The reason is that the truncated-and-windowed impulse response methods always give a result with approximately the same passband and stopband peak ripple. Therefore, always the more stringent

* Given any linear-phase FIR filter with non negative zero-phase characteristic, it is possible to extract the minimum-phase spectral factor using the firminphase function.

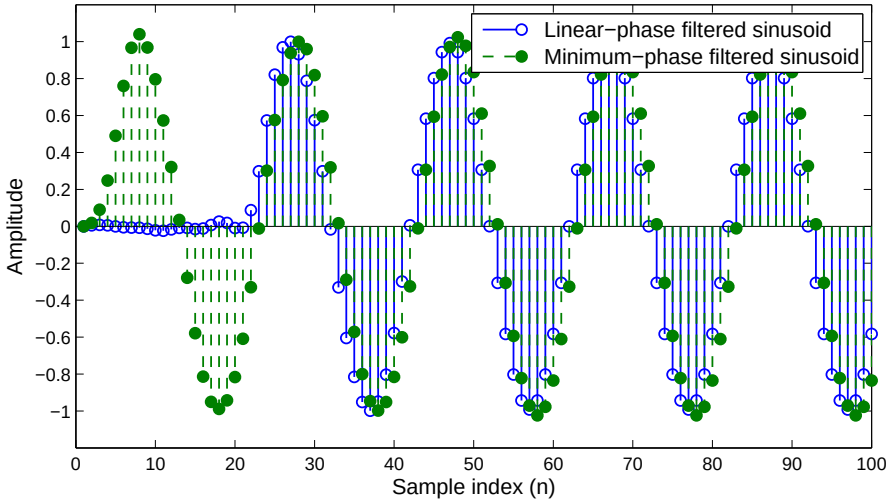


Figure 1.7: *Sinusoid filtered with a linear-phase filter and a minimum-phase filter of the same order.*

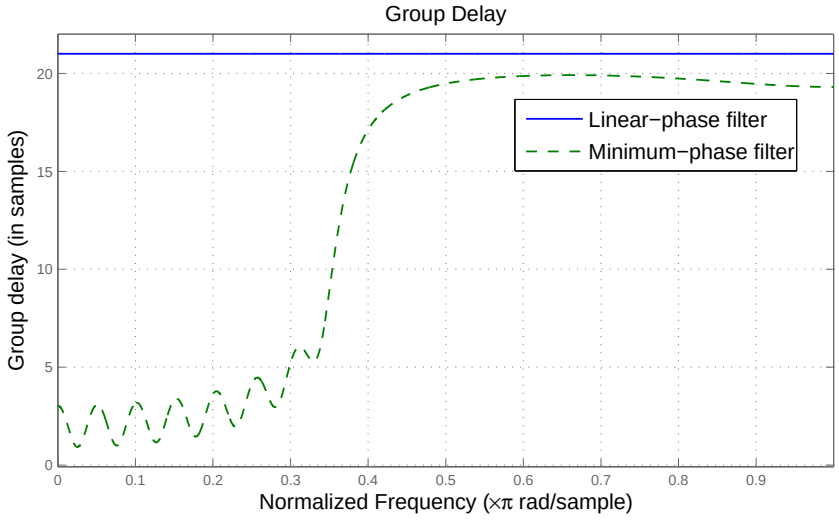


Figure 1.8: *Group-delay of a linear-phase filter and a minimum-phase filter of the same order.*

peak ripple constraint is satisfied, resulting in exceeding (possibly significantly) all other ripple constraints at the expense of unnecessarily large filter order.

To illustrate this, we turn to a different equiripple design in which both the peak ripples and the transition width are fixed. Referring back to the triangle in Figure 1.2, this means the resulting filter order will come from the design algorithm.

Example 7 *Consider the following specifications:*

Specifications Set 2

1. *Cutoff frequency:* 0.375π *rad/sample*
2. *Transition width:* 0.15π *rad/sample*
3. *Maximum passband ripple: 0.13 dB*
4. *Minimum stopband attenuation: 60 dB*

An equiripple design of this filter,

```
Fp  = 0.375 - 0.15/2; Fst = 0.375 + 0.15/2;
Hf  = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,0.13,60);
Heq = design(Hf,'equiripple');
cost(Heq)
```

results in a filter of 37th order (38 taps) as indicated by the `cost` *command. By comparison, a Kaiser-window design requires a 49th order filter (50 taps) to meet the same specifications. The passband details can be seen in Figure 1.9. It is evident that the Kaiser-window design over-satisfies the requirements significantly.*

Minimum-phase designs with fixed transition width and peak passband/stopband ripple

The same procedure to design minimum-phase filters with fixed filter order and fixed transition width can be used to design minimum-phase filters with fixed transition width and peak passband/stopband ripple. In this case, rather than obtaining smaller ripples, the benefit is meeting the same transition width and peak passband/stopband ripples with a reduced filter order.

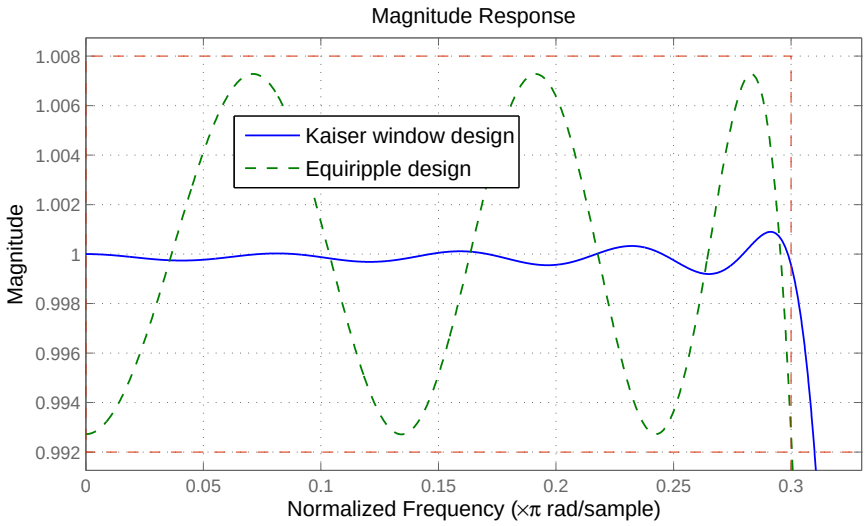


Figure 1.9: *Passband ripple details for both the Kaiser-window-designed FIR filter and the equiripple-designed FIR filter. The Kaiser-window design over-satisfies the requirement at the expense of increase number of taps.*

Example 8 *Consider the following specifications:*

Specifications Set 3

1. *Cutoff frequency:* 0.13π *rad/sample*
2. *Transition width:* 0.02π *rad/sample*
3. *Maximum passband ripple: 0.175 dB*
4. *Minimum stopband attenuation: 60 dB*

The minimum order needed to meet such specifications with a linear-phase FIR filter is 262. This filter must be the result of an optimal equiripple design. If we relax the linear-phase constraint however, the `equiripple` *design (based on the algorithm proposed in [8]) results in a minimum-phase FIR filter of 216th order that meets the specifications:*

```
Hf   = fdesign.lowpass('Fp,Fst,Ap,Ast',.12,.14,.175,60);
Hmin = design(Hf,'equiripple','minphase',true);
cost(Hmin)
```

Note that for these designs minimum-phase filters will have a much smaller transient delay not only because of their minimum-phase property but also because their filter order is lower than that of a comparable linear-phase filter. In fact this is true in general we had previously seen that a filter with the same transition width and filter order with minimum-phase has a smaller delay than a corresponding linear-phase filter. Moreover, the minimum-phase filter has smaller ripples so that the two filters are not really comparable. In order to compare apples to apples, the order of the linear-phase filter would have to increased until the ripples are the same as those of the minimum-phase design. This increase in filter order would of course further increase the delay of the filter.

1.3.3 Optimal equiripple designs with fixed peak ripple and filter order

So far we have illustrated equiripple designs with fixed transition width and fixed order and designs with fixed transition width and fixed peak ripple values. The Filter Design Toolbox also provides algorithms for designs with fixed peak ripple values and fixed filter order [10]. This gives maximum flexibility in utilizing the degrees of freedom available to design an FIR filter.

We have seen that, when compared to Kaiser-window designs, fixing the transition width and filter order results in an optimal equiripple design with smaller peak ripple values, while fixing the transition width and peak ripple values results in a filter with less number of taps. Naturally, fixing the filter order and the peak ripple values should result in a smaller transition width.

Example 9 *Consider the following design of an equiripple with the same cutoff frequency as in Example 7. The filter order is set to be the same as that needed for a Kaiser-window design to meet the ripple specifications*

```
Hf  = fdesign.lowpass('N,Fc,Ap,Ast',49,0.375,0.13,60);
Heq = design(Hf,'equiripple');
```

The comparison of this new design with the Kaiser-window design is shown in Figure 1.10. The transition width has been reduced from 0.15π *to approximately* 0.11π.

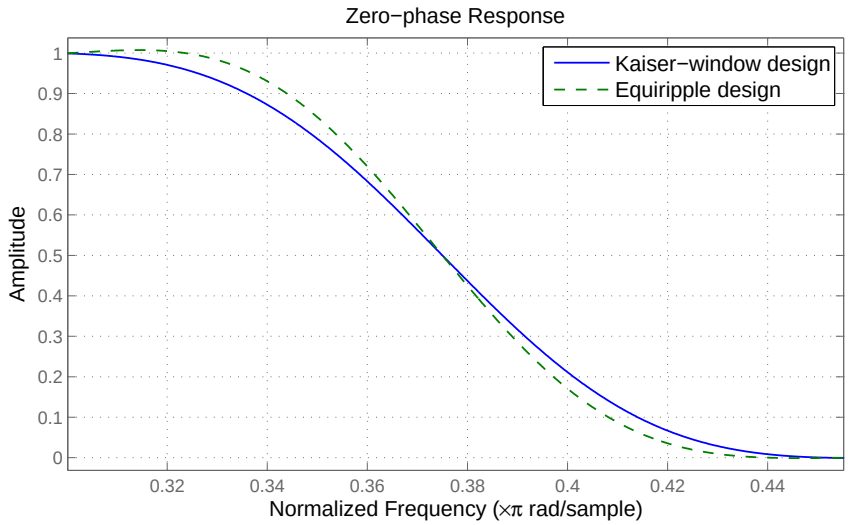


Figure 1.10: *Comparison of a Kaiser-window-designed FIR filter and an optimal equiripple FIR filter of the same order and peak ripple values. The equiripple design results in a reduced transition-width.*

Minimum-phase designs with fixed peak ripple and filter order

Once again, if linear-phase is not a requirement, a minimum-phase filter can be designed that is a superior in some sense to a comparable linear-phase filter. In this case, for the same filter order and peak ripple value, a minimum-phase design results in a smaller transition width than a linear-phase design.

Example 10 *Compared to the 50th order linear-phase design* `Heq`*, the following design has a noticeably smaller transition width:*

```
Hmin = design(Hf,'equiripple','minphase',true);
```

1.3.4 Constrained-band equiripple designs

Sometimes when designing lowpass filters (for instance for decimation purposes) it is necessary to guarantee that the stopband of the filter begins at a specific frequency value and that the filter provides a given minimum stopband attenuation.

If the filter order is fixed - for instance when using specialized hardware - there are two alternatives available in the Filter Design Toolbox for optimal equiripple designs. One possibility is to fix the transition width, the other is to fix the passband ripple.

Example 11 *For example, the design specifications of Example 7 call for a stopband that extends from* 0.45π *to* π *and provide a minimum stopband attenuation of 60 dB. Instead of a minimum-order design, suppose the filter order available is 40 (41 taps). One way to design this filter is to provide the same maximum passband ripple of 0.13 dB but to give up control of the transition width. The result will be a filter with the smallest possible transition width for any linear-phase FIR filter of that order that meets the given specifications.*

```
Hf  = fdesign.lowpass('N,Fst,Ap,Ast',40,.45,0.13,60);
Heq = design(Hf,'equiripple');
```

If instead we want to fix the transition width but not constrain the passband ripple, an equiripple design will result in a filter with the smallest possible passband ripple for any linear-phase FIR filter of that order that meets the given specifications.

```
Hf  = fdesign.lowpass('N,Fp,Fst,Ast',40,.3,.45,60);
Heq = design(Hf,'equiripple');
```

The passband details of the two filters are shown in Figure 1.11. Note that both filters meet the Specifications Set 2 because the order used (40) is larger than the minimum order required (37) by an equiripple linear phase filter to meet such specifications. The filters differ in how they "use" the extra number of taps to better approximate the ideal lowpass filter.

1.3.5 Sloped equiripple filters

In many cases it is desirable to minimize the energy in the stopband of a signal being filtered. One common case is in the design of decimation filters. In this case, the energy in the stopband of a signal after being filtered aliases into the passband region. To minimize the amount of aliasing, we want to minimize the stopband energy. Least-squares filters can be used for this, however, the drawback is that the passband presents larger fluctuations than may be desirable. An alternative is to design optimal

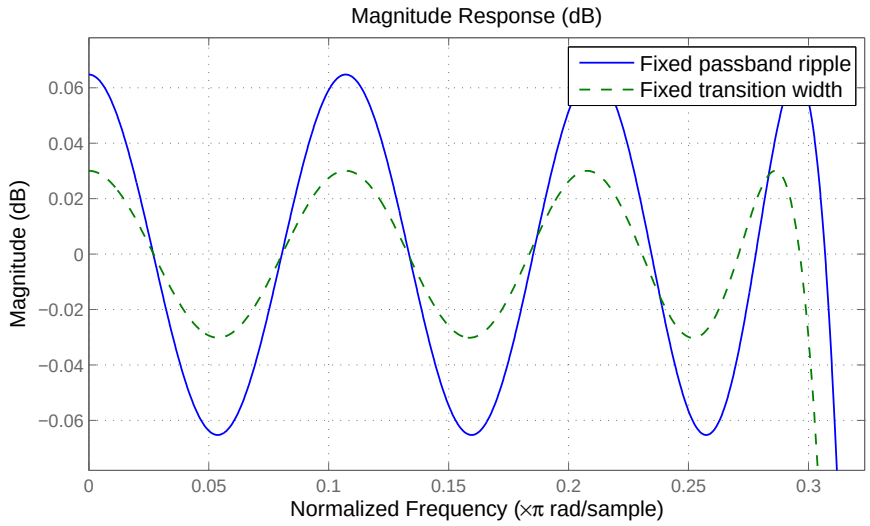


Figure 1.11: *Comparison of a two optimal equiripple FIR filters of 40th order. Both filters have the same stopband-edge frequency and minimum stopband attenuation. One is optimized to minimize the transition width while the other is optimized to minimize the passband ripple.*

equiripple filters but allowing for a slope in the stopband of the filter. The passband remains equiripple thus minimizing the distortion of the input signal in that region.

There are many ways of shaping the slope of the stopband. One way [11] is to allow the stopband to decay as $(1/f)^k$, that is as a power of the inverse of frequency. This corresponds to a decay of $6k$ dB per octave. Another way of shaping the slope is to allow it to decay in logarithmic fashion so that the decay appears linear in dB scale.

Of course there is a price to pay for the sloped stopband. Since the design provides smaller stopband energy than a regular equiripple design, the passband ripple, although equiripple, is larger than that of a regular equiripple design. Also, the minimum stopband attenuation measured in dB is smaller than that of a regular equiripple design.

Example 12 *Consider an equiripple design similar to that of Example 2 but with a stopband decaying as* $(1/f)^2$*:*

```
Fp  = 0.4 - 0.06/2; Fst = 0.4 + 0.06/2;
```

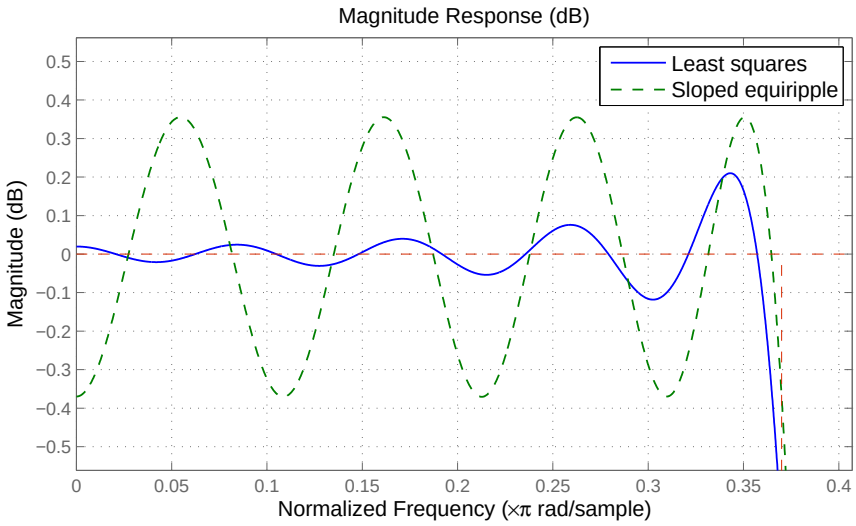


Figure 1.12: *Passband details of a sloped optimal equiripple FIR design and an optimal least-squares FIR design. The equiripple filter has a smaller peak error or smaller transition width depending on the interpretation.*

```
Hf  = fdesign.lowpass('N,Fp,Fst',42,Fp,Fst);
Hsloped = design(Hf,'equiripple','StopbandShape','1/f',...
   'StopbandDecay',2);
```

results in a stopband energy of approximately 8.4095e-05, not much larger that the least-squares design (6.6213e-005), while having a smaller transition width (or peak passband ripple - depending on the interpretation). The passband details of both the least-squares design and the sloped equiripple design are shown in Figure 1.12 (in dB). The stopband details are shown in Figure 1.13 (also in dB).

If we constrain the filter order, the passband ripple, and the minimum stopband attenuation, it is easy to see the trade-off between a steeper slope and the minimum stopband attenuation that can be achieved. Something has to give and since everything else is constrained, the transition width increases as the slope increases as well.

Example 13 *Design two filters with the same filter order, same passband ripple, and same stopband attenuation. The slope of the stopband decay is zero for the first filter and 40 for the second.*

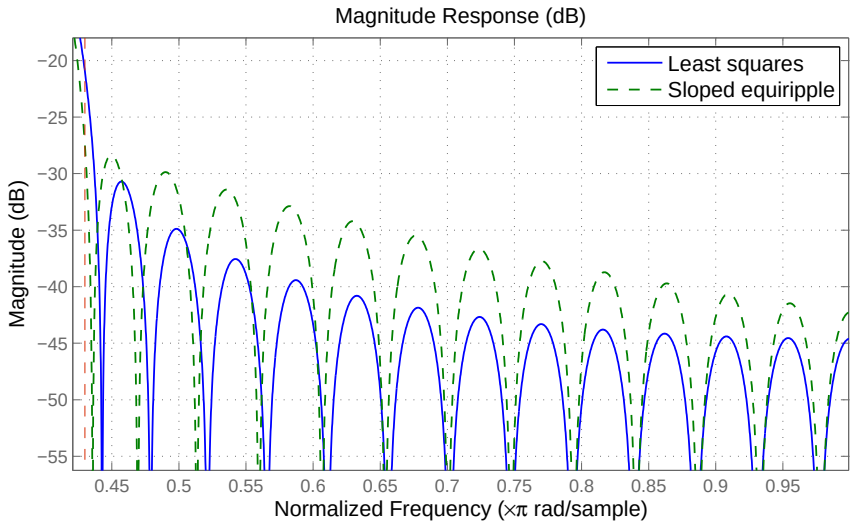


Figure 1.13: *Stopband details of a sloped optimal equiripple FIR design and an optimal least-squares FIR design. The overall error of the equiripple filter approaches that of the least-squares design.*

```
Hf   = fdesign.lowpass('N,Fst,Ap,Ast',30,.3,0.4,40);
Heq  = design(Hf,'equiripple','StopbandShape','linear',...
   'StopbandDecay',0);
Heq2 = design(Hf,'equiripple','StopbandShape','linear',...
   'StopbandDecay',40);
```

The second filter provides better total attenuation throughout the stopband. Since everything else is constrained, the transition width is larger as a consequence. This is easy to see with `fvtool(Heq,Heq2)`.

It is also possible to design minimum-phase sloped equiripple filters. These designs possess similar advantages over linear-phase designs as those described for other equiripple designs when linearity of phase is not a design requirement.

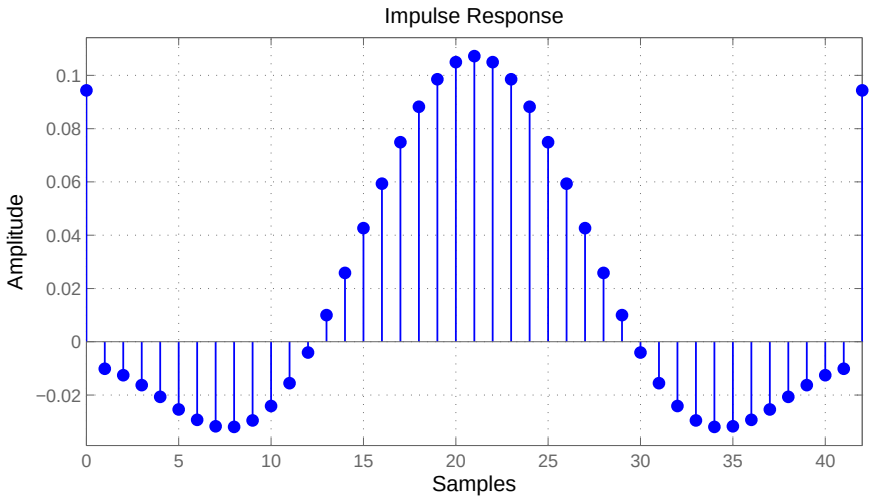


Figure 1.14: *Impulse response of equiripple filter showing anomalous end-points. These end points are the result of an equiripple response.*

1.4 Unusual end-points in the impulse response of equiripple designs

In some cases the impulse response of an equiripple design appears to be anomalous. In particular, the end points seem to be incorrect. However, these end points are in fact a consequence of an equiripple stopband. If we remove the anomaly by say repeating the value in the sample closest to the end point, the stopband is no longer equiripple.

Example 14 *Consider the design of this lowpass filter with band edges that are quite close to DC:*

```
Hf  = fdesign.lowpass('N,Fp,Fst',42,0.1,0.12);
Heq = design(Hf,'equiripple');
fvtool(Heq,'Analysis','impulse')
```

The impulse response is show in Figure 1.14. Notice that the two end points seem completely out of place.

If the anomalous end-points are a problem*, it is not always feasible to modify them by replacing their value with that of their nearest neighbor because sometimes it is more than just the two end points that are anomalous. Moreover, simply using the nearest neighbor removes the equiripple property of the stopband in an uncontrolled way. It is preferable to use a sloped equiripple design to control the shape of the stopband and at the same time remove the anomalies.

Example 15 *Compare the magnitude response and the impulse response of these two designs using* `fvtool`.

```
Hf   = fdesign.lowpass('N,Fp,Fst',800,.064,.066);
Heq  = design(Hf,'equiripple','Wpass',1 ,'Wstop',10);
Heq2 = design(Hf,'equiripple','Wpass',1 ,'Wstop',10,...
  'StopbandShape','linear','StopbandDecay',80);
fvtool(Heq,Heq2)
```

A look at the impulse response reveals that adding a sloped stopband has altered several of the end points of the impulse response removing the discontinuity.

1.5 Maximally-flat FIR filters

Maximally-flat filters are filters in which both the passband and stopband are as flat as possible given a specific filter order.

As we should come to expect, the flatness in the bands comes at the expense of a large transition band (which will also be maximum). There is one less degree of freedom with these filters than with those we have look at so far. The only way to decrease the transition band is to increase the filter order.

Example 16 *Figure 1.15 shows two maximally-flat FIR filters. Both filters have a cutoff frequency of* 0.3π. *The filter with smaller transition width has twice the filter order as the other.*

* For instance, when the impulse response is very long, sometimes only a subset of it is stored in memory and the rest of the values are computed by curve fitting or some other interpolation scheme. In such cases the anomalous points pose a problem because the curve fitting is unlikely to reproduce them accurately.

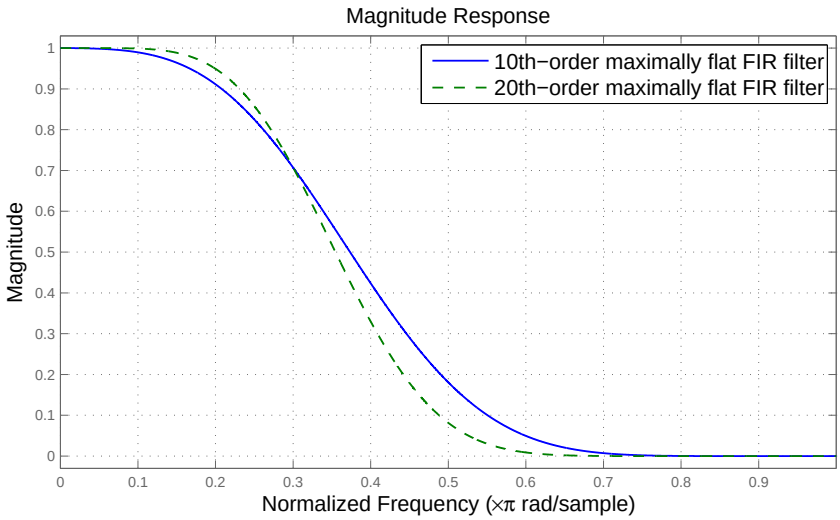


Figure 1.15: *Maximally-flat FIR filters. The smaller transition width in one of the filters is achieved by increasing the filter order.*

The Maximally-flat stopband of the filter means that its stopband attenuation is very large. However, this comes at the price of a very large transition width. These filters are seldom used in practice (in particular with fixed-point implementations) because when the filter's coefficients are quantized to a finite number of bits, the large stopband cannot be achieved (and often is not required anyway) but the large transition band is still a liability.

Example 17 *Compare the stopband attenuation of a maximally-flat FIR filter implemented using double-precision floating-point arithmetic with that of the same filter implemented with 16-bit fixed-point coefficients. The comparison of the two implementations is shown in Figure 1.16. The fixed-point implementation starts to differ significantly from the floating-point implementation at about 75-80 dB. Nevertheless, both filters have the same large transition width.*

The maximally-flat passband may be desirable because it causes minimal distortion of the signal to be filtered in the frequency band that we wish to conserve after filtering. So it may seem that a maximally-flat passband and an equiripple or perhaps sloped stopband could be a thought-after combination. However, if a small amount of ripple is allowed in the

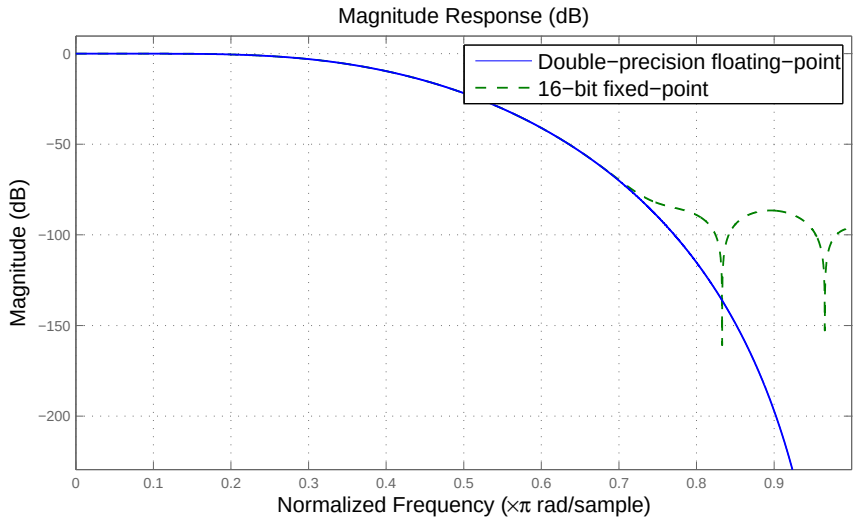


Figure 1.16: *A maximally-flat FIR filter implemented with double-precision floating-point arithmetic and the same filter implemented with 16-bit fixed-point coefficients.*

passband it is always possible to get a smaller transition band and most applications can sustain a small amount of passband ripple.

Example 18 *We can approach a maximally-flat passband by making the passband ripple of an equiripple design progressively smaller. However, for the same filter order and stopband attenuation, the transition width increases as a result. Consider the following two filter designs:*

```
Hf   = fdesign.lowpass('N,Fc,Ap,Ast',70,.3,1e-3,80);
Heq  = design(Hf,'equiripple');
Hf2  = fdesign.lowpass('N,Fc,Ap,Ast',70,.3,1e-8,80);
Heq2 = design(Hf2,'equiripple');
```

The two filters are shown in Figure 1.17. It is generally best to allow some passband ripple as long as the application at hand supports it given that a smaller transition band results. The passband details are shown in Figure 1.18.

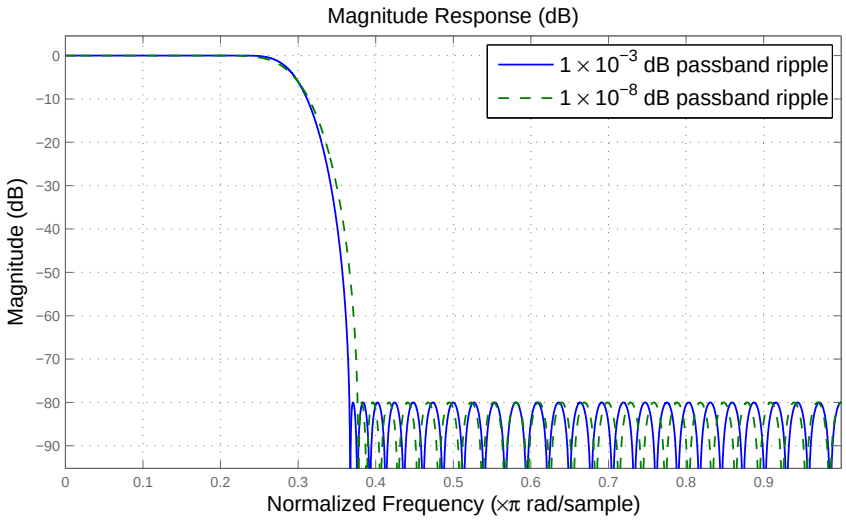


Figure 1.17: *Equiripple filters with passband approximating maximal flatness. The better the approximation, the larger the transition band.*

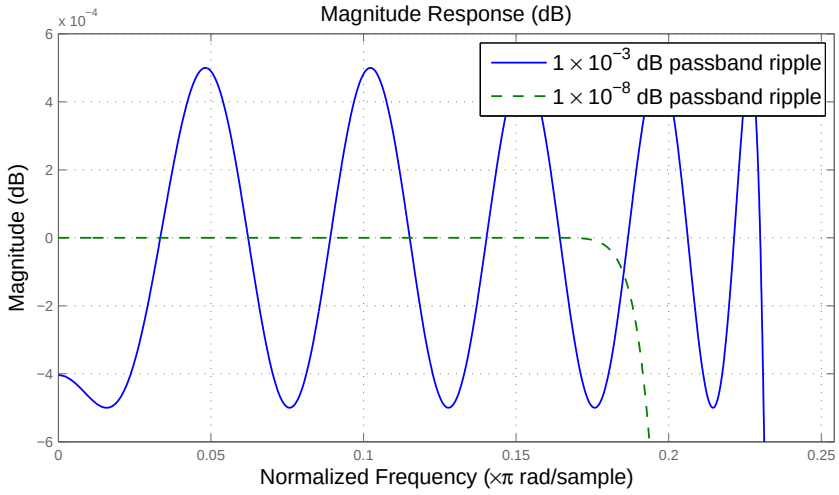


Figure 1.18: *Passband details of equiripple filters with very small passband ripple. The flatter passband is obtained at the expense of a larger transition band, i.e. a smaller usable passband.*

1.6 Summary and look ahead

Understanding FIR filter design is a matter of understanding the trade offs involved and the degrees of freedom available. A drawback of FIR filters is that the tend to require a large filter order and therefore a high computational cost to achieve the specifications desired. There are many ways of addressing this. One is to use IIR filters. Another is to use multistage and/or multirate techniques that use various FIR filters connected in cascade (in series) in such a way that each filter shares part of the filtering duties while having reduced complexity when compared to a single-stage design. The idea is that for certain specifications to combined complexity of the filters used in multistage design is lower than the complexity of a comparable single-stage design.

We will be looking at all these approaches in the following chapters. We will then look into implementation of filters and discuss issues that arise when implementing a filter using fixed-point arithmetic.

Chapter 2

Basic IIR Filter Design

Overview

One of the drawbacks of FIR filters is that they require a large filter order to meet some design specifications. If the ripples are kept constant, the filter order grows inversely proportional to the transition width. By using feedback, it is possible to meet a set of design specifications with a far smaller filter order than a comparable FIR filter*. This is the idea behind IIR filter design. The feedback in the filter has the effect that an impulse applied to the filter results in a response that never decays to zero, hence the term infinite impulse response (IIR).

We will start this chapter by discussing classical IIR filter design. The design steps consist of designing the IIR filter in the analog-time domain where closed-form solutions are well known and then using the bilinear transformation to convert the design to the digital domain. We will see that the degrees of freedom available are directly liked to the design algorithm chosen. Butterworth filters provide very little control over the resulting design since it is basically a maximally-flat design. Chebyshev designs increase the degrees of freedom by allowing ripples in the passband (type I Chebyshev) or the stopband (type II Chebyshev). Elliptic filters allow for maximum degrees of freedom by allowing for ripples in both the passband and the stopband. In fact, Chebyshev and Butterworth designs can be seen as a special case of elliptic designs, so usually one should only concentrate on elliptic filter design, decreasing the passband

* However, see Chapter 5.

ripples and/or increasing the stopband attenuation enough to approach Chebyshev or Butterworth designs when so desired.

After looking at classical designs, we move on to examine design techniques that are based on optimization directly in the digital domain.

2.1 Why IIR filters

IIR filters are usually used when computational resources are at a premium. However, stable, causal IIR filters cannot have perfectly linear phase, so that IIR filters tend to be avoided when linearity of phase is a requirement. Also, computational savings that are comparable to using IIR filters can be achieved for certain design parameters by using multistage/multirate FIR filter designs (see Chapter 5). These designs have the advantage of having linear-phase capability, robustness to quantization, stability, and good pipeline-ability.

Another important reason to use IIR filters is their relatively small group delay compared to FIR filters. This results in a shorter transient response to input stimuli. Even if we are able to match or surpass the implementation cost of an IIR filter by using multirate/multistage FIR designs, the transient response of such FIR designs tends to be much larger than that of a comparable IIR design. However, minimum-phase FIR designs such as those seen in Chapter 1, can have group-delays that are comparable or even lower than IIR filters that meet the same specifications.

2.2 Classical IIR design

We begin by discussing filters that are designed in the analog-time domain and then transformed to the digital domain by means of the bilinear transformation. This methodology covers the following IIR design techniques: Butterworth, Chebyshev type I, Chebyshev type II, and elliptic (Cauer). We follow the methodology presented in [12]-[13] in which the 3-dB point is always a design specification.

2.2.1 Cutoff frequency and the 3-dB point

For FIR filters, we have started by designating the cutoff frequency for an ideal filter (i.e. one with zero transition-width). However, depending on the design method, the cutoff frequency may lack any meaningful value. For windowed-impulse-response designs, the magnitude response will typically cross the cutoff frequency at the point where the magnitude is equal to 0.5. In decibels, this point corresponds to $20\log_{10}(0.5) = -6.0206$. The reason for this is that with window-based designs, the passband and stopband ripples are about the same. For other designs, such as equiripple, the point at which the magnitude response crosses the middle of the transition band will depend on the ratio of the passband and stopband ripples.

By contrast, the convention for classical IIR designs is to define the cutoff frequency as the point at which the power of an input signal is attenuated by one-half. This means that the cutoff frequency in decibels corresponds to $10\log_{10}(0.5) = -3.0103$ which we loosely call the 3-dB point*.

To avoid confusion, we will call the point at which the power is reduced by one-half the 3-dB point rather than the cutoff frequency. When entering specifications, we use the string `'F3dB'` rather than `'Fc'`.

2.2.2 Butterworth filters

Butterworth filters are maximally-flat IIR filters. For this reason, the only design parameters are the cutoff frequency and the filter order.

Example 19 *Design a 7th order Butterworth filter with a 3 dB point of* 0.3π.

```
Hf = fdesign.lowpass('N,F3db',7,0.3);
Hb = design(Hf,'butter');
```

The design is shown in Figure 2.1. The plot also includes an FIR filter designed with a cutoff frequency of 0.3π. *Notice that the attenuation at* 0.3π *is 3-dB for the Butterworth filter and 6-dB for the FIR filter.*

As with maximally-flat FIR filters, the flatness in the passband and stopband cause the transition band to be very wide. The only way of reducing the transition width is to increase the filter order.

* Just to be clear, the gain of the filter in decibels is -3.0103. The attenuation, i.e. the distance between the passband gain and this point is approximately 3 dB since distance is always positive.

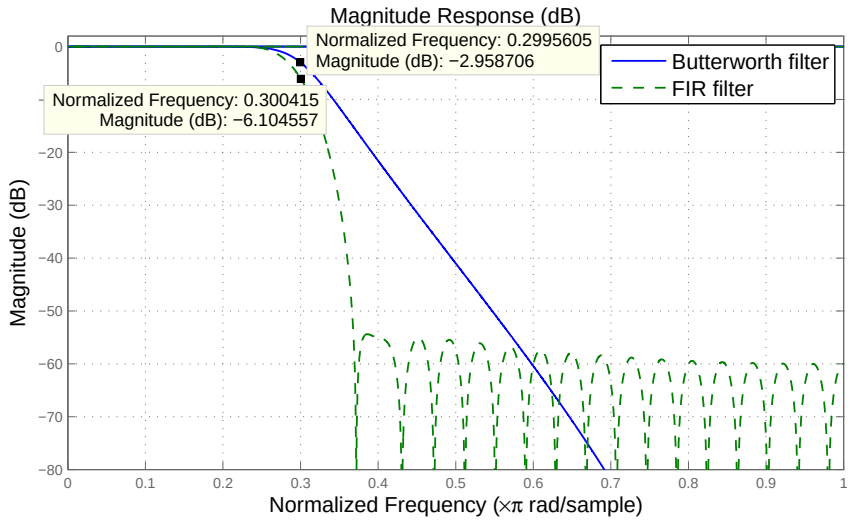


Figure 2.1: *7th order Butterworth filter with a 3-dB frequency of* 0.3π*. Also shown is an FIR filter with a cutoff frequency of* 0.3π*.*

Example 20 *Compare the 7th-order Butterworth filter of the previous example, with a 12th-order Butterworth filter designed with the same 3-dB point. The magnitude-squared responses are shown in Figure 2.2.*

```
Hf2 = fdesign.lowpass('N,F3db',12,0.3);
Hb2 = design(Hf2,'butter');
```

The transition-width is decreased as the filter order increases. Notice that both filters intersect at the point where the magnitude-squared of the filter is equal to 0.5, i.e., the point at which the power of the input signal is attenuated by one half.

2.2.3 Chebyshev type I filters

Chebyshev type I filters can attain a smaller transition width than a Butterworth filter of the same order by allowing for ripples in the passband of the filter. The stopband is, as with Butterworth filters, maximally flat. For a given filter order, the trade-off is thus between passband ripple and transition width.

Example 21 *Compare the 7th-order Butterworth filter from previous examples with a 7th-order Chebyshev type I filter with 1 dB of peak-to-peak passband ripple.*

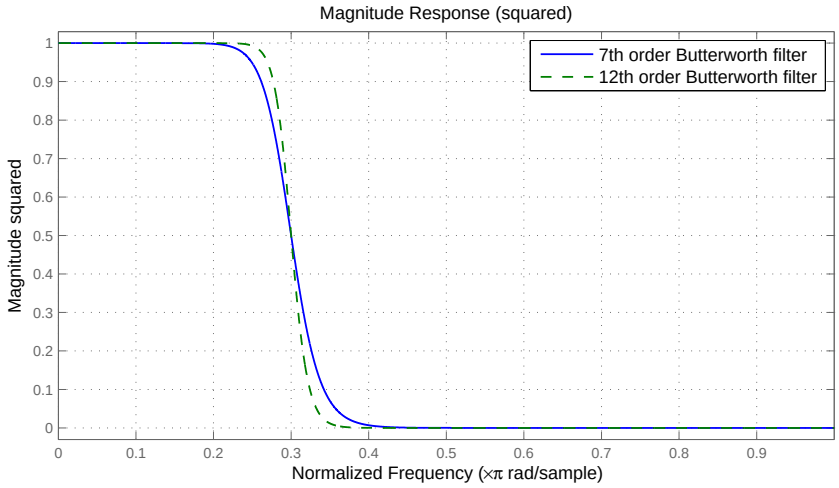


Figure 2.2: *Comparison of a 7th-order and a 12th-order Butterworth filter. The filters intersect at the 3-dB point.*

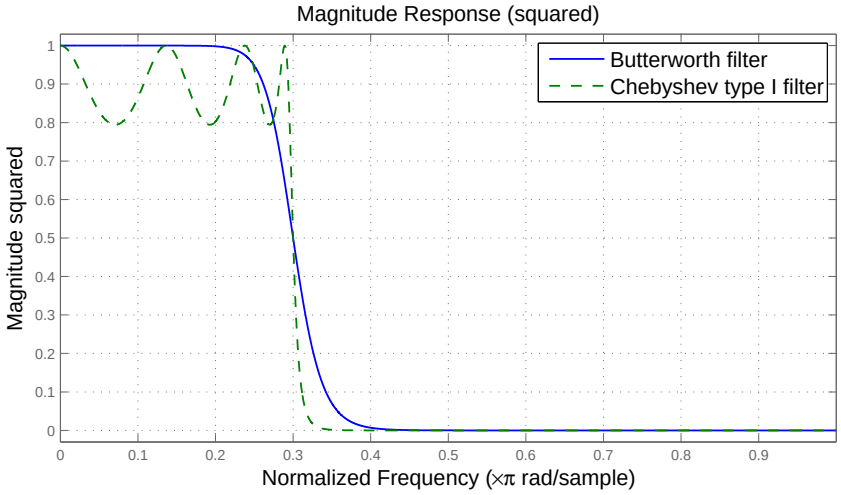


Figure 2.3: *Comparison of a 7th-order Butterworth and a 7th-order Chebyshev type I filter. Once again, the filters intersect at the 3-dB point.*

```
Hf = fdesign.lowpass('N,F3db,Ap',7,0.3,1);
Hc = design(Hf,'cheby1');
```

The magnitude-squared responses are shown in Figure 2.3. The passband ripple

of the Chebyshev filter can be made arbitrarily small, until the filter becomes the same as the Butterworth filter. That exemplifies the trade-off between passband ripple and transition width.

With IIR filters, we not only need to consider the ripple/transition-width trade-offs, but also the degree of phase distortion. We know that linear-phase throughout the entire Nyquist interval is not possible. Therefore, we want to look at how far from linear the phase response is. A good way to look at this is to look at the group-delay (which ideally would be constant) and see how flat it is.

Of the classical IIR filters considered here*, Butterworth filters have the least amount of phase distortion (phase non-linearity). Since Chebyshev and elliptic filters can approximate Butterworth filters arbitrarily closely by making the ripples smaller and smaller, it is clear that the amount of ripple is related to the amount of phase distortion. Therefore, when we refer to the trade-off between ripples and transition-width, we should really say that the trade-off is between the transition-width and the amount of ripple/phase-distortion. As an example, we show the group-delay of the Butterworth and Chebyshev type I filters of the previous example in Figure 2.4. Notice that not only is the group-delay of the Chebyshev filter less flat, it is also larger for frequencies below the 3-dB point (which is the region we care about). This illustrates that although the filters have the same order, a Chebyshev type I filter will have a larger transient-delay than a Butterworth filter.

2.2.4 Chebyshev type II designs

Unlike FIR maximally-flat filters, it is possible to attain very large stopband attenuations with Butterworth or Chebyshev type I filters even when the filter is implemented using fixed-point arithmetic. Nevertheless, few applications require attenuations of hundreds of decibels. With Butterworth filters, such large attenuations come at the price of a large transition width. With Chebyshev type I filters, a trade-off can be made between transition-width and passband ripple. However, attaining a small transition width may result in unacceptably high passband ripple.

* Bessel filters have maximally-flat group-delay responses. However, this property is lost when the bilinear transformation is used to convert the filter from analog to digital. It is possible to use other conversion techniques such as impulse-invariance with Bessel filters [14].

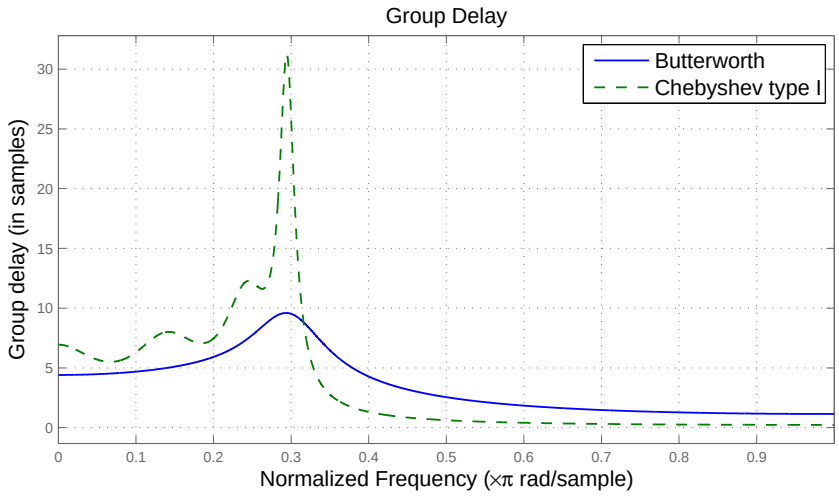


Figure 2.4: *Group-delay responses for a 7th-order Butterworth and a 7th-order Chebyshev type I filter. Both filters have a 3-dB point of* 0.3π.

In most cases a Chebyshev type II (inverse Chebyshev) filter is preferable to both a Chebyshev type I filter and a Butterworth filter. These filters have maximally-flat passband and equiripple stopband. Since extremely large attenuations are typically not required, we may be able to attain the transition-width required by allowing for some stopband ripple.

Example 22 *Design a 6th order filter with a 3-dB point of* 0.45π. *The filter must have an attenuation of at least 80 dB at frequencies above* 0.75π *and the passband ripple must not exceed 0.8 dB.*

```
Hf1 = fdesign.lowpass('N,F3db',6,0.45);
Hf2 = fdesign.lowpass('N,F3db,Ap',6,0.45,0.8);
Hf3 = fdesign.lowpass('N,F3db,Ast',6,0.45,80);
Hb  = design(Hf1,'butter');
Hc1 = design(Hf2,'cheby1');
Hc2 = design(Hf3,'cheby2');
```

The three designs are shown in Figure 2.5. Only the Chebyshev type II filter reaches the required attenuation of 80 dB by 0.75π.

The group-delay responses for the designs of the previous example are shown in Figure 2.6. Although the Butterworth's group-delay is slightly

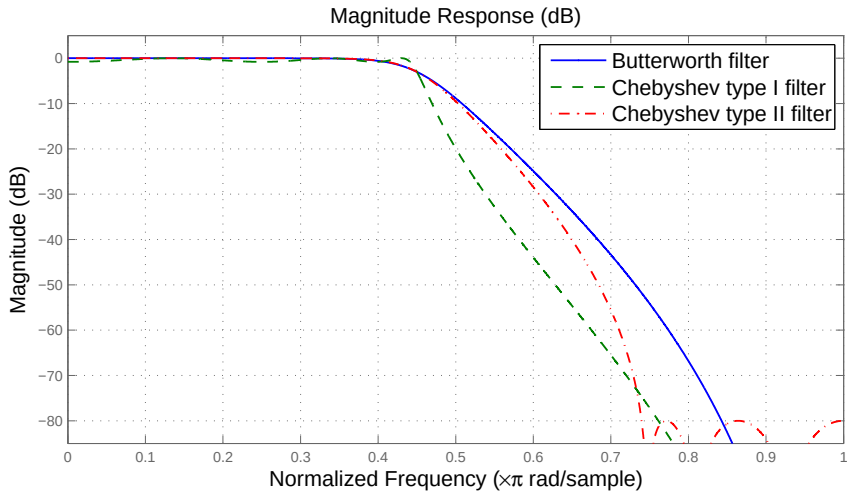


Figure 2.5: *6th order Butterworth, Chebyshev type I, and Chebyshev type II filters with a 3-dB point of* 0.45π.

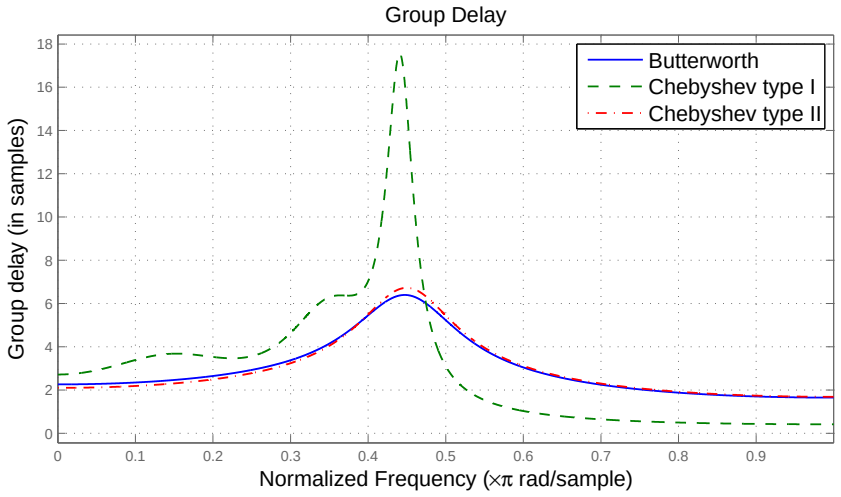


Figure 2.6: *Group-delay responses for filters from Example 22.*

flatter than that of the Chebyshev type II, the latter's group-delay is smaller than the former's for most of the passband.

Even though all three designs in the previous example are of 6th order, the Chebyshev type II implementation actually requires more multipliers as we now explain.

All the designs result in three second-order sections. However, the maximally-flat stopband of the Butterworth and Chebyshev type I filters is achieved by placing multiple zeros at $w = \pi$. The result is that the numerators for all three sections are given by $1 + 2z^{-1} + z^{-2}$. The corresponding coefficients, $\{1,2,1\}$ require no actual multiplications (the multiplication by two can be implemented with a simple shift). There are two multiplications for each denominator of each section plus one more multiplier that provides the correct passband gain (0 dB)*. In other words, both the Butterworth filter and the Chebyshev type I filter can be implemented with 7 multipliers. By contrast, the Chebyshev type II design requires, at least 9 multipliers to implement.

2.2.5 Elliptic filters

Elliptic (Cauer) filters generalize Chebyshev and Butterworth filters by allowing for ripple in both the passband and the stopband of a filter. By making the ripples smaller and smaller, we can approximate arbitrarily close either Chebyshev or Butterworth filters using elliptic filters. The approximation is not just of the magnitude response, but the phase response (and consequently the group-delay response) as well.

We have already said that most applications do not require arbitrarily large stopband attenuations, hence Chebyshev type II filters are generally preferable to Chebyshev type I and Butterworth filters. However, Chebyshev type II filters are maximally-flat in the passband and as a consequence have rather larger transition bands. Since most applications can sustain some amount of passband ripple, the transition band can be reduced by using an elliptic filter with the same amount of stopband attenuation as a Chebyshev type II filter (and the same filter order). The reduced transition band is attained by allowing for some amount of passband ripple. As with Chebyshev type I filters, the passband ripples of elliptic filters cause the group-delay to become larger and more distorted than that of Butterworth or Chebyshev type II filters.

Example 23 *Design an elliptic filter that meets the specifications of Example 22.*

```
Hf4 = fdesign.lowpass('N,F3db,Ap,Ast',6,0.45,.8,80);
He  = design(Hf4,'ellip');
```

* For fixed-point implementations, it may be preferable to break-up this scaling gain into several gains, one per section, because of dynamic range issues.

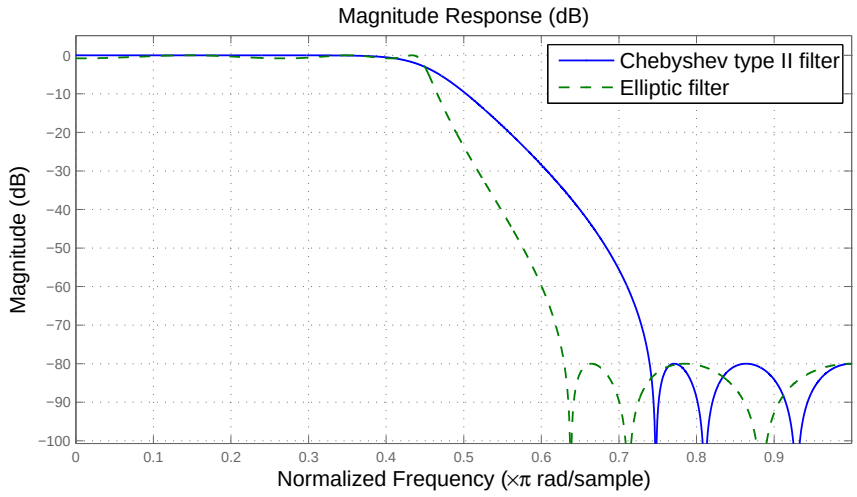


Figure 2.7: *Elliptic and Chebyshev type II filters with a 3-dB point of* 0.45π.

The resulting filter is shown in Figure 2.7 along with the Chebyshev type II filter we had already designed. The passband ripple allows for the 80 dB attenuation to be reached by about 0.63π *rad/sample, far before the required* 0.75π. *If the target application does not benefit from this the passband ripple can be reduced in order to have smaller passband distortion.*

The group-delay of the elliptic filter is similar but even less flat than that of the Chebyshev type I filter. Figure 2.8 shows the group-delay responses of the Chebyshev type I design of Example 22 and the elliptic design of Example 23.

2.2.6 Minimum-order designs

We can use any of the design methods described above for design problems with fixed passband/stopband ripples and fixed transition-widths. Naturally, elliptic filters will result in the lowest filter order due to the permissibility of ripples in both passband and stopband. In fact, elliptic filters are the IIR version of optimal equiripple designs.

Example 24 *Design a highpass IIR filter with a maximum passband ripple of 1 dB and a minimum stopband attenuation of 60 dB. The stopband must extend to* 0.45π *and the passband must begin at* 0.55π.

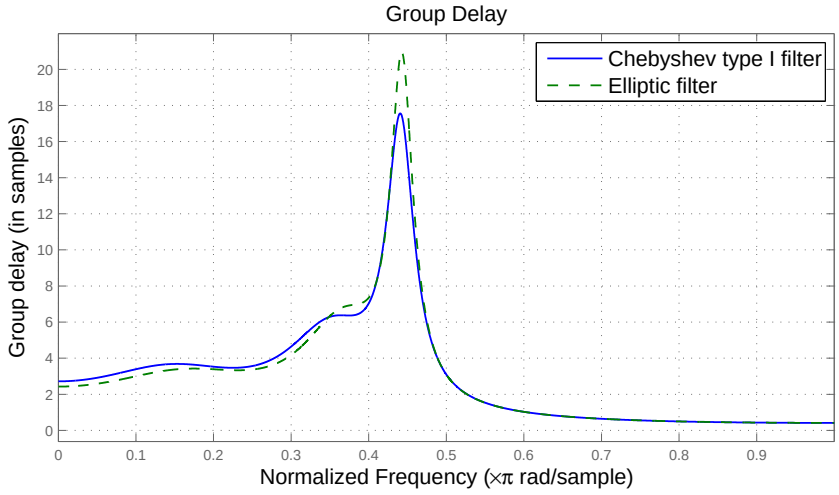


Figure 2.8: *Group-delay response of elliptic and Chebyshev type I filters with a 3-dB point of* 0.45π.

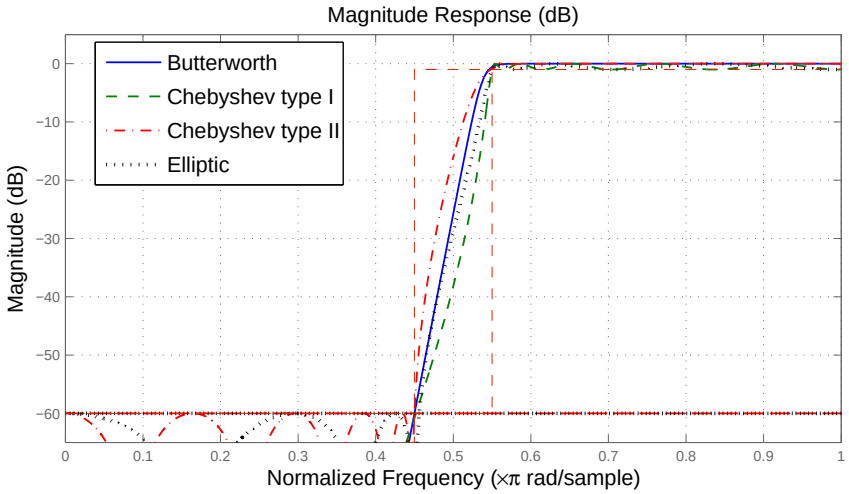


Figure 2.9: *Magnitude responses of highpass filters designed with all four classical design methods.*

```
Hf = fdesign.highpass('Fst,Fp,Ast,Ap',.45,.55,60,1);
Hd = design(Hf,'alliir');
```

The response of all four filters can be visualized using `fvtool(Hd)` *(see Figure 2.9). Since we have designed all four filters,* `Hd` *is an array of filters.* `Hd(1)`

is the Butterworth filter, `Hd(2)` *is the Chebyshev type I,* `Hd(3)` *the Chebyshev type II, and* `Hd(4)` *the elliptic filter. By inspecting the cost of each filter (e.g.* `cost(Hd(2))`*) we can see that the Butterworth filter takes the most multipliers (and adders) to implement while the elliptic filter takes the least. The Chebyshev designs are comparable.*

As with Chebyshev type II filters, elliptic filters do not have trivial numerators of the form $1 - 2z^{-1} + z^{-2}$*. In contrast, the actual cost of Butterworth and Chebyshev type I filters is a little lower than what is indicated by the cost function as long as we are able to implement the multiplication by two without using an actual multiplier (the cost function does not assume that this is the case; it is implementation agnostic).

2.2.7 Comparison to FIR filters

We have said that one reason to use IIR filters is because of their small implementation cost when compared to FIR counterparts. We'd like to illustrate this with an example.

Example 25 *Consider the following design specifications:*

Specifications Set 4

1. *Cutoff frequency:* 0.0625π *rad/sample*
2. *Transition width:* 0.008π *rad/sample*
3. *Maximum passband ripple: 1 dB*
4. *Minimum stopband attenuation: 80 dB*

```
Fc = 0.0625;
TW = 0.008;
Fp = Fc-TW/2;
Fst= Fc+TW/2;
Ap = 1;
Ast= 80;
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast);
```

* Notice the negative sign for the term $2z^{-1}$. This is because we are designing highpass filters rather than lowpass.

If we design an elliptic filter,

```
He = design(Hf,'ellip');
cost(He);
```

the cost method indicates that 20 non-trivial multipliers and 20 adders are needed to implement the filter.

An equiripple design,

```
Heq = design(Hf,'equiripple');
```

requires a whopping 642 multipliers and 641 adders. If we use a symmetric direct-form structure to implement the filter,

```
Heqs = design(Hf,'equiripple','FilterStructure','dfsymfir');
```

the number of multipliers is halved to 321 (the number of adders remains 641).

Clearly there is not contest between the FIR and the IIR design. In the following chapters we will see that with the use FIR multirate/multistage techniques we will be able to achieve designs that are more efficient than the elliptic design shown here in terms not of the number of multipliers, but of the number of multiplications per input sample (MPIS). However, we will then go on to show that by using efficient multirate IIR filters based on allpass decompositions, we can achieve even lower MPIS. Note that for the designs discussed so far, both the FIR and IIR cases (being single-rate), the number of MPIS is the same as the number of multipliers.

In addition to computational cost, another common reason for using IIR filters is their small group-delay when compared to FIR filters. However, let us compare with minimum-phase FIR filters via an example.

Example 26 *Consider the following bandpass filter specifications:*

Specifications Set 5

1. *Lower cutoff frequency:* 0.35π *rad/sample*
2. *Upper cutoff frequency:* 0.55π *rad/sample*
3. *Lower/upper transition width:* 0.1π *rad/sample*
4. *Maximum passband ripple: 0.01 dB*

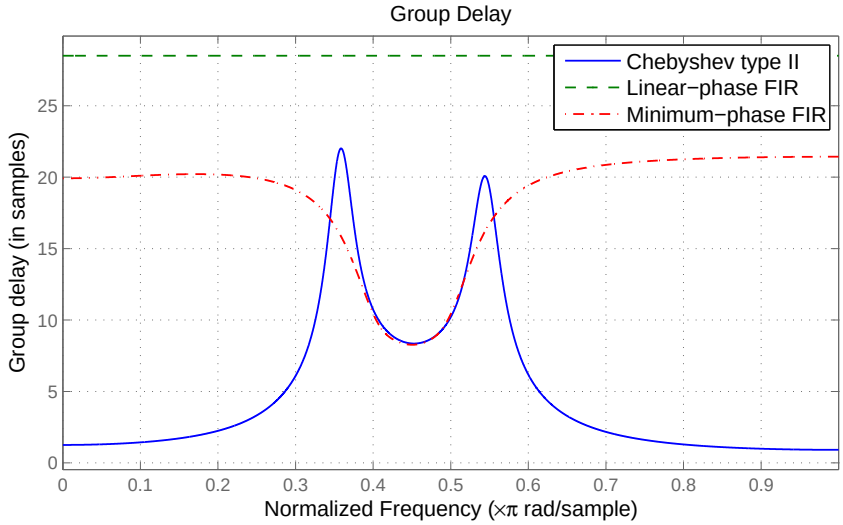


Figure 2.10: *Group-delays of FIR and IIR bandpass filters.*

5. *Minimum stopband attenuation: 40 dB*

```
Fc1  = 0.35;
Fc2  = 0.55;
TW   = 0.1;
Fst1 = Fc1-TW/2;
Fp1  = Fc1+TW/2;
Fp2  = Fc2-TW/2;
Fst2 = Fc2+TW/2;
Ap   = 1e-2;
Ast  = 40;
Hf = fdesign.bandpass(Fst1,Fp1,Fp2,Fst2,Ast,Ap,Ast);
```

In the passband, $[0.4\pi, 0.5\pi]$, of all classical IIR designs, the Chebyshev type II will have the lowest group delay. However, a minimum-phase equiripple FIR filter has an even lower group-delay in the passband as can be seen in Figure 2.10.

2.3 IIR designs directly in the digital domain

In addition to the classical methods based on bilinear transformation on analog designs, the Filter Design Toolbox includes design algorithms based

on optimization directly in the digital domain. The advantage of these algorithms is that they allow for independent control of the numerator and denominator orders, providing more flexibility than classical designs. Moreover, these digital-domain designs allow for optimization of different norms, from least-squares (2-norm) all the way to infinity-norms (equiripple).

We have already stated that elliptic filters are optimal equiripple IIR filters. Essentially the same filters can be designed use the `iirlpnorm` design algorithm if the $\mathcal{L}_\infty$ norm is used as the pth norm.

Example 27 *The following design yields virtually the same filter as the elliptic design of Example 23:*

```
Hf5 = fdesign.lowpass('Nb,Na,Fp,Fst',6,6,.4425,.63);
Hd  = design(Hf5,'iirlpnorm','Wpass',1,'Wstop',450);
```

Unlike classical IIR designs, the passband gain of the least pth norm design is not bounded by zero dB. However, if we normalize the passband gain to 0 dB, we can verify that this filter is virtually identical to the elliptic filter within a scale factor.

To illustrate the extra flexibility afforded by independent control of the numerator and denominator orders, consider the following example in which have a larger order numerator than denominator.

Example 28 *We want to use the* `iirlpnorm` *function to design a filter with less than 0.8 dB passband ripple and at least 80 dB attenuation at* 0.75π *and beyond. All designs we showed so far used a 6th order. The two designs that met the specifications, elliptic and Chebyshev type II, would have failed to meet the specifications had a 4th-order been used instead. Here, we will use a 6th-order numerator and a 4th-order denominator and still meet the specs.*

```
Hf5 = fdesign.lowpass('Nb,Na,Fp,Fst',6,4,.4425,.75);
Hd  = design(Hf5,'iirlpnorm','Wpass',1,'Wstop',25);
```

The design is shown along with the Chebyshev type II filter designed previously in Figure 2.11. The passband ripple is only about 0.043 dB. Despite using less multipliers than the Chebyshev type II filter, the least pth norm filter has a steeper transition width thanks to the fact that there is some passband ripple.

We should note that unlike the classical IIR designs, it is not possible to directly control the location of the 3-dB point with the least pth norm design algorithm.

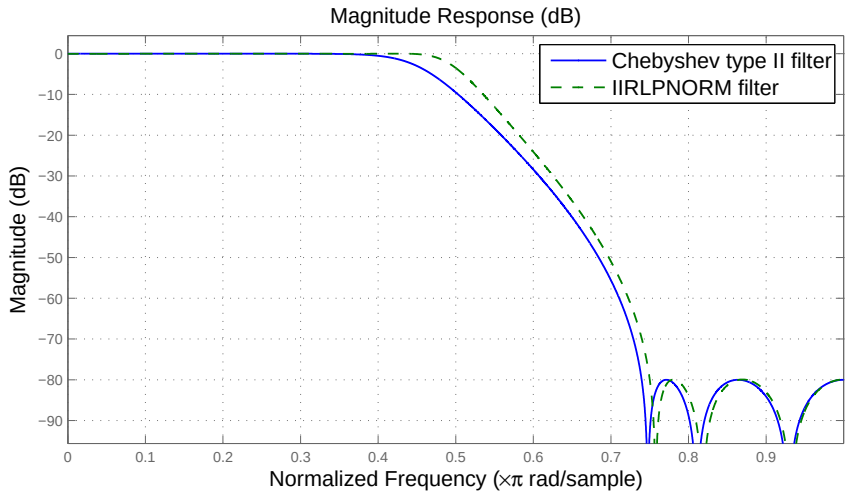


Figure 2.11: *Least pth norm IIR filter with a 6th order numerator and a 4th order denominator. Also shown is the Chebyshev type II filter design in Example 22.*

2.4 Summary and look ahead

Trade-offs and degrees of freedom in IIR filter design are essentially the same as those in FIR filter design. This is particularly true when ripples are allowed in the passband and/or stopband of the filter. Elliptic filters are IIR optimal equiripple filters that use the smallest filter order possible to meet a set of specifications involving ripples and transition band. Butterworth and Chebyshev filters are special cases of elliptic filters with zero passband and/or stopband ripples. However, most applications can sustain some amount of ripple, making elliptic filters usually the way to go. Further flexibility can be obtained by controlling individually the numerator and denominators orders. Least pth norm designs allow for such control.

IIR filters (elliptic in particular) can meet a set of specifications with far fewer multipliers than what an FIR filter would require. However, there is a price to pay in terms of phase distortion, implementation complexity, stability issues when using finite-precision arithmetic, pipelineability and so forth. If the design specifications are well-suited for a multirate/multistage FIR filter design, we may be able to obtain more efficient designs without the drawbacks of IIR filters. We will study such

approaches in the coming chapters. However, we point out once again that these multistage implementations tend to have large transient delays, so that if this is a critical factor, IIR filters may be preferable (although minimum-phase FIR filters can be an interesting option as we have seen).

Chapter 3

Nyquist Filters

Overview

Nyquist filters are a special class of filters which are useful for multirate implementations. Nyquist filters also find applications in digital communications systems where they are used for pulse shaping (often simultaneously performing multirate duties). The widely used raised-cosine filter is a special case of Nyquist filter that is often used in communications standards.

Nyquist filters are also called Lth-band filters because the passband of their magnitude response occupies roughly $1/L$ of the Nyquist interval. The special case, $L = 2$ is widely used and is referred to as Halfband filters. Halfband filters can be very efficient for interpolation/decimation by a factor of 2.

Nyquist filters are typically designed via FIR approximations. However, IIR designs are also possible. We will show FIR and IIR designs of halfband filters. The IIR case results in extremely efficient filters and through special structures such as parallel (coupled) allpass-based structures and wave digital filters can be efficiently implemented for multirate applications.

Later, in Chapter 5, we will see a special property of Nyquist filters when used in multirate applications. The cascade of Nyquist interpolators (or decimators) results in an overall Nyquist interpolator (or decimator). This makes Nyquist filters highly desirable for multistage designs.

Moreover, as we will see, Nyquist filters have very small passband rip-

ple even when the filter is quantized for fixed-point implementations. This makes Nyquist filters ideally suitable for embedded applications.

3.1 Design of Nyquist filters

Although not strictly necessary, the most common design of a Nyquist filter is as a special case of a lowpass filter. Nonetheless, highpass and bandpass designs are also possible. When designing a Nyquist filter, the value of the band L must be specified. Nyquist filters will reduce the bandwidth of a signal by a factor of L.

With Nyquist filters, the passband ripple and the stopband ripple cannot be independently specified. Therefore, in all designs, only the stopband ripple (and/or the transition width) will be given. The passband ripple will be a result of the design. In most cases, the resulting passband ripple is very small. This is particularly true for IIR Nyquist filters.

Example 29 *Design a Kaiser window* 5*th-band Nyquist FIR filter of* 84*th order and 80 dB of stopband attenuation*[*]*,*

```
L = 5; % 5th-band Nyquist filter
f = fdesign.nyquist(L,'N,Ast',84,80);
h = design(f,'kaiserwin');
```

Both the transition-width and the passband ripple of the filter are a result of the design. The filter has a cutoff frequency given by π/L *(*0.2π *in this case). The passband details of the magnitude response are shown in Figure 3.1. Note that the peak to peak ripples are indeed quite small (on the order of* 2×10^{-3}*).*

A characteristic of Nyquist filters is that every Lth sample of their impulse response is equal to zero (save for the middle sample). As we will see when we discuss multirate filter design, this characteristic is desirable for interpolation because it means that the input samples will pass through the filter unchanged. Moreover, for both interpolation and decimation purposes, it means that one of the L polyphase branches will result in a simple delay (no multipliers/adders) reducing the implementation cost

[*] Close inspection of the stopband will show that the 80 dB attenuation is not quite achieved. This happens sometimes with Kaiser-window designs because the relation between the adjustable parameter in the Kaiser window and the stopband attenuation is derived in somewhat empirical fashion.

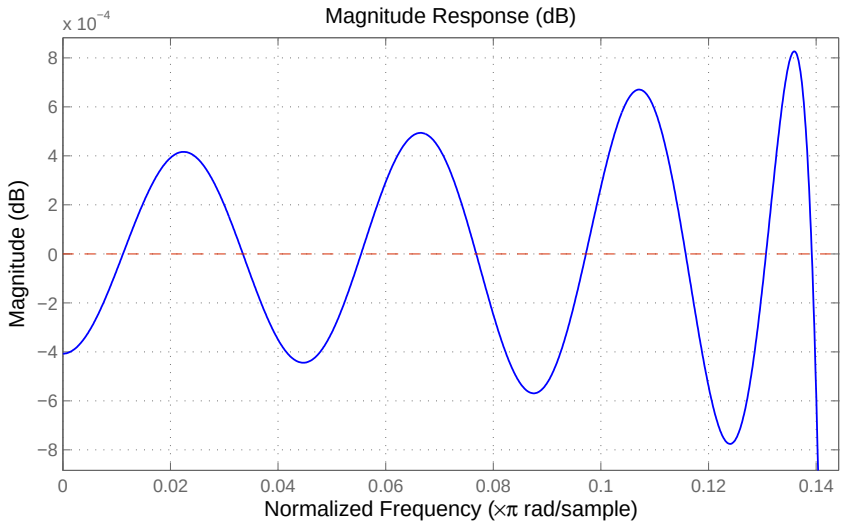


Figure 3.1: *Passband ripple details for* 5*th-band Nyquist FIR filter designed with a Kaiser window.*

for such branch. The impulse response for the filter designed in Exampled 29 can be seen in Figure 3.2. Note that every 5th sample is equal to zero except for the middle sample which is the peak value of the impulse response.

3.1.1 Equiripple Nyquist filters

If instead of specifying the order and stopband attenuation of an FIR Nyquist filter design we specify the order and transition width or the transition width and the stopband attenuation, equiripple designs are possible. It must be pointed out however that it is not trivial to design equiripple FIR Nyquist filters given the time-domain constraint (the fact that the impulse response must be exactly zero every L samples). Equiripple designs may have convergence issues for large order/small transition width combinations and may result in bogus filters. Filters with equiripple passband and sloped stopband are also possible*

Although equiripple designs are possible, their advantages relative to

* Except in the halfband (L=2) case. In that case, symmetry characteristics of the magnitude response about the 0.5π point result in same passband/stopband ripples. Therefore a sloped stopband will also mean a sloped passband.

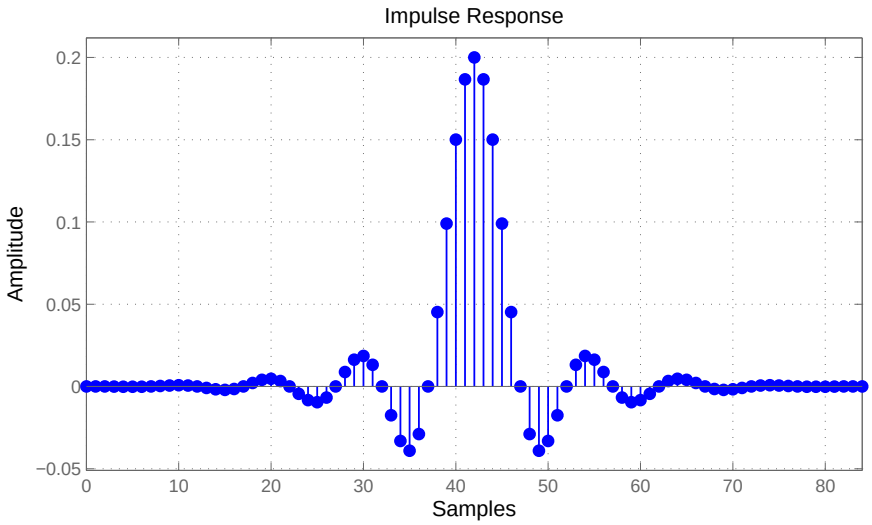


Figure 3.2: *Impulse response of* 5*th-band Nyquist FIR filter. Every 5th sample is equal to zero (except at the middle).*

Kaiser-window designs are not as clear for Nyquist filters as they are for regular filter designs. In particular, the passband ripples of Kaiser-window designs may be smaller than those of equiripple designs. Moreover, as we will see in Chapter 4, the increasing attenuation in the stopband may be desirable for interpolation and especially for decimation purposes.

Example 30 *We will compare a Kaiser window design to both an equiripple design with and without a sloped stopband.*

```
f  = fdesign.nyquist(4,'N,TW',72,.1); % 4th-Band Nyquist filter
h  = design(f,'kaiserwin');
h2 = design(f,'equiripple');
h3 = design(f,'equiripple','StopbandShape','linear','StopbandDecay',20);
```

While it is clear that the Kaiser-window design has the smallest minimum stopband attenuation, it does provide increased stopband attenuation as frequency increases. The magnitude responses are shown in Figure 3.3.

Moreover, the Kaiser window design results in better passband ripple performance than either equiripple design (this can be verified with the `measure` *command). The passband ripple details are shown in Figure 3.4.*

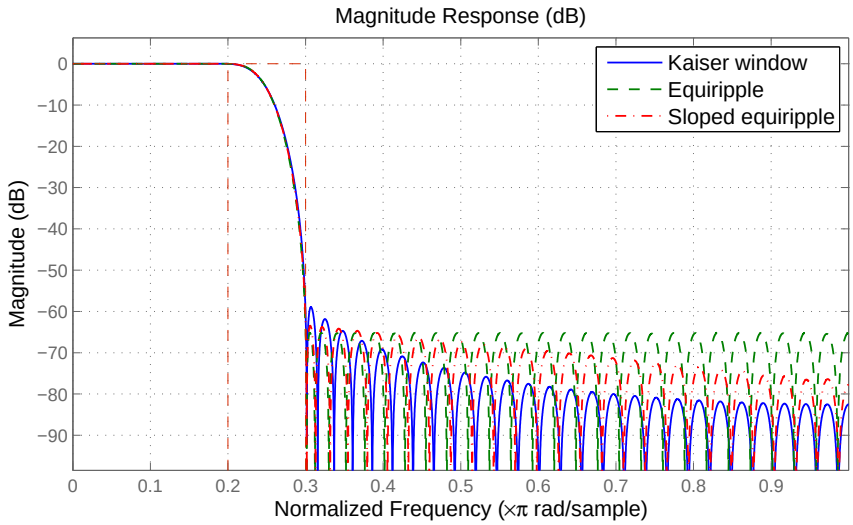


Figure 3.3: *Comparison of Kaiser window, equiripple, and sloped equiripple Nyquist filter designs of 72nd order and a transition width of* 0.1π.

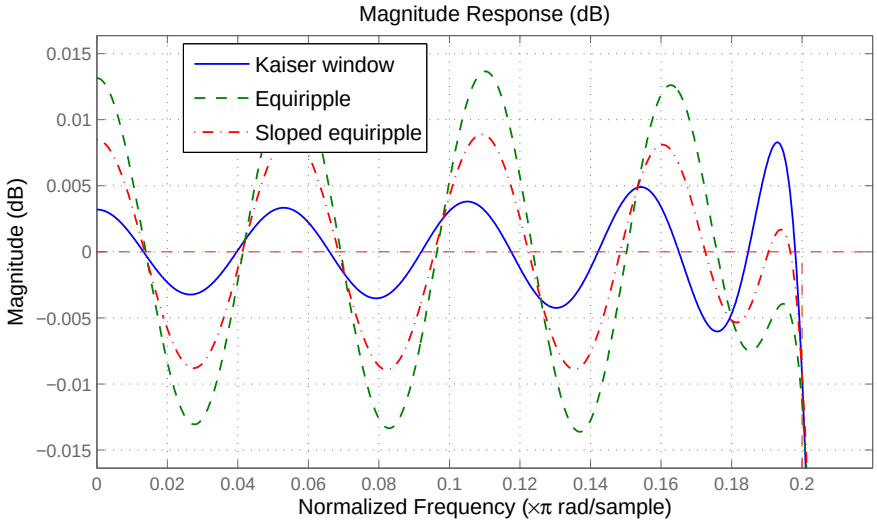


Figure 3.4: *Passband ripple details for filters designed in Example 30.*

3.1.2 Minimum-order Nyquist filters

It is possible to design either equiripple or Kaiser-window FIR filters that meet a transition-band requirement as well as a minimum-stopband requirement with minimum order. It is important to note that while the minimum stopband attenuation is the same for both designs. the maximum peak-to-peak passband ripple of each design will be different (of course so to will be the resulting filter order).

What is interesting is that while equiripple designs tend to be of smaller order, the peak-to-peak passband ripple of Kaiser window designs tends to be smaller. This coupled with the increased *overall* attenuation in the stopband due to its non-equiripple nature, may make Kaiser window designs appealing despite the slightly larger implementation cost.

Example 31 *Consider the following design examples, compare the results of the* `measure` *command and the* `cost` *command for both Kaiser window designs and equiripple designs.*

```
f   = fdesign.nyquist(8,'TW,Ast',0.1,60);
f2  = fdesign.nyquist(4,'TW,Ast',0.2,50);
hk  = design(f,'kaiserwin');
he  = design(f,'equiripple');
hk2 = design(f2,'kaiserwin');
he2 = design(f2,'equiripple');
```

Both cases illustrate the trade-off between filter order and passband ripple for Nyquist filter design choices.

3.2 Halfband filters

Halfband filter are a special case of Nyquist filters when $L = 2$. This filters reduce the bandwidth of a signal roughly by half and are suitable for decimation/interpolation by a factor of 2 (see Chapter 4).

As previously stated, Nyquist filters are characterized by the fact that every Lth sample of its impulse response (i.e. every Lth filter coefficient) is equal to zero. In the case of halfband filters this fact is particularly appealing since it means that roughly half of its coefficients are equal to zero. This of course makes them very efficient to implement. As such, halfband

filters are widely used in particular for multirate applications including multistage applications (see Chapter 5).

The cutoff frequency for a halfband filter is always 0.5π. Moreover, the passband and stopband ripples are identical, limiting the degrees of freedom in the design. The specifications set follow the usual triangle metaphor shown in Figure 1.2, taking into account the limitations just described. As such, three different specifications are available for halfband designs. In each case, two of the three angles of the triangle are specified (as just stated, the stopband ripple automatically determines the passband ripple).

```
f  = fdesign.halfband('N,TW',N,TW);
f2 = fdesign.halfband('N,Ast',N,Ast);
f3 = fdesign.halfband('TW,Ast',TW,Ast);
```

In all three cases, either Kaiser window or equiripple FIR designs are possible. Moreover, in the first case, it is also possible to design least-squares FIR filters.

Unlike the general case, there are usually no convergence issues with equiripple halfband filter designs. In addition to that, the fact the passband ripple is the same as the stopband ripple means that regardless of the design algorithm, the resulting peak-to-peak passband ripples will be about the same.

So for halfband filters, the advantages of optimal equiripple designs over Kaiser window designs resurface in a more clear manner than for other Nyquist filters. For a given set of specifications, the equiripple designs will have either a larger minimum stopband attenuation, a smaller transition width, or a smaller number of coefficients.

Example 32 *Consider Kaiser window and equiripple designs for the following to cases:*

```
f  = fdesign.halfband('N,TW',50,0.1);
f2 = fdesign.halfband('TW,Ast',0.1,75);
```

Using the `measure` *command in the first case shows that the resulting stopband attenuation (and consequently the passband ripple) is better in the equiripple case. Similarly, using the* `cost` *command in the second case shows that the same performance is achieved with fewer coefficients in the equiripple case.*

Sloped equiripple designs are also possible, but in the case of halfband filters, they will result in similarly sloped passbands, given the symmetry constraints on halfband filters.

3.2.1 IIR halfband filters

Nyquist filters can also be designed as IIR filters. The most interesting case is that of halfband IIR filters [20]-[21]. These filters are extremely efficient especially when implemented as two allpass branches in parallel. For IIR halfband filters, the parallel allpass decomposition (8.1) becomes

$$H(z) = \frac{1}{2}\left(\hat{A}_0(z^2) + z^{-1}\hat{A}_1(z^2)\right) \tag{3.1}$$

with $A_0(z) = \hat{A}_0(z^2)$ and $A_1(z) = z^{-1}\hat{A}_1(z^2)$.

Such implementations are ideally suited for polyphase multirate architectures (see Chapter 4). In Chapter 5 we will show that using IIR halfband-based multirate/multistage designs result in extremely efficient filters that outperform FIR counterparts.

IIR halfband filters have similar symmetry constrains as FIR halfbands. However, in the case of IIR filters, the symmetry is about the half-power point (3 dB point) rather than the half-magnitude point (6 dB point) of FIR halfbands. Because of this symmetry, it is not possible to design halfband Chebyshev filters. Only Butterworth and elliptic halfband filters are possible from the classical designs. A design algorithm synthesized directly in the digital domain [22] results in approximately linear phase in the passband of the filter (at the expense of either higher order or reduced stopband attenuation compared to elliptic halfbands).

Unlike the FIR case, it is important to point out that IIR halfbands are *not* necessarily Nyquist filters. In particular, Butterworth and elliptic designs result in non-Nyquist halfbands. As a result of this, although they can be used for polyphase decimation/interpolation, in the interpolation case, it will not be true that the input samples are preserved unchanged at the output.

The reason some IIR halfbands are not Nyquist, is that $\hat{A}_0(z^2)$ and $\hat{A}_1(z^2)$ may not be equal to a delay. However, in the case of approximately linear phase designs, $\hat{A}_0(z^2) = z^{-D}$ for some integer D. Therefore, the filter is Nyquist and when used for interpolation, the input samples are preserved unchaged (just delayed).

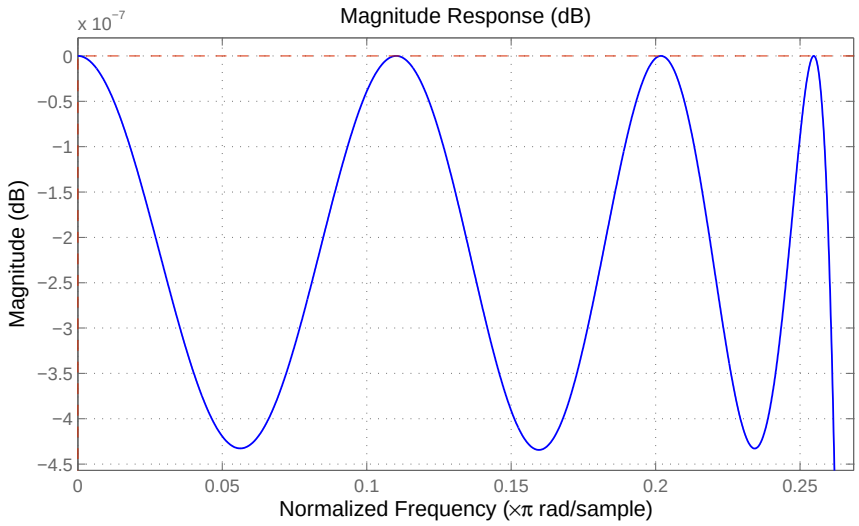


Figure 3.5: *Passband details of quasi-linear phase IIR halfband filter with 70 dB of stopband attenuation.*

In all cases, given only moderate stopband attenuations, the resulting passband ripples of IIR halfband filters is very small. This fact coupled with the fact that (as previously stated) many applications do not require hundreds of decibels of stopband attenuation, limits the appeal of Butterworth halfband filters.

Example 33 *Consider the following quasi-linear phase IIR halfband design. The fact the we specify a 70 dB minimum stopband attenuation results in a peak-to-peak passband ripple of only* 4×10^{-7} *dB!*

```
f = fdesign.halfband('N,Ast',12,70);
h = design(f,'iirlinphase');
```

The passband details are shown in Figure 3.5.

Minimum-order designs

Minimum-order designs provide a good framework to compare implementation cost of FIR halfband filters vs. IIR halfbands. Elliptic halfbands are the most efficient while quasi-linear phase halfbands give up some efficiency in the name of phase linearity. Either case is significantly more efficient than FIR equiripple halfband.

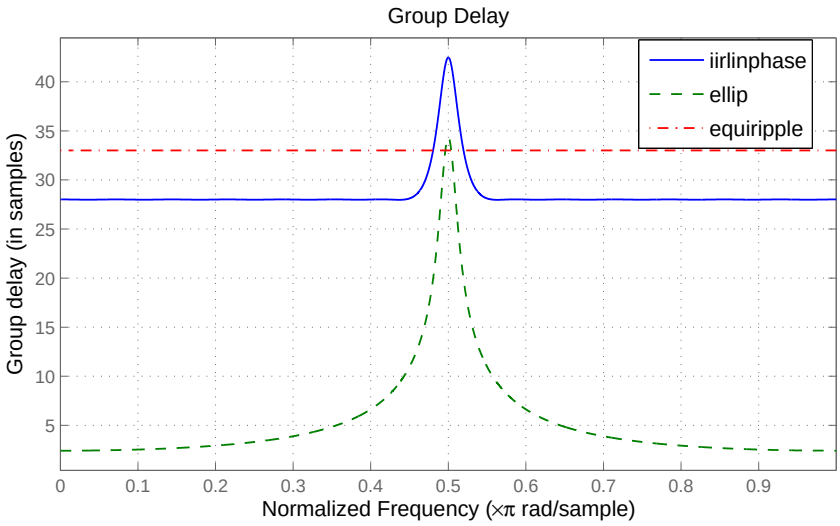


Figure 3.6: *Group-delay comparison for IIR and FIR halfband filters.*

Example 34 *Compare the following three minimum-order designs,*

```
f = fdesign.halfband('TW,Ast',0.1,60);
hlp = design(f,'iirlinphase');
hel = design(f,'ellip');
heq = design(f,'equiripple');
```

Using the `cost` *command, we can see that the FIR equiripple design requires 35 multipliers, the quasi-linear phase IIR design requires 15 multipliers, while the elliptic design requires only 6 multipliers.*

A plot of the group-delay of all three filters shows that while the elliptic design clearly has the lowest group delay, it is also the one with the most nonlinear phase. The passband group-delay of the quasi-linear phase design is almost flat as expected (see Figure 3.6).

The passband ripple of the equiripple design is quite small, yet it is orders of magnitude larger than the passband ripple of either IIR design.

3.3 Summary and look ahead

Nyquist filters have very useful properties that make them worth considering. They tend to result in efficient designs because of the fact that every

lth sample of their impulse response is zero. They also tend to have very small passband ripples.

Halfband filters are particularly interesting given their high efficiency. IIR halfband filters are even more efficient and have extremely small passband ripples.

Given all their advantages, Nyquist filters (FIR or IIR) should be the first choice for multirate applications (see Chapter 4). The fact that their cutoff frequency is given by π/L results in transition-band overlap in decimation applications. This however is not a problem given that no aliasing will occur in the band of interest ((see Chapter 4 and Appendix B.)

Moreover, we will see (Chapter 5) that cascade multirate Nyquist filters possess an interesting property: the overall equivalent multirate filter is also Nyquist. For example, a decimation/interpolation filter with a rate change of say 8 can be implemented as a cascade of three halfband decimation/interpolation filters (each with a rate change of 2). The three halfband filters in cascade act a single Nyquist filter (the equivalent impulse response of the cascade will have a value of zero every 8th sample) but can be significantly more efficient than a single-stage design. This is particularly true if IIR halfband designs are used.

Chapter 4

Multirate Filter Design

Overview

Multirate signal processing is a key tool to use in many signal processing implementations. The main reason to use multirate signal processing is efficiency. Digital filters are a key part of multirate signal processing as they are an enabling component for changing the sampling rate of a signal.

If multirate concepts are not fresh, it may be helpful to read through Appendix B prior to reading through this chapter.

In this chapter we will talk about designing filters for multirate systems. Whether we are increasing or decreasing the sampling rate of a signal, a filter is usually required. The most common filter type used for multirate applications is a lowpass filter.* Nyquist filters, described in Chapter 3, are the preferred design to use for multirate applications.

We start by presenting filters in the context of reducing the sampling rate of a signal (decimation). We want to emphasize the following: if you reduce the bandwidth of a signal through filtering, you should reduce its sampling rate accordingly.

We will see that the above statement applies whether the bandwidth is decreased by an integer or a fractional factor. Understanding fractional sampling-rate reduction requires understanding interpolation. We present first interpolation as simply finding samples lying between existing samples (not necessarily increasing the sampling rate). We then use this to

* However, as we will see, highpass, bandpass and in general any filter that reduces the bandwidth of a signal may be suitable for multirate applications.

show how to increase the sampling-rate of a signal by an integer factor. Finally we extend this paradigm in order to perform fractional decimation/interpolation.

4.1 Reducing the sampling rate of a signal

In the previous chapters we have discussed the basics of FIR and IIR filter design. We have concentrated mostly on lowpass filters. A lowpass filter reduces the bandwidth of the signal being filtered by some factor. When the bandwidth of a signal is reduced, we usually want to reduce the sampling rate in a commensurate manner. Otherwise, we are left with redundant data, given that we are over-satisfying the Nyquist sampling criterion. Any processing, storage, or transmission of such redundant data will result in unnecessary use of resources.

Specifically, if the input signal $x[n]$ has a bandwidth B_x and the filtered signal $y[n]$ has a bandwidth B_y related to the input bandwidth by

$$\frac{B_x}{B_y} = M$$

we should reduce the sampling frequency of $y[n]$ by a factor of M.

4.1.1 Decimating by an integer factor

If M is an integer the procedure is straightforward. We can downsample $y[n]$ by keeping one of every M samples

$$y_{down}[m] = y[nM].$$

The idea is depicted in Figure 4.1 and is referred to as *decimation*. The filter $H(z)$ is a lowpass filter that roughly reduces the bandwidth by a factor of M. Once the bandwidth is reduced, the sampling-rate should be reduced accordingly. The downsampler that follows $H(z)$ reduces the sampling rate by keeping one out of every M of its input samples.

However, the procedure as described so far is inefficient in that after the filter processes a set of M samples, only one sample kept. All the computation involved in obtaining the other $M-1$ samples is wasted.

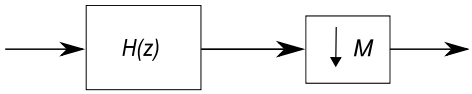


Figure 4.1: *Reducing the sampling rate after the bandwidth of the signal is reduced.*

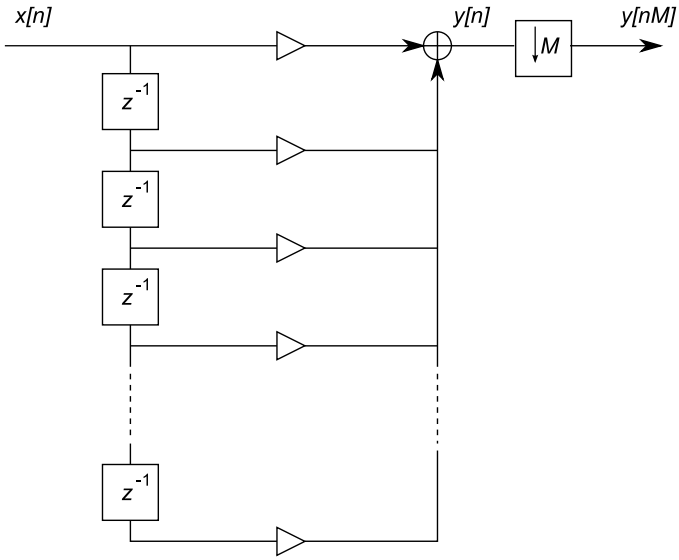


Figure 4.2: *FIR filter implemented in direct-form followed by a downsampler.*

Efficient FIR filters stemming from decimation

For FIR filters, a better paradigm is easy to implement in which we never actually compute any of the samples that we would throw away anyway. To illustrate the idea consider an FIR filter implemented in direct-form followed by a downsampler by M as shown in Figure 4.2. There is no point computing all additions/multiplications indicated only to throw out the result of most of them. Instead, we can downsample before the multipliers so that only computations that are required are performed [3], [23], [24]. The idea is shown in Figure 4.3.

For any filter of length $N+1$ (order N), even though the filter still requires $N+1$ multipliers, they are only used once every M samples. In other words, the average number of multiplications per input sample (MPIS) is $\frac{N+1}{M}$. That is of course a savings of a factor of M on average.

Unfortunately, for IIR filters, it is not straightforward to implement the

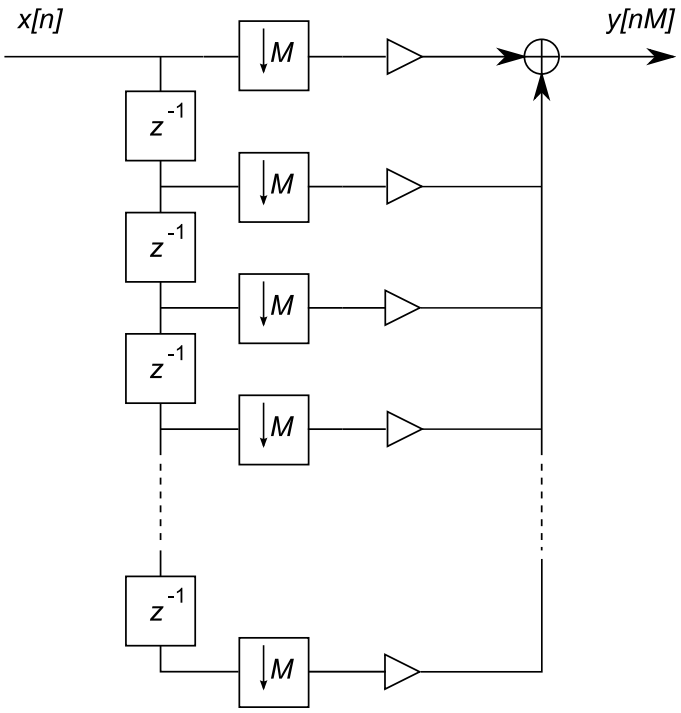


Figure 4.3: *Efficient FIR decimation.*

procedure shown due to the feedback. There is a way to implement decimation efficiently with certain IIR designs using an allpass-based decomposition as described in §8.1.2. We will discuss these efficient multirate IIR filters later.

Determining the decimation factor So far we have described a filter that reduces the bandwidth by a factor M as if it were an ideal filter. Of course, with practical filters we have to worry about the finite amount of attenuation in the stopband, the distortion in the passband, and the non-zero width transition band between passband and stopband. We'd like to explore how does using a non-ideal filter come into play when we decimate.

In order to answer this, consider what happens to the output signal once it is downsampled by M (see also Appendix B). In the same way that sampling an analog signal introduces spectral replicas of the analog signal's spectrum, downsampling a digital signal introduces replicas of its spectrum. In the case of downsampling, the number of replicas introduced

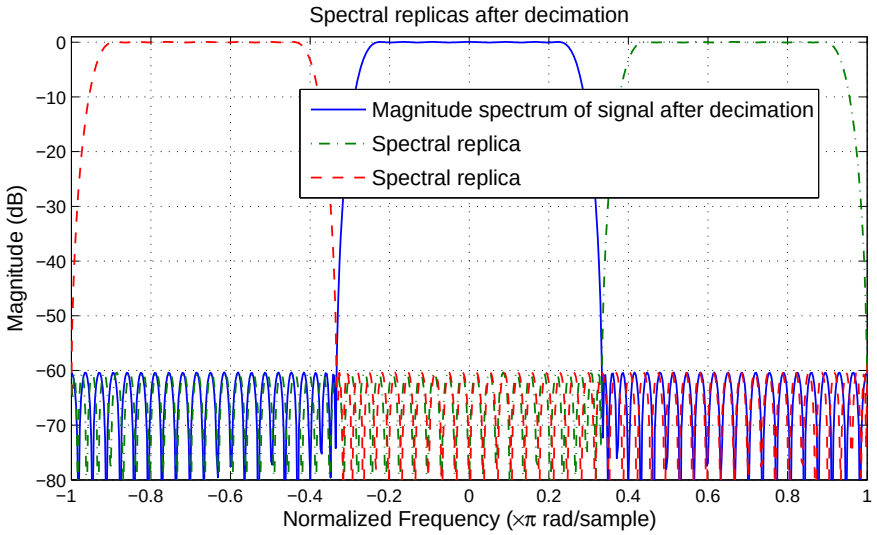


Figure 4.4: *Spectrum of decimated signal with $\omega_{st} = \pi/M$ along with spectral replicas.*

is equal to $M - 1$.

We'd like to compare two different cases, one in which the cutoff frequency is π/M, and another in which the stopband-edge frequency is π/M. First let's look at the latter case.

For illustration purposes, assume $M = 3$ and assume that the spectrum of the signal to be filtered is flat and occupies the entire Nyquist interval. This means that the spectrum of the output signal (prior to downsampling) will take the shape of the filter. The conclusions we will reach are valid for any value of M and any input signal (but the aliasing will be lower than what we show if the input signal is already somewhat attenuated in the stopband; we are showing the worst-case scenario). The spectrum of the decimated signal along with the replicas are shown in Figure 4.4. Aliasing invariably occurs because of the finite amount of stopband attenuation. However, presumably we select the amount of stopband attenuation so that the aliasing is tolerable. Notice the problem with using filters with equiripple stopband when decimating. The energy in the stopband is large and it all aliases into the spectrum of the signal we are interested in. Also notice that the usable bandwidth extends from zero to the passband-edge frequency. The passband-edge frequency in this case is given by $\pi/M - T_w$, where T_w is the transition-width.

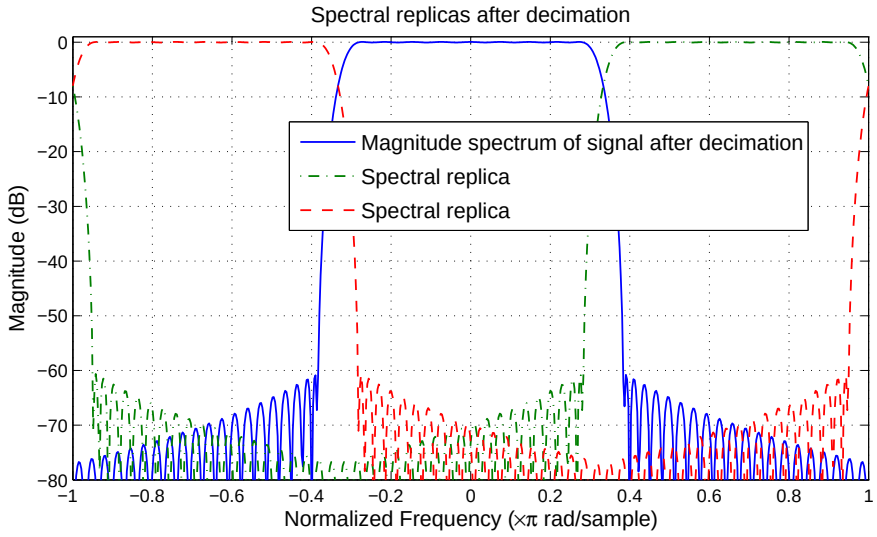


Figure 4.5: *Spectrum of decimated signal with $\omega_c = \pi/M$ along with spectral replicas.*

Now let's look at the case where the cutoff frequency is set to π/M. Recall that for FIR filters the cutoff frequency is typically located at the center of the transition band. The spectrum of the decimated signal along with its corresponding replicas for this case are shown in Figure 4.5. Since the cutoff frequency is in the middle of the transition band, the transition bands alias into each other. This is not a problem since that region is distorted by the filter anyway so that it should not contain any information of interest. Notice that we have used a sloped stopband to reduce the total amount of aliasing that occurs. Of course this could have been done in the previous case as well. Also, the usable passband extends now to $\pi/M - T_w/2$ where once again T_w is the transition width. This means that the usable passband is larger in this case than in the case $\omega_{st} = \pi/M$ by $T_w/2$. In order to have the same usable passband in both cases, we would have to make transition width of the case $\omega_{st} = \pi/M$ half as wide as the transition width of the case $\omega_c = \pi/M$. Given that for FIR filters the filter order grows inversely proportional to the transition width, this would imply having to increase the filter order by a factor of about two in order to obtain the same usable bandwidth with $\omega_{st} = \pi/M$.

While both selections, $\omega_{st} = \pi/M$, or $\omega_c = \pi/M$ are valid. There are clear advantages to using $\omega_c = \pi/M$. In practice, we usually have some

amount of excess bandwidth in a sampled signal so that the aliasing that occurs in the transition band does not affect the application. For instance, consider an audio signal. The most common sampling frequency is 44.1 kHz, but the maximum frequency we are interested in is 20 kHz. That means there is 4.1/2 kHz available at each side of the spectrum for transition bands of filters. See [11] for more on this.

All this to say that usually the decimation factor M should still be selected as the ratio ω_c/π even when the non-ideal nature of practical FIR filters is taken into consideration. This fits in well with the use of Nyquist filters. Since an Mth-band Nyquist filter has a cutoff frequency of π/M, it is well-suited for applications in which we want to reduce the sampling rate by a factor of M.

Example 35 *A 4th-band Nyquist filter reduces the bandwidth by a factor of 4. We can simultaneously reduce the sampling rate by a factor of 4 by designing a decimator Nyquist filter,*

```
f = fdesign.decimator(4,'Nyquist',4,'TW,Ast',0.1,60);
h = design(f,'kaiserwin');
```

Computational savings with FIR decimators We'd like to return to the comparison between an FIR and an IIR filter that meet the same set of specifications. In §2.2.6 we saw that for the specification set 4 an elliptic IIR filter requires 20 multipliers and 20 MPIS to implement assuming a direct-form II second-order sections implementation. An optimal equiripple filter requires 642 multipliers if we do not take advantage of the symmetry in the filter coefficients. However, given that the cutoff frequency is $\pi/16$, we can use the efficient decimation techniques described above to implement the FIR filter with an average of 40.125 MPIS even without taking advantage of the symmetric coefficients.*

```
M   = 16; % Decimation factor
Fc  = 0.0625;
TW  = 0.008;
Fp  = Fc-TW/2;
Fst = Fc+TW/2;
```

* An efficient decimation structure that does take advantage of the symmetry in the coefficients is shown on page 77 of [24].

```
Ap  = 1;
Ast = 80;
Hf  = fdesign.decimator(M,'lowpass','Fp,Fst,Ap,Ast',...
      Fp,Fst,Ap,Ast);
Hd  = design(Hf,'equiripple');
cost(Hd)
```

Through the use of multirate techniques, we are within a factor of two of the number of MPIS required to implement an IIR filter that meets the specifications. However, we still have perfect linear phase and no stability issues to worry about.

For the case just discussed, a Kaiser-window-designed* Nyquist filter would require more computations per input sample than a regular lowpass filter. The main reason is the passband ripple. With the given specifications, the Kaiser-window design will have a mere 0.0015986 dB of passband ripple. The cost to pay for this vast over-satisfying of the specifications is a computational cost of 73.6875 MPIS.

In Appendix C we study various approaches to filter design for these same specifications. We will see that multistage/multirate designs and in particular multistage Nyquist multirate designs will be the most efficient of all approaches looked at for these specifications. This is true despite the fact that such multistage Nyquist designs also will vastly over-satisfy the passband ripple requirements.

Decimating by a factor of 2

When reducing the bandwidth/decimating by a factor of two, if the cutoff frequency is set to $\pi/2$ (and transition-band aliasing is tolerated), FIR or IIR halfband filters can be used to very efficiently implement the filter.

In the FIR case, the halfband decimator by 2 takes full advantage of the fact that about half the filter coefficients are zero.

Example 36 *Consider the following design:*

```
f = fdesign.decimator(2,'halfband','TW,Ast',0.1,80);
h = design(f,'equiripple');
cost(h)
```

* Equiripple designs do not converge for these specifications given the Nyquist constraint on the impulse response.

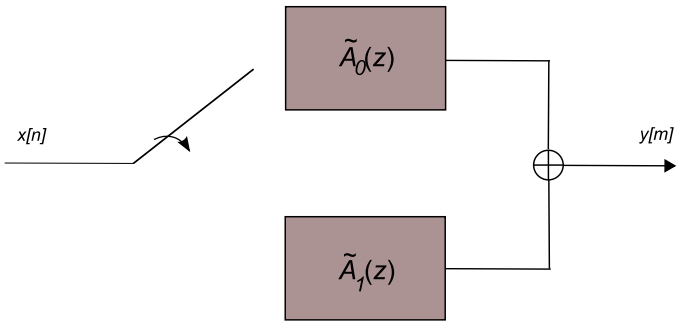


Figure 4.6: *Allpass-based IIR polyphase implementation of decimation by 2.*

The halfband is already efficient in that for a filter of length 95, it only requires 49 multipliers. By reducing the sampling rate, a further gain factor of two is obtained in computational cost. The number of MPIS is 24.5.

In the IIR case, efficient allpass-based polyphase implementations are possible for halfband designs. The idea is two make use of the fact that conceptually, decimation is lowpass filtering followed by downsampling. By using (3.1) and the Noble identities, it is simple to obtain the implementation shown in Figure 4.6. Note that use of the Noble identities means that the allpass filters are now given in terms of powers of z rather than of z^2. Let's look at the computational cost of using an IIR halfband filter for the specifications from the previous example.

Example 37 *For the same specifications contained in the variable* `f` *above, we can design either an elliptic halfband decimator or a quasi linear phase halfband decimator:*

```
h1 = design(f,'ellip');
h2 = design(f,'iirlinphase');
cost(h1)
cost(h2)
```

As usual, the elliptic design is the most efficient, requiring only 6 multipliers and 3 MPIS. If approximate linear phase is desired, the `iirlinphase` *design requires 19 multipliers and 9.5 MPIS.*

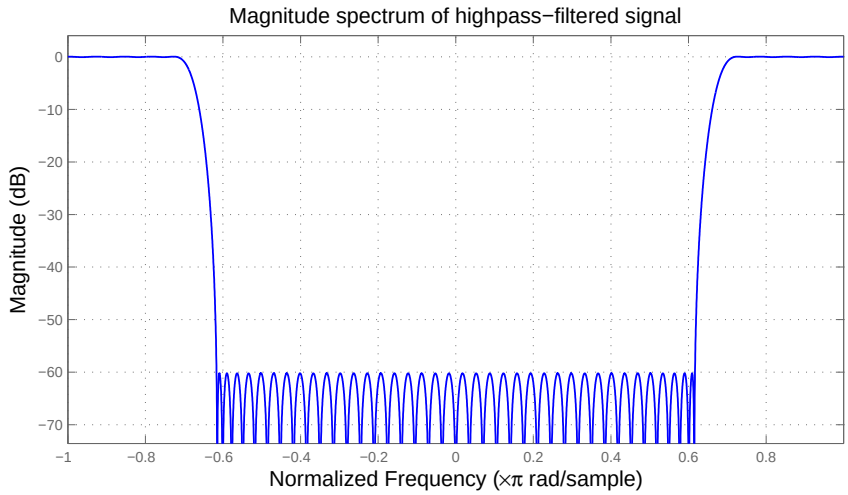


Figure 4.7: *Spectrum of highpass-filtered signal.*

Decimating with highpass/bandpass filters

The motivation to reduce the sampling rate whenever the bandwidth of a signal is reduced remains true regardless of the shape of the filter used to reduce the bandwidth. If we use a highpass or a bandpass filter to reduce the bandwidth by an integer factor M, we would still want to reduce the sampling frequency accordingly. The resulting signal after downsampling is called the *baseband equivalent* of a bandpass signal. Any further processing at baseband is less costly because we use fewer samples once the sampling frequency is reduced. The bandpass signal and its baseband equivalent contain the same information. We could for instance generate an analog bandpass signal that corresponds to the digital bandpass signal by starting from the baseband equivalent.

As long as the filters are FIR, we can continue to use the efficient implementation we showed since the direct-form structure can be used regardless of the response type of the filter. If a highpass halfband filter is used, efficient polyphase IIR designs are a possibility as well.

Example 38 *For example, consider using a highpass filter that reduces the bandwidth by a factor $M = 3$. If we do not downsample, and the input to the filter has a flat spectrum, the output signal will have the spectrum of the filter. Something like what is shown in Figure 4.7. If we downsample the filtered signal by a*

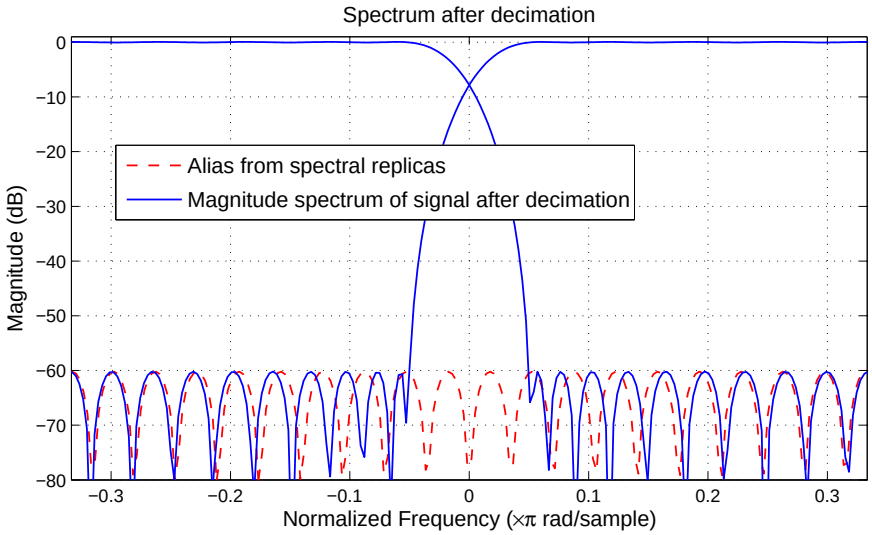


Figure 4.8: *Spectrum of downsampled highpass-filtered signal.*

factor of M, the resulting spectrum is moved to baseband as shown (along with the aliasing from replicas) in Figure 4.8. The spectrum is shown relative to the high sampling frequency, but we only show the portion of the Nyquist interval that corresponds to the downsampled signal. For this reason, the interval plotted is $[-\pi/3, \pi/3]$. *Notice that in this case, some of the aliasing comes from the replicas and some of it is "self-inflicted" so to speak. Nevertheless, as before, the amount of aliasing is controlled by the stopband attenuation. In this example, no single replica introduces more than -60 dB of aliasing in the usable passband.*

Notice that the previous example amounts to removing the "white space" that arises after highpass filtering. Since the energy in the signal is negligible in this white space, we best not waste resources by processing information corresponding to that band.

There are some restrictions as to where the bandpass that is retained lies relative to the sampling rate [24]. Even if the filter were ideal, its response needs to be such that it retains a bandwidth M times smaller than the Nyquist interval and lying in one of the intervals

$$k\pi/M < |\omega| < (k+1)\pi/M \quad k = 0, 1, \ldots, M-1.$$

The cases $k = 0$ and $k = M-1$ correspond to lowpass and highpass filters respectively. These cases are thus never a problem. However, if the

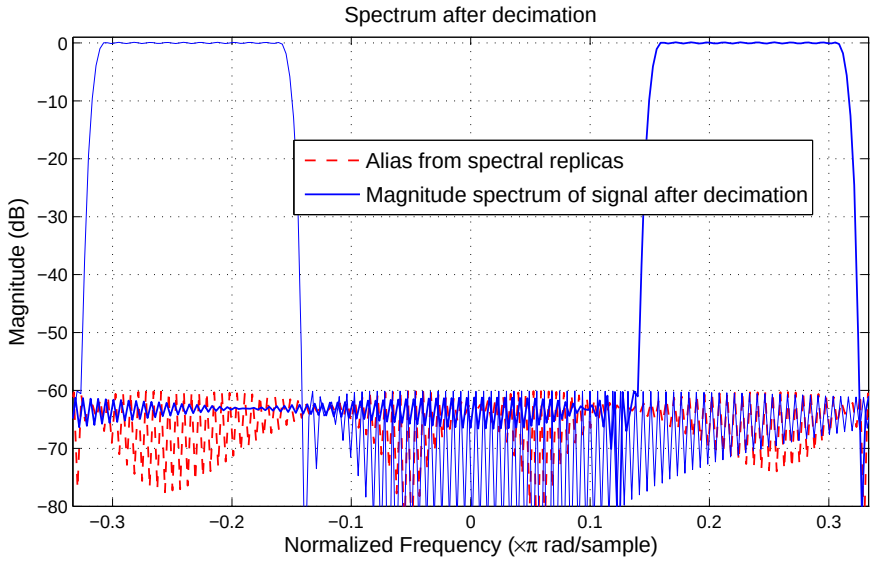


Figure 4.9: *Spectrum of bandpass-filtered signal downsampled by a factor $M = 3$.*

bandpass filter does not meet the restrictions just mentioned, we should still try to downsample as much as possible.

Example 39 *Suppose we are interested in retaining the band between 0.35π and 0.5167π. The bandwidth is reduced by a factor $M = 6$. However, the band-edges are not between $k\pi/M$ and $(k+1)\pi/M$ for $M = 4,5,6$. The band-edges do lie between $\pi/3$ and $2\pi/3$, so if we design a bandpass filter we can at least decimate by 3.*

```
TW  = 0.1/6; % Transition width
Fc1 = 0.35;
Fc2 = 0.5167;
M   = 3; % Decimation factor
Hf  = fdesign.decimator(M,'bandpass','Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2',...
      Fc1-TW/2,Fc1+TW/2,Fc2-TW/2,Fc2+TW/2,60,.2,60);
Hd  = design(Hf,'equiripple');
```

The spectrum of the bandpass-filtered signal after downsampling is shown in Figure 4.9. The frequency is relative to the high sampling rate prior to downsampling. Since we downsampled by $M = 3$, the new Nyquist interval after downsampling

will be $[-\pi/3, \pi/3]$. *Notice that we haven't been able to remove all the white space since we could only downsample by 3 even though the bandwidth was reduced by 6.*

Decimation when working with Hertz

So far we have used normalized frequency in all our decimation designs. If we'd like to specify the frequency specifications in Hertz, we should simply keep in mind that the sampling frequency that is used is the sampling frequency of the input signal (prior to downsampling).

Example 40 *Suppose we have a signal sampled at 30 kHz and we reduce its bandwidth by a factor of 3. The band of interest extends from 0 Hz to 4250 Hz. In order to remove redundancy, we reduce the sampling-rate by a factor of 3 as well. We use a 3rd-band Nyquist filter, therefore the cutoff frequency is equal to* $F_s/(2M) = 5000$ *Hz. The transition band is set to 1500 Hz so that it extends from 4250 Hz to 5750 Hz.*

```
M    = 3;
Band = 3;
Fs   = 30e3;
Hf   = fdesign.decimator(M,'Nyquist',Band,'TW,Ast',1500,65,Fs);
Hd   = design(Hf,'kaiserwin');
```

4.1.2 Decimating by a non-integer factor

If the bandwidth reduction factor is not an integer, we would still like to reduce the sampling frequency accordingly. A simple approximation would be to downsample the output by the largest integer that is not greater than the bandwidth reduction factor. This is less than optimal however and if the bandwidth reduction factor is less than two, there would be no efficiency improvements even if we reduce the bandwidth to almost (but not quite) half the original bandwidth.

A better solution would be to reduce the sampling rate by a non-integer factor as well. This implies computing some samples between existing samples as shown in Figure 4.10. Computing new samples in between existing samples amounts to interpolation. We will return to discuss fractional decimation in §4.4, but before we do so, we'll look at interpolation.

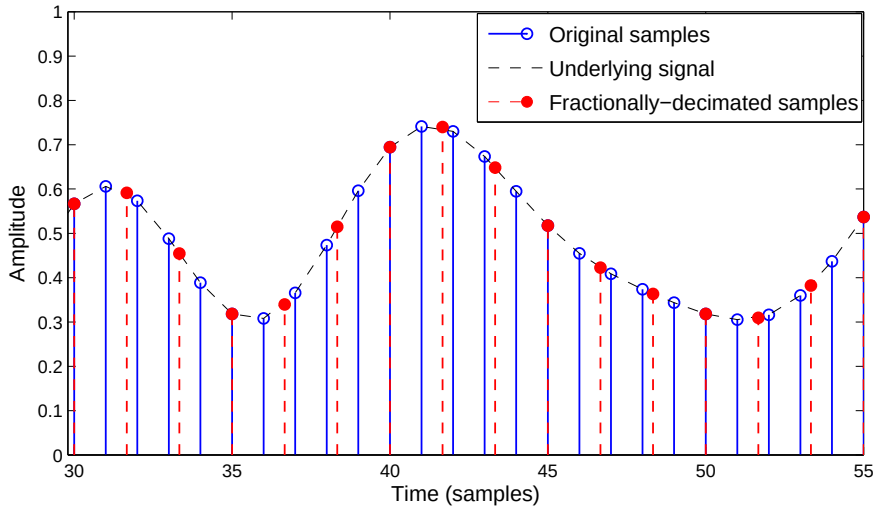


Figure 4.10: *Illustration of fractional decimation.*

4.2 Interpolation

Roughly speaking, interpolation consists of computing new samples between existing samples. In the context of signal processing, ideally we interpolate by finding the desired points on the underlying continuous-time analog signal that corresponds to the samples we have at hand. This is done without actually converting the signal back to continuous time. The process is referred to as sinc interpolation since an ideal lowpass filter (with a sinc-shaped impulse response) is involved (see Appendix B).

4.2.1 Fractionally advancing/delaying a signal

Usually interpolation is presented in the context of increasing the sampling-rate of a signal, but strictly speaking, this is not necessary. Consider the situation shown in Figure 4.11. We'd like to compute the interpolated samples from the existing samples. For each existing sample we compute a new sample, therefore the sampling rate remains the same.

The interpolated samples in Figure 4.11 are found by fractionally advancing the signal in time by a factor α. Equivalently, the samples can be thought to be obtained by fractionally delaying the signal by a factor

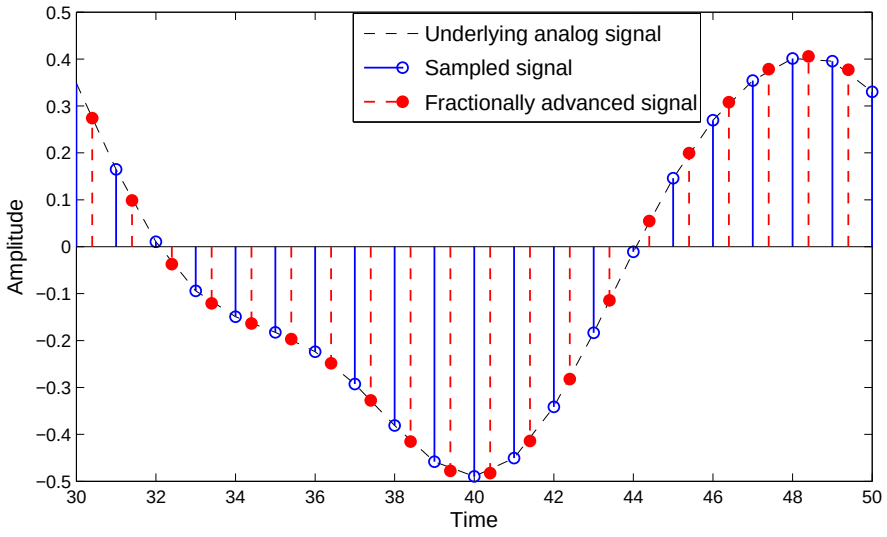


Figure 4.11: *Fractional advance by a factor* $\alpha = 0.4$.

$\beta = 1 - \alpha$. Both α and β are a fraction between zero and one. Thus, if $x[n]$ is our signal prior to interpolation, the signal consisting of interpolated values is computed by passing the signal through a filter with transfer function

$$H_{\text{frac}}(z) = z^{\alpha}.$$

Of course advancing a signal in time is a non-causal operation. Since the advance is less than one sample, we can make it causal by delaying everything by one sample,

$$H_{\text{causal}} = z^{-1}z^{\alpha} = z^{\alpha-1} = z^{-\beta}.$$

which shows the equivalence of a causal advance by a factor of α and a delay (notice the negative sign) by a factor of $\beta = 1 - \alpha$.

By taking the inverse Fourier transform, the impulse response of the fractional-advance filter z^{α} can be easily found to be

$$h_{\text{frac}}[n] = \frac{sin(\pi(n+\alpha))}{\pi(n+\alpha)}, \quad -\infty < n < \infty \tag{4.1}$$

Not surprisingly, the ideal fractional advance has an infinite impulse response that is also infinitely non-causal. We will look at various FIR and

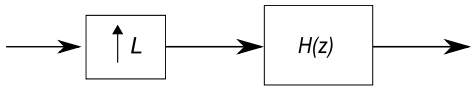


Figure 4.12: *Conceptual diagram for increasing the sampling rate.*

IIR approximations to this filter later, but for now we will look at how a bank of fractional advance filters can be used to increase the sampling-rate of a signal.

4.2.2 Increasing the sampling-rate of a signal

In Appendix B we have already looked at what increasing the sampling-rate of a signal means in the frequency domain. In short, increasing the sampling-rate is accomplished by lowpass filtering the signal with a digital filter operating at the same rate we wish to increase the sampling rate to*. The key idea is to *digitally* produce a signal that would be exactly the same as a signal we had obtained by sampling a band-limited continuous time signal at the higher sampling rate.

Conceptually, increasing the sampling rate is accomplished as depicted in Figure 4.12. The upsampler inserts $L-1$ zeros between existing samples in order to increase the sampling rate of the signal. This step in itself does not modify the spectral content at all. All it does, is to change the Nyquist interval to encompass a bandwidth that is now L times larger than before upsampling. Because of this upsampling, the new Nyquist interval now contains $L-1$ spectral replicas of the baseband spectrum. In order to interpolate between the original samples, replacing the zeros that have been inserted with actual interpolated samples, it is necessary to lowpass filter the upsampled signal in such a way that the $L-1$ unwanted spectral replicas are removed (see Appendix B). The filter $H(z)$ performs such lowpass filtering.

In practice, increasing the sampling rate is never performed as shown in Figure 4.12. The reason is that inserting all those zeros means the low-

* Whenever we increase the sampling rate, we do not lowpass filter in order to reduce bandwidth. On the contrary, we want to use all the information in the existing signal in order to compute the new samples. The reason we lowpass filter is to remove spectral replicas that are encompassed by the new Nyquist interval when the rate is increased. See Appendix B for more on this.

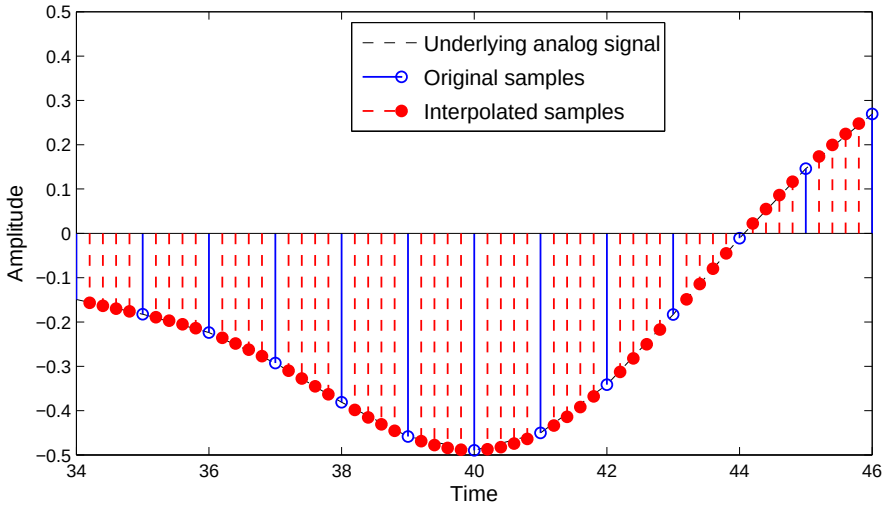


Figure 4.13: *Illustration of ideal band-limited interpolation in the time domain.*

pass filter would be performing a lot of unnecessary computations when its input is just a zero. Instead, a so-called polyphase approach is taken in order to efficiently increase the sampling rate. The polyphase approach is easiest to understand by looking at things in the time domain.

The situation in the time domain is depicted in Figure 4.13 assuming we want to increase the rate by a factor $L = 5$. We have seen that in order to produce each sample between existing samples, we need a fractional advance filter. Looking at Figure 4.13, we can see that we need to take every input sample $x_T[n]$ and produce 5 output samples $\{x_{T'}[m]\}$, $m = 5n + k$, $k = 0, \ldots, 4$ as follows (note that $T = 0.5$ and $T' = T/5 = 0.1$):

- $x_{T'}[5n] = x_T[n]$
- $x_{T'}[5n + 1] = x_T[n + \frac{1}{5}]$
- $x_{T'}[5n + 2] = x_T[n + \frac{2}{5}]$
- $x_{T'}[5n + 3] = x_T[n + \frac{3}{5}]$
- $x_{T'}[5n + 4] = x_T[n + \frac{4}{5}]$

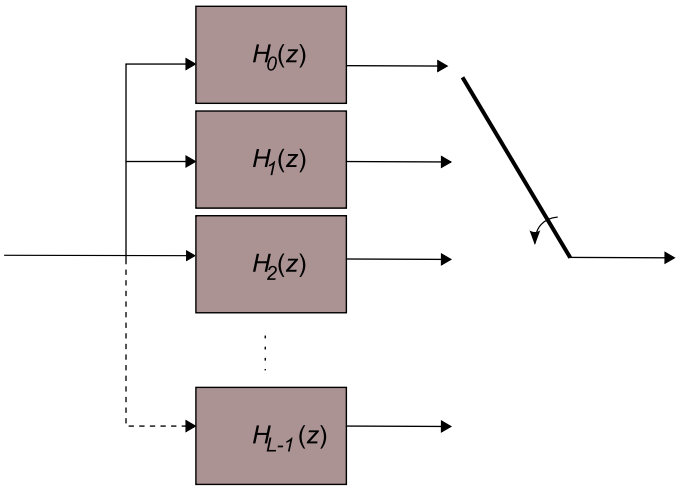


Figure 4.14: *Bank of filters used for interpolation. Each filter performs a fractional advance of the input signal and has a different phase. The overall implementation is referred to as a polyphase interpolation filter. For every input sample, we cycle through all polyphase branches.*

In general, the ideal interpolator consists of a bank of L filters, $H_k(z)$, $k = 0,\ldots,L-1$ which will fractionally advance the input signal by a factor k/L, $k = 0,\ldots,L-1$. The outputs of the filters are then interleaved (i.e. only one filter needs to operate per high rate output sample) to produce the interpolated signal. This is depicted in Figure 4.14

The L filters that comprise the filter bank are the fractional advance filters,

$$H_k(z) = z^{k/L}, \quad k = 0,\ldots,L-1.$$

Evaluating on the unit circle, we have

$$H_k(e^{j\omega}) = e^{j\omega k/L}, \quad k = 0,\ldots,L-1$$

so that each filter $H_k(e^{j\omega})$ is allpass, i.e. $\left|H_k(e^{j\omega})\right| = 1$ and has linear phase, $\arg\{H_k(e^{j\omega})\} = \omega k/L$. *

* The term polyphase stems from this derivation. Each filter in the filter bank has a different phase. The interpolator filter consists of L polyphase parallel branches, each branch tasked with computing one of the L interpolated outputs. The time-domain view of the ideal interpolation filter thus has the polyphase structure built-in.

Herein lies the impossibility of designing these filters in an exact manner. We cannot design them as FIR filters because no FIR filter can be allpass (except for a pure delay). We cannot design them as IIR filters, because no stable IIR filter can have linear phase. However, it is clear how we want to approximate the ideal interpolation filter bank.

FIR approximations will produce the exact linear phase, while approximating an allpass response as best possible. On the other hand, IIR approximations will be exactly allpass, while trying to produce the required phase.

It is enlightening* to realize that the filters comprising the filter bank are the polyphase components of the ideal interpolation filter derived in (1.2)!

Indeed, the impulse response of each fractional advance filter in the filter bank is given by the inverse DTFT,

$$h_k[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{j\omega k/L} e^{j\omega n} d\omega = \frac{\sin\left(\pi \frac{Ln+k}{L}\right)}{\pi \frac{Ln+k}{L}}$$

this corresponds to taking the transfer function corresponding to the ideal lowpass filter (1.1) with the cutoff frequency set to π/L and rewriting it in the form

$$\begin{aligned} H_{\mathrm{LP}}(z) = &\ldots + h[0] + z^{-L}h[L] + \ldots \\ &+ z^{-1}(\ldots + h[1] + z^{-L}h[L+1] + \ldots) \\ &\vdots \\ &+ z^{-(L-1)}(\ldots + h[L-1] + z^{-L}h[2L-1] + \ldots) \end{aligned}$$

which can be written as

$$H_{\mathrm{LP}}(z) = H_0(z^L) + z^{-1}H_1(z^L) + \ldots + z^{-(L-1)}H_{L-1}(z^L).$$

The sub-filters $H_k(z)$ are the polyphase components (each one is preceded by a different phase, z^{-k}) of the original transfer function. The impulse responses of each sub-filter correspond to the L decimated sequences of the ideal impulse response by again writing uniquely $m = Ln + k$, $k = 0, \ldots, L-1$ in (1.2).

* The reality is that this should be expected given that we had already determined that the ideal interpolator was an ideal lowpass filter. Yet it is nice to see that things fit in even though we have approach the derivation of the ideal interpolator from two different perspectives.

4.2.3 Design of FIR interpolation filters

While interpolation filters are simply lowpass filters that can be designed with the various techniques outlined previously, the polyphase filters that compose the ideal interpolation filter give some insight on things to be looking for when designing interpolation filters.

Consider an interpolation by a factor of L. The ideal L polyphase filters will have a group-delay given by

$$-\tfrac{k}{L}, \quad k = 0, \ldots, L-1$$

For simplicity, consider an FIR approximation to the ideal interpolation filter where the order is of the form $N = 2LM$. Then each polyphase filter will have order $N/L = 2M$.

Note that the ideal interpolation filter is infinitely non-causal. After finite length truncation, it is possible to make the approximation causal by delaying by half the filter order, $N/2$. However, because we will implement in efficient polyphase form, we can make each polyphase component causal by delaying it by M samples.

The delay will mean the introduction of a phase component in the response of each polyphase component. So that instead of approximating the ideal fractional advance $e^{j\omega k/L}$ the polyphase components will approximate $e^{j\omega(k/L-M)}$. The group-delay will consequently be of the form

$$-\frac{d\phi(\omega)}{d\omega} = -\frac{d\omega(k/L-M)}{d\omega} = M - k/L.$$

A problem that arises is that even though the FIR approximation to the ideal interpolation filter is symmetric and thus has linear phase, the polyphase components are not necessarily symmetric and thus will not necessarily have exact linear phase. However, for each non symmetric polyphase filter, there is a mirror image polyphase filter which will have the exact same magnitude response with a mirror image group-delay that will compensate any phase distortion.

Example 41 *We design a Nyquist filter that interpolates by a factor $L = 4$. The Nyquist filter operates at the high rate (four times the input rate). It is a 4th-band filter with a cutoff frequency given by $\pi/4$. This filter will remove 3 spectral replicas encompassed by the augmented Nyquist interval once the signal is upsampled. The filter is implement in efficient polyphase manner.*

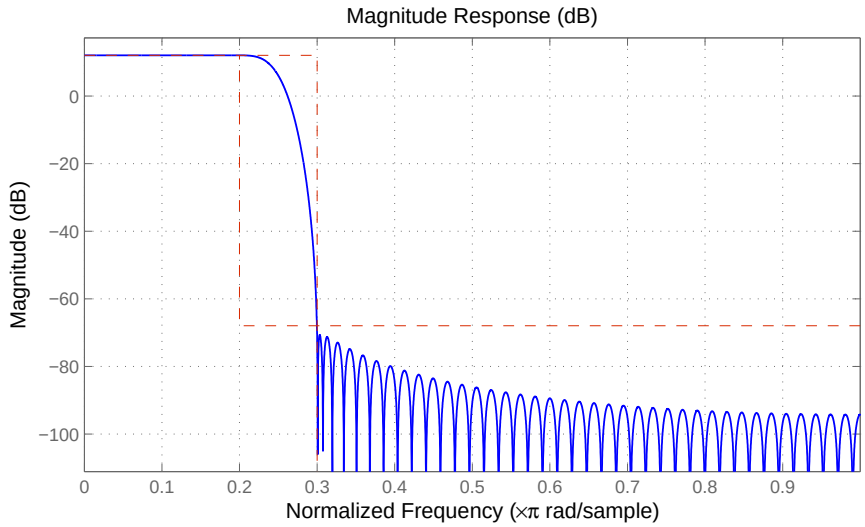


Figure 4.15: *FIR Nyquist interpolate by 4 filter designed with a Kaiser window. The filter removes 3 spectral replicas and operates at four times the input rate.*

```
L    = 4; % Interpolation factor
Band = 4; % Band for Nyquist design
Hf   = fdesign.interpolator(L,'Nyquist',Band,'TW,Ast',0.1,80);
Hint = design(Hf,'kaiserwin');
```

The resulting Nyquist filter is shown in Figure 4.15.

For the design from the previous example, it is worth taking a look at the magnitude response and group-delay of the polyphase components of the resulting FIR filter. The magnitude response is shown in Figure 4.16 while the group-delay is shown in Figure 4.17. The magnitude response reveals that only one of the polyphase filters is a perfect allpass, while the others are approximations that fade-off at high frequency. The magnitude responses of the 2nd and 4th polyphase sub-filters are identical. The group-delays show a fractional advance of 0, 0.25, 0.5, and 0.75 for the four polyphase sub-filters. This advance is relative to a nominal delay of $M = 13$ samples. The filter length is $2LM + 1 = 105$. Note that two of the polyphase components do not have perfectly flat group delays. However, the nonlinear shape of one compensates for the other so that overall the interpolation filter has linear phase.

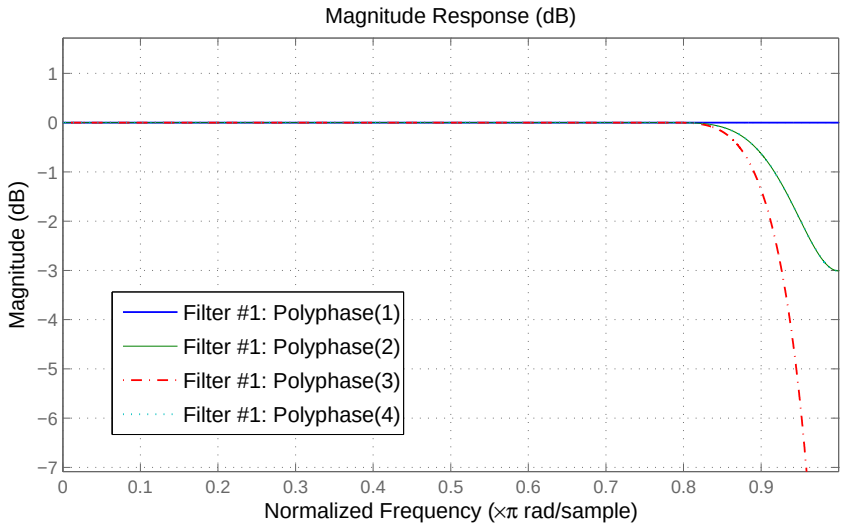


Figure 4.16: *Magnitude response of polyphase components for a Nyquist FIR interpolate-by-four filter.*

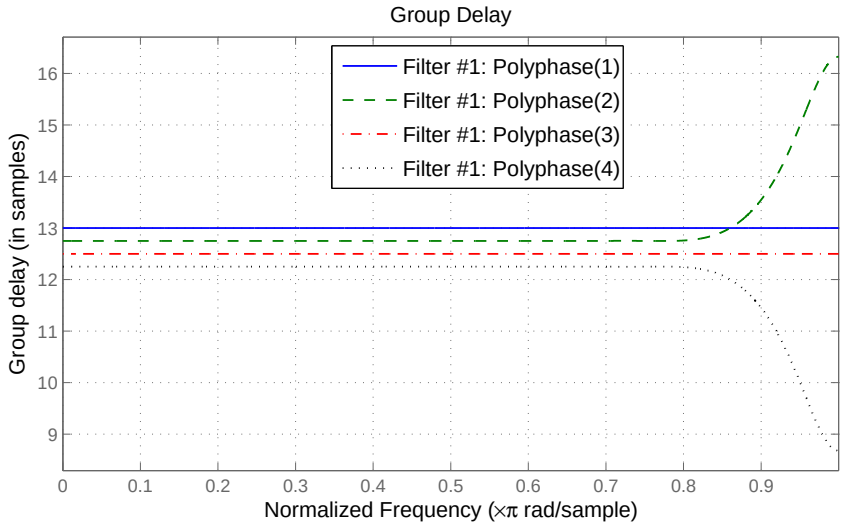


Figure 4.17: *Group-delay of polyphase components for a Nyquist FIR interpolate-by-four filter.*

Notice that the polyphase components can be obtained by taking every Lth sample of the lowpass impulse response as stated above. For example, it is easy to verify this for the second polyphase:

```
p = polyphase(Hint);
isequal(p(2,1:end-1),Hint.Numerator(2:4:end)) % Returns true
```

In general, for all Nyquist designs, one of the polyphase components will have perfectly flat magnitude response and group-delay. This polyphase component is a simple delay which provides a fractional advance of zero. The presence of this polyphase components ensures that the input samples are "passed undisturbed" to the output (they are simply delayed). In general, a non-ideal non-Nyquist lowpass filter will not have a perfect delay as one of its polyphase branches and therefore will alter somewhat the input samples. However, this is usually not a major issue in terms of distortion. Nevertheless, Nyquist filters are often used because of their reduced implementation cost given that one of the polyphase branches is simply a delay.

4.2.4 Design of IIR halfband interpolators

As with decimation, the use of halfband filters for interpolation is enticing given their efficiency. In the case of interpolation by a factor $L = 2$, FIR halfband filters (being Nyquist filters) will have one of their polyphase branches given simply by a delay.

IIR filters will not necessarily have one of their branches given by a simple delay because we do not use their impulse response directly to implement them. However, quasi-linear phase IIR filters are implemented in such a way that one of their polyphase branches is a pure delay. Nevertheless, elliptic halfband interpolation filters that meet the same specifications are still more efficient even though they do not have a simple delay as one of its polyphase branches (and therefore, as stated in Chapter 4, elliptic halfband filters are not Nyquist and as a result, they modify the input samples when interpolating).

The allpass-based IIR polyphase structure used to implement interpolation by two is shown in Figure 4.18. Note that because we have used the Noble identities, the transfer functions of the allpass filters in (3.1) are given as powers of z rather than powers of z^2.

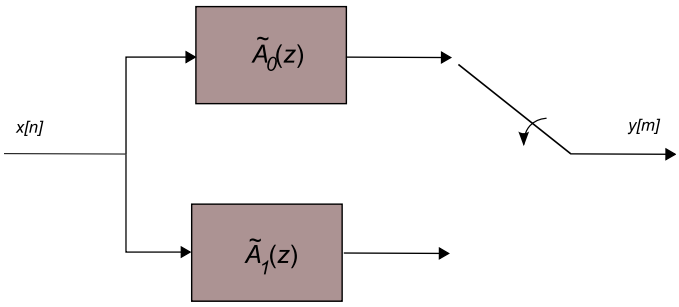


Figure 4.18: *Allpass-based IIR polyphase implementation of interpolation by 2.*

Example 42 *Design elliptic and quasi linear-phase IIR halfband interpolate-by-two filters for the given specifications.*

```
Hf     = fdesign.interpolator(2,'halfband','TW,Ast',0.08,55);
Hlin   = design(Hf,'iirlinphase');
Hellip = design(Hf,'ellip');
cost(Hlin)
cost(Hellip)
```

The two polyphase branches for either design are perfect allpass filters. The filters deviate from the ideal interpolation filter in their phase response. The group-delay of the polyphase branches for the quasi linear-phase IIR design is shown in Figure 4.19. Note that one of the branches is a pure delay and therefore has perfectly flat group-delay. The group-delay of the polyphase sub-filters for the elliptic design is shown in Figure 4.20. Note how neither of the polyphase components has a flat group-delay in this case.

4.2.5 Design of interpolators when working with Hertz

So far we have presented the design of interpolation filters in the context of normalized frequency. If working with Hertz, the key point to keep in mind is that when increasing the sampling rate we should not reduce the bandwidth of the signal. Once again, the interpolation filter is used not to reduce bandwidth, but to remove spectral replicas.

Example 43 *Suppose we have an audio signal sampled at 48 kHz and we want to increase the sampling frequency to 96 kHz. The band of interest extends to 20*

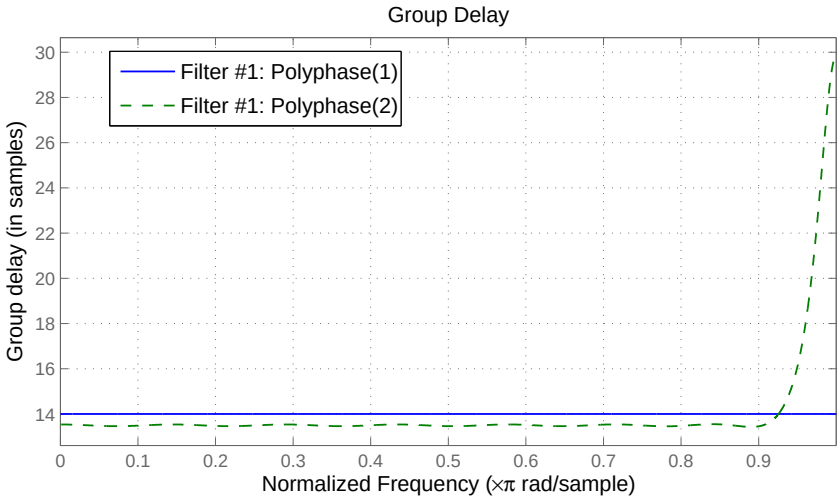


Figure 4.19: *Group-delay of polyphase components for a quasi linear-phase IIR halfband interpolate-by-two filter.*

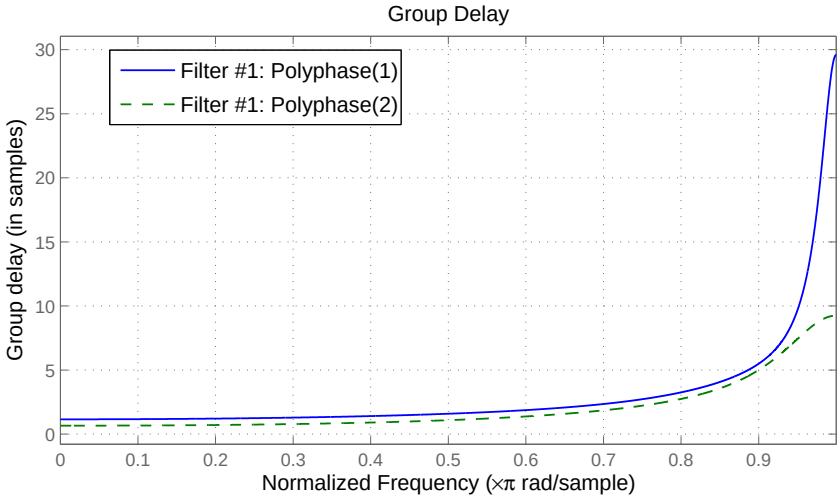


Figure 4.20: *Group-delay of polyphase components for an elliptic IIR halfband interpolate-by-two filter.*

kHz. Since we want to increase the sampling frequency by a factor of 2, we can use a (FIR or IIR) halfband filter,

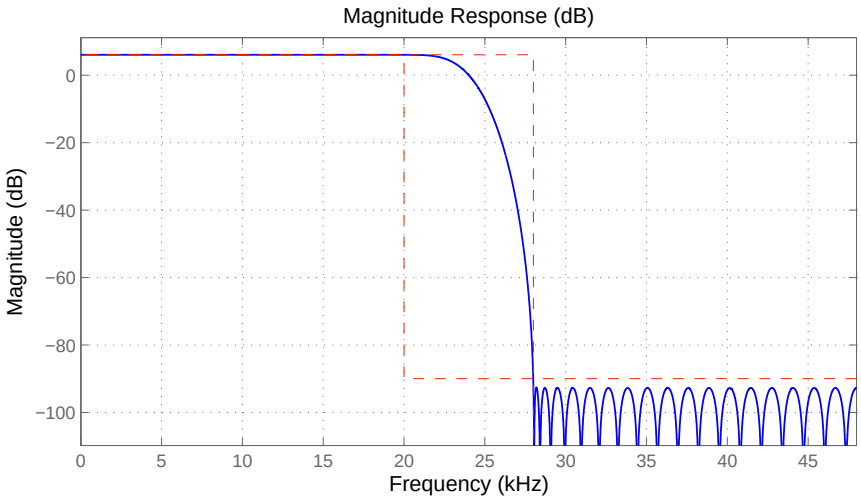


Figure 4.21: *Magnitude response of halfband filter used to increase the sampling rate by 2.*

```
f = fdesign.interpolator(2,'halfband','TW,Ast',8000,96,96000);
H = design(f,'equiripple');
```

The magnitude response of the filter can be seen in Figure 4.21. Note that the passband extends to 20 kHz as desired. The cutoff frequency for the filter is 24 kHz. The spectral replica of the original audio signal centered around 48 kHz will be removed by the halfband filter.

4.3 Increasing the sampling rate by a fractional factor

So far we have discussed increasing the sampling rate by an integer factor. In some cases, we want to increase the sampling rate by a fractional factor. This can be done trivially for rational factors.

Suppose we want to increase the rate by a factor of L/M with $L > M$. A trivial way of doing this is to perform a full interpolation by a factor of L and then discarding the samples we don't want, that is we keep one out of every M samples after interpolating.

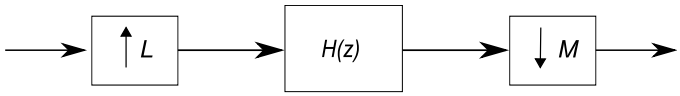


Figure 4.22: *Conceptual configuration for increasing the sampling rate of a signal by a fractional factor.* $L > M$.

Figure 4.22 illustrates the idea. The role of the upsampler+filter combination is the same as for the case when we increase the rate by an integer factor. Once we have increased the rate, we discard samples we don't need, keeping only the ones required for the new rate.

While the procedure we have just described is trivial, it is inefficient since many of the samples that have been computed via interpolation are subsequently discarded. Instead, to implement fractional interpolation efficiently we compute only the samples we intend to keep.

Specifically, if the polyphase filters are $H_0(z), H_1(z), \ldots, H_{L-1}(z)$, instead of using $H_0(z)$ to compute the first output, $H_1(z)$ to compute the second output and so forth, we use $H_0(z)$ for the first output, we skip $M-1$ polyphase filters and use $H_M(z)$ for the second output and so forth.

As an example, if $L = 3$ and $M = 2$, the sequence of polyphase filters used are $H_0(z)$, then skip $H_1(z)$, then use $H_2(z)$, then, for the next input, skip $H_0(z)$, then use $H_1(z)$, then skip $H_2(z)$, and then start again by using $H_0(z)$.

The idea in general is shown in Figure 4.23. The structure resembles that of interpolation by an integer factor except that only the branches that are going to produce an output we will keep are used for every input sample.

Example 44 *We will work with Hertz to illustrate the fact that when we interpolate (by either an integer or a fractional factor) we never reduce the bandwidth of the input signal.*

Suppose we have a signal sampled at 500 Hz. The band of interest for the signal is from 0 to 200 Hz, i.e. a transition width of 100 Hz has been allocated. Say we want to increase the sampling rate to 1600 Hz. We choose to use a 16th-band Nyquist filter. The transition width is set to 100 Hz in order to not disturb information contained in the band of interest.

```
L      = 16;
M      = 5;
```

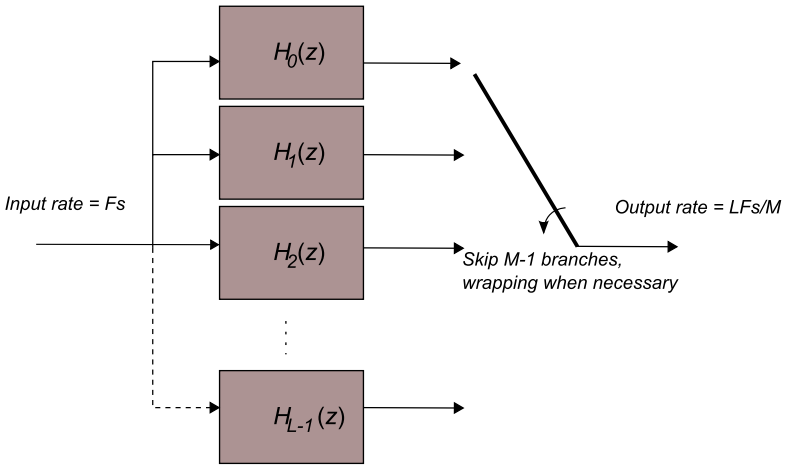


Figure 4.23: *Bank of filters used for fractional interpolation. Unlike interpolation by an integer factor, not all polyphase branches are used for each input sample.*

```
Band  = L;
Fs    = 500*L; % High sampling rate
TW    = 100;
Ast   = 80;
Hf    = fdesign.rsrc(L,M,'Nyquist',Band,'TW,Ast',TW,Ast,Fs);
Hrsrc = design(Hf,'kaiserwin');
```

Note that the sampling frequency we had to set for the design was that corresponding to full interpolation by 16, i.e. 16×500*. This is because, as we have explained, the filter is designed as a regular interpolate-by-16 filter, but is implemented efficiently so that no sample to be discarded is actually computed. The magnitude response of the filter is shown in Figure 4.24. The passband of the filter extends to 200 Hz so that the band of interest is left undisturbed. The resulting sampling frequency is 1600. However, this filter removes 15 spectral replicas as if the sampling frequency was being increased to 8000. The downsampling by 5 produces 4 spectral replicas centered around multiples of the resulting sampling frequency, i.e. 1600.*

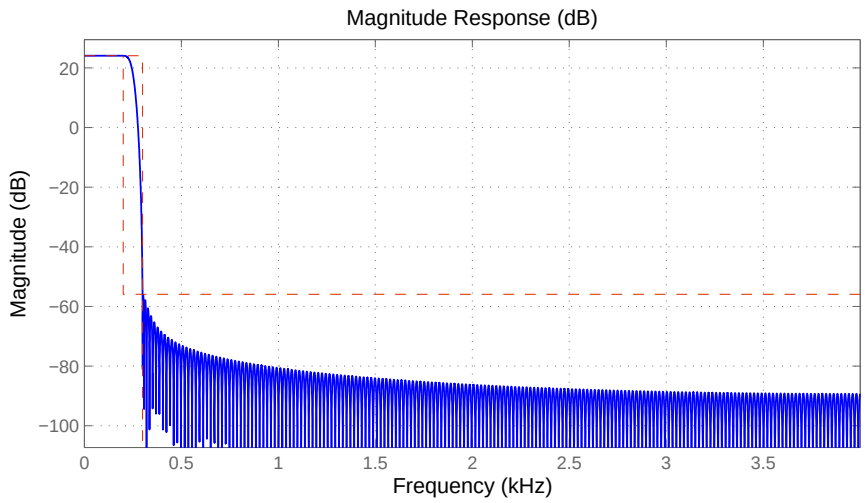


Figure 4.24: *Magnitude response of filter used for fractional interpolation. The passband extends to 200 Hz so that the band of interest is left untouched.*

4.4 Fractional decimation

We now return to the problem of decimating by a fractional factor. For now, we will restrict the discussion to rational values. As we mentioned before, in order to perform fractional decimation, it is necessary to compute samples lying between existing samples. We now know how to do so by using interpolation.

It turns out that the implementation we have just described for fractional interpolation can be used for fractional decimation as well. The only difference is that $L < M$ in this case. (Conceptually, Figure 4.22 still applies with the caveat that now $L < M$.)

In terms of designing the filter however, we need to be careful with where we set the cutoff frequency since it is no longer true that we want to keep the full spectrum of the input signal.

Indeed, fractional decimation is appropriate when the bandwidth of a signal is being reduced by a fractional factor. We use fractional decimation to reduce the rate by a corresponding factor.

Suppose for instance that we have a signal sampled at 1000 Hz. We are interested in keeping the information contained from 0 to 280 Hz. We decide to both lowpass filter the signal and reduce the bandwidth by a

factor of 7/10.

```
L     = 7;
M     = 10;
Band  = M;
Fs    = 1000*L; % High sampling rate
TW    = 140;
Ast   = 80;
Hf    = fdesign.rsrc(L,M,'Nyquist',Band,'TW,Ast',TW,Ast,Fs);
Hrsrc = design(Hf,'kaiserwin');
```

Note that once again, the sampling frequency set for the filter designed is set as if full interpolation were to be implemented. The choice of the band however, is made as M rather than L (compare to the previous example) so that the resulting filter reduces the bandwidth by a factor of L/M. The transition width is chosen so that the passband of the filter extends to the desired 280 Hz. The computation is simple, $f_c - TW/2$, where f_c is the cutoff frequency which is given by $f_s/(2\text{Band}) = 350$ Hz. Subtracting half the transition width, i.e. 70, means the passband of the filter will extend to 280 Hz as intended. The passband details for the filter can be seen in Figure 4.25.

If the input signal had a flat spectrum occupying the full Nyquist interval (worst case), after filtering and fractional decimation, the spectrum of the signal would take the shape of the filter. Its spectral replicas would be centered around multiples of the new sampling frequency, i.e. 700 Hz. The baseband spectrum along with the first positive two spectral replicas is shown in Figure 4.26. Note that as before, we have allowed for transition band overlap, but the baseband of interest presents no significant aliasing.

4.5 Summary and look ahead

In this chapter we have presented design considerations for multirate filters. In particular, it is important to understand how the cutoff frequency for filters should be set commensurate to bandwidth reduction in the case of decreasing the sampling rate. Also, we presented the rather obvious fact that when increasing the sampling rate of a signal, we do not wish to reduce its bandwidth.

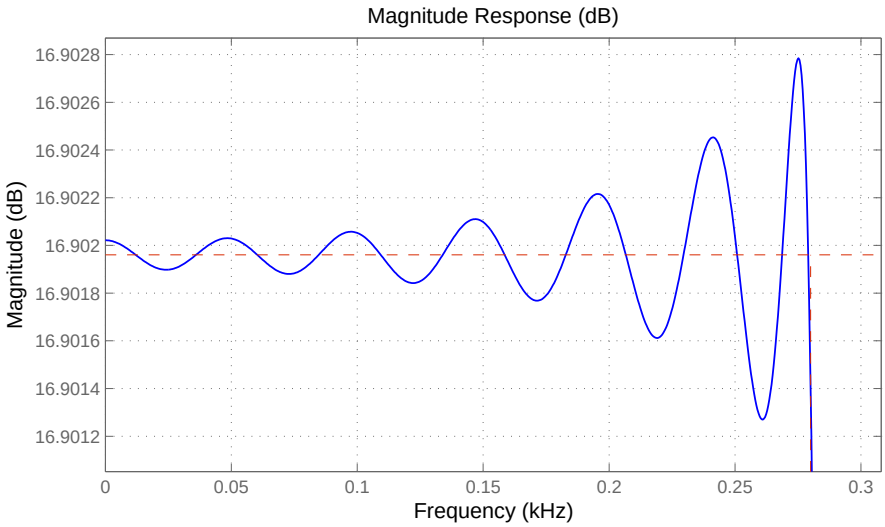


Figure 4.25: *Passband details of filter designed for fractional decimation. The passband extends to 280 Hz so that the band of interest is left untouched.*

Lowpass filters are the most common filters for multirate applications. However, highpass and bandpass filters can also be used for both decreasing or increasing the sampling rate of a signal. Although we did not touch upon the case when increasing the sampling rate, using a bandpass rather than a lowpass for such case is simply a matter of choosing which spectral replicas should be removed.

We have presented Nyquist filters as the preferred type of lowpass filters for multirate applications, however any lowpass filter in principle can be used.

When using Nyquist filters, equiripple FIR filters are not always the best choice. Kaiser window designs may have a smaller passband ripple along with increasing stopband attenuation that may be desirable for removal of spectral replicas/aliasing attenuation.

IIR halfband filters have presented as an extremely efficient way of implementing decimate/interpolate by two filters.

In the next chapter we will see that multirate Nyquist filters have the interesting property that cascades of such filters remain Nyquist overall. As such, we will see that to perform efficient multirate filtering, multistage designs can be a very attractive solution. In particular, cascading efficient

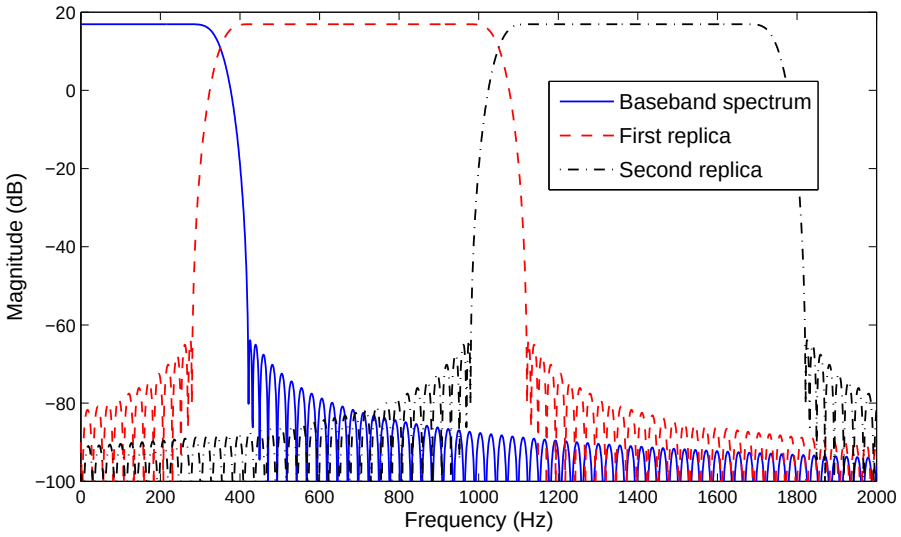


Figure 4.26: *Baseband spectrum and two positives spectral replicas of fractionally decimated signal. The input signal is assumed to have a flat spectrum which occupies the entire Nyquist interval.*

halfband filters may be a very desirable way of implementing changes in sampling rate by a factor given by a power of two.

Chapter 5

Multistage/Multirate Filter Design

Overview

Generally, given a fixed passband ripple/stopband attenuation, the narrower the transition band of a filter, the more expensive it is to implement. A very effective way of improving the efficiency of filter designs is to use several stages connected in cascade (series). The idea is that one stage addresses the narrow transition band but manages to do so without requiring a high implementation cost, while subsequent stages make-up for compromises that have to be taken in order for the first stage to be efficient (usually this means that subsequent stages remove remaining spectral replicas).

In this chapter we start by discussing the so-called interpolated FIR (IFIR) filter design method. This methods breaks down the design into two stages. It uses upsampling of an impulse response in order to achieve a narrow transition band while only adding zeros to the impulse response (therefore not increasing the complexity). The upsampling introduces spectral replicas that are removed by the second stage filter which is a lowpass filter with less stringent transition bandwidth requirements.

We will show that multirate implementations of multistage designs are the most effective way of implementing these filters. The idea follows upon the notion we have already discussed in Chapter 4 of reducing the sampling rate whenever the bandwidth of a signal is reduced.

We then extend the multistage implementation to more than two stages. Determining the optimal number of stages (and the decimation factor of each stage) in order to most efficiently implement a filter for a given set of specifications is a hard manual task. Fortunately, we show tools which perform these designs automatically for us.

As mentioned in Chapter 4, Nyquist filters should be the preferred designs used for multirate applications. The use of Nyquist filters in multistage implementations will be explored. The interesting property that Nyquist filters retain the Nyquist property when implemented in a multirate/multistage fashion is discussed. We will show that these designs are very efficient.

A particular case of Nyquist filters which are very efficient for multirate/multistage applications are halfband filters. We will see how we can obtain very efficient designs using halfband filters to implement some (or all) of the stages in a multistage design. Moreover, we can use IIR halfband multirate filters implemented in polyphase form in order to increase the efficiency of designs even further.

Finally, we show that efficient multistage implementations apply to interpolation as well as decimation. Multistage interpolation implementations will typically be mirror images of multistage decimation. Whereas with multistage decimation the simplest filter always should be the first stage of the cascade (operating at the highest sampling frequency), with multistage interpolation it is the last stage that contains the simplest filter (since in this case it is the last stage that operates at the highest sampling frequency).

5.1 Interpolated FIR (IFIR) designs

For any given FIR design algorithm, if the peak ripple specifications remain the same, the filter order required to meet a given specifications set is inversely proportional to the transition width allowed.

When the transition width is small, such as in the Specifications Set 3, the filter order required may be quite large. This is one of the primary disadvantages of FIR filters. We have already seen that relaxing the linear phase requirement results in a significant savings in the number of filter coefficients.

The so-called *interpolated FIR* (IFIR) approach [25],[26],[27] yields lin-

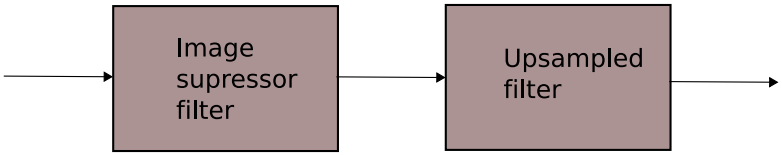


Figure 5.1: *The IFIR implementation. An upsampled filter is cascaded with an image suppressor filter to attain an overall design with a reduced computational cost.*

ear phase FIR filters that can meet the given specifications with a reduced number of multipliers.

The idea is rather simple. Since the length of the filter grows as the transition width shrinks, we don't design a filter for a given (small) transition width. Rather, we design a filter for a multiple L of the transition width. This filter will have a significantly smaller length than a direct design for the original (small) transition width. Then, we *upsample* the impulse response by a factor equal to the multiple of the transition width, L. Upsampling will cause the designed filter to compress, meeting the original specifications without introducing extra multipliers (it only introduces zeros, resulting in a larger delay). The price to pay is the appearance of spectral replicas of the desired filter response within the Nyquist interval. These replicas must be removed by a second filter (called in this context the interpolation filter or image suppressor filter) that is cascaded with the original to obtain the desired overall response. Although this extra filter introduces additional multipliers, it is possible in many cases to still have overall computational savings relative to conventional designs. The implementation is shown in Figure 5.1.

The idea is depicted by example in Figure 5.2 for the case of an upsampling factor of 3. The "relaxed" design is approximately of one third the length of the desired design, if the latter were to be designed directly. The upsampled design has the same transition width as the desired design. All that is left is to remove the spectral replica introduced by upsampling. This is the job of the image suppressor filter.

As an example of the computational cost savings, consider once again the design Specifications Set 3. The number of multipliers required for a single linear phase design was 263. An IFIR design can attain the same specs with 127 multipliers when using an upsampling factor of 6:

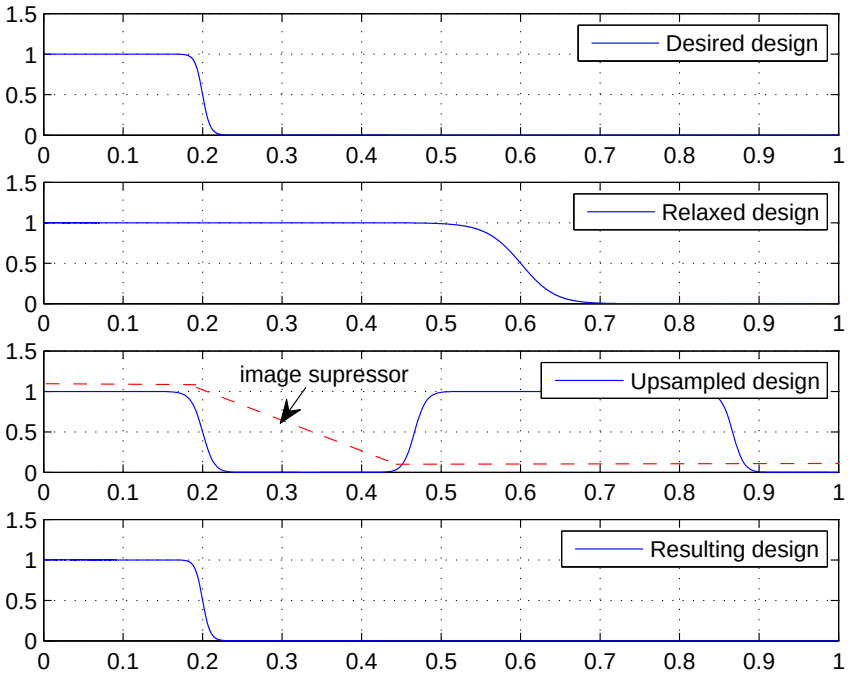


Figure 5.2: *Illustration of the IFIR design paradigm. Two filters are used to attain stringent transition width specifications with reduced total multiplier count when compared to a single filter design.*

```
Hf    = fdesign.lowpass('Fp,Fst,Ap,Ast',.12,.14,.175,60);
Hifir = design(Hf,'ifir','UpsamplingFactor',6);
cost(Hifir)
```

The response of the upsampled filter and the image suppressor filter is shown in Figure 5.3. The overall response, compared to a single linear phase equiripple design is shown in Figure 5.4.

5.1.1 Further IFIR optimizations

A drawback in the IFIR design is that the passband ripples of the two filters are combined in a disorderly fashion. In the worst case scenario, they

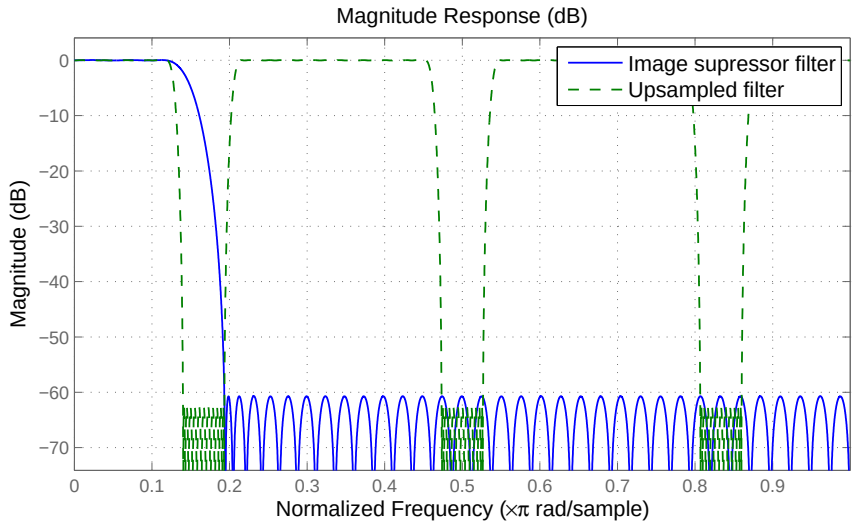


Figure 5.3: *Magnitude response of the upsampled filter and the image suppressor filter in an IFIR design.*

can add up, requiring the design to ensure that the sum of the two peak passband ripples does not exceed the original set of specifications. Close inspection of the passband of the overall design in the previous example, shown in Figure 5.5, reveals a rather chaotic behavior (but certainly within spec.) of the ripple.

Further optimized designs, [4], [28], attain a much cleaner passband behavior by jointly optimizing the design of the two filters to work better together. This results in a filter that can meet the specifications set with an even further reduction in the number of multipliers. The savings are especially significant for the image suppressor filter, which is greatly simplified by this joint optimization.

Utilizing this joint optimization, the Specifications Set 3 can be met with only 74 multipliers, once again for an upsampling factor of 6.

```
Hf   = fdesign.lowpass('Fp,Fst,Ap,Ast',.12,.14,.175,60);
Hifir = design(Hf,'ifir','UpsamplingFactor',6,...
           'JointOptimization',true);
cost(Hifir)
```

The manner in which the two filters work together is best described

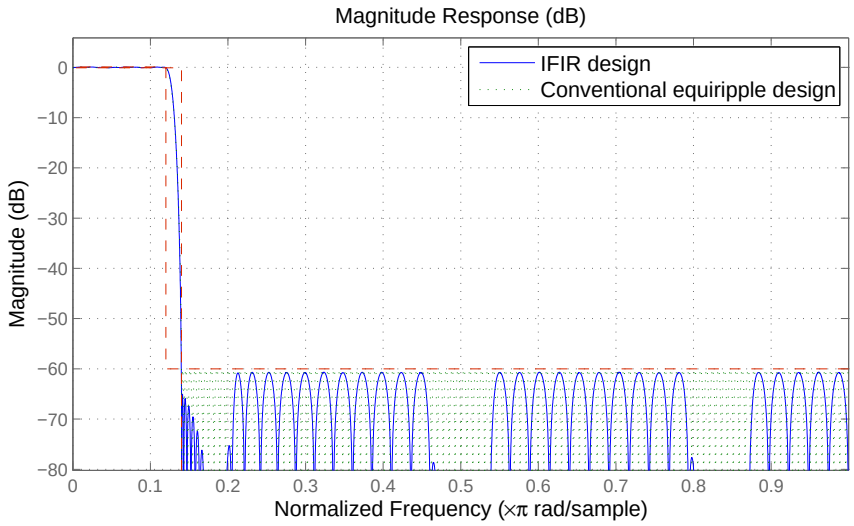


Figure 5.4: *Overall magnitude response of an IFIR design and a conventional equiripple design. The IFIR implementation requires 127 multipliers vs. 263 for the conventional implementation.*

by looking at their magnitude responses, shown in Figure 5.6. By pre-compensating for a severe "droop" in the image suppressor filter, a flat passband can be achieved with dramatic savings in the number of multipliers required for the image suppressor filter. Out of the 74 multipliers required, 29 are for the image suppressor filter and 45 for the upsampled filter. By contrast, in the previous IFIR design, 78 of the 127 multipliers correspond to the image suppressor filter, while 49 correspond to the upsampled filter.

The passband details of the overall design show a nice equiripple behavior, hinting at a much better optimized design. The passband details are shown in Figure 5.7.

The reader interested in further modifications/improvements to IFIR filters is referred to [29].

5.1.2 Multirate implementation of IFIR design

When designing an IFIR filter, the upsampling factor L used must be such that the (normalized) stopband-edge frequency ω_s satisfies $L\omega_s < \pi$. This

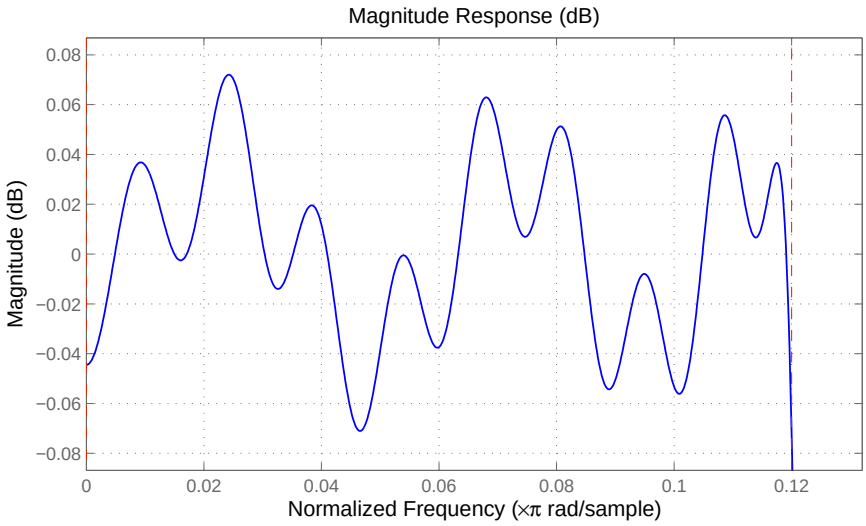


Figure 5.5: *Passband details of an IFIR design revealing a rather chaotic behavior of the ripple.*

implies that the bandwidth of the output signal would be reduced by a factor of L.

It is convenient from a computational cost perspective to reduce the sampling frequency of the filtered signal, since at that point the Nyquist criterion is being unnecessarily over-satisfied. Subsequent processing of the filtered signal without reducing its sampling rate would incur in unnecessary (and expensive) redundant processing of information.

The idea is to downsample the filtered signal by a factor of L to match the reduction in bandwidth due to filtering. If we denote by $I(z)$ the image suppressor filter and by $U(z^L)$ the upsampled filter, we would have a cascade of these two filters and a downsampler as shown in Figure 5.8. Using the Noble identities, we can "commute" the downsampler and $U(z^L)$ to obtain the implementation shown in Figure 5.9. The combination of $I(z)$ and the downsampler form a decimator which can be implemented efficiently in polyphase form.

Example 45 *Consider the same specifications as before, but now casted as a decimation filter design:*

```
Hf   = fdesign.decimator(6,'Lowpass',...
```

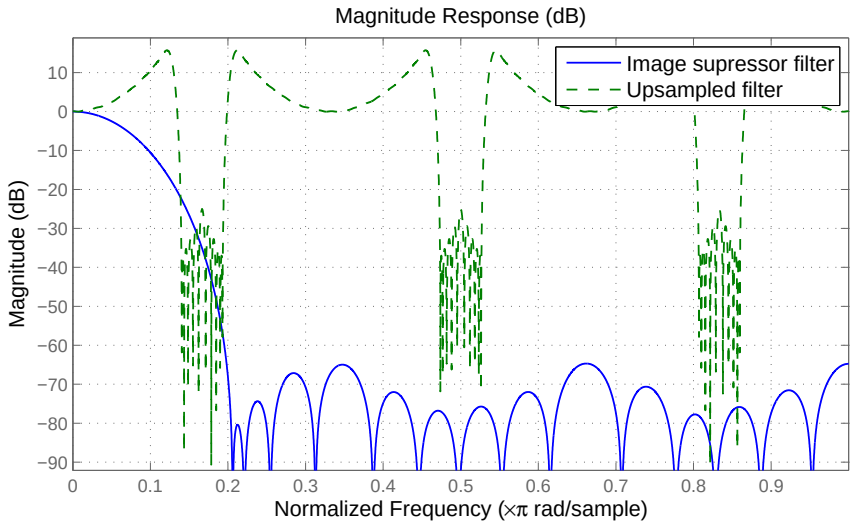


Figure 5.6: *Magnitude response of the upsampled filter and the image suppressor filter in an optimized IFIR design. The two filters are jointly optimized in the design to achieve a specifications set with a reduced number of multipliers.*

```
          'Fp,Fst,Ap,Ast',.12,.14,.175,60);
Hifir = design(Hf,'ifir','UpsamplingFactor',6,...
          'JointOptimization',true);
cost(Hifir)
```

Looking at the results from the `cost` *function, we can see that while the number of multipliers remains 74 (the same as for a single-rate design), the number of multiplications per input sample has been reduced substantially from 74 to 12.333. The number 12.333 is computed as follows. Of the 29 multipliers used for the image suppressor filter, because of its efficient decimating implementation, only 29/6 multiplications are performed on average per input sample. Because of decimation, only one out of 6 input samples (to the entire cascade) ever reaches* $U(z)$*, so per input sample, the total number of multiplications on average is* $29/6 + 45/6 = 12.333$*.*

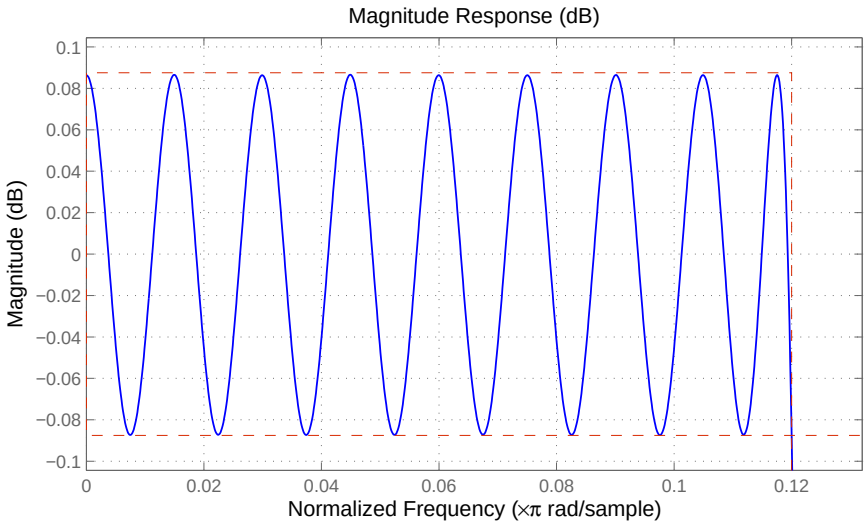


Figure 5.7: *Passband details of an optimized IFIR design. The optimized design exhibits nice equiripple behavior.*

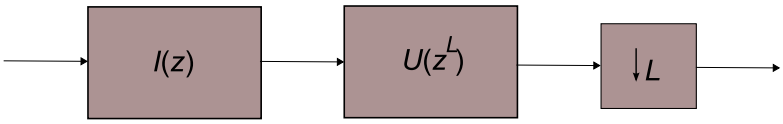


Figure 5.8: *Cascading an IFIR implementation with a downsampler.*

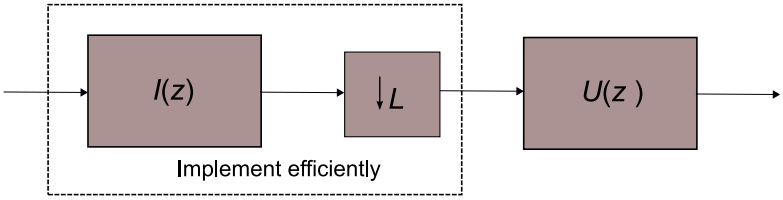


Figure 5.9: *Interchange of the downsampler and the upsampled filter using the Noble identities.*

5.2 Multistage/Multirate Designs

The procedure outlined in the previous section can be generalized to multiple stages. There are a couple of ways of thinking about the philosophy behind this. Keeping in mind that the image suppressor filter $I(z)$ is nothing but a lowpass filter, as we increase the upsampling factor L, so that the upsampled filter $U(z)$ becomes simpler, the order required by $I(z)$ to remove the spectral replicas as they come closer to each other increases as well. In other words, decreasing the complexity and computational cost of $U(z)$ results in an increased filter order for $I(z)$. If the specifications for $I(z)$ become too stringent, the cost of implementing this filter may outweigh the benefits of the savings afforded to $U(z)$. In such a situation, one may consider using an IFIR approach to implement $I(z)$ itself. After all, the original idea behind the IFIR approach is to implement a lowpass filter as a cascade of two filters, $U(z)$ and $I(z)$. Since $I(z)$ is a lowpass filter itself, why not apply the IFIR approach recursively and break up $I(z)$ into a cascade of two filters. Of course, of these two new filters, one of them is also a lowpass image suppressor filter which can in turn be implemented as a cascade of two filters and so on.

In general this line of thinking leads us to implementing a filter in multiple stages. The multirate implementation yields a multirate-multistage design which lowpass filters data while decreasing the sampling rate at each stage for maximum computational efficiency. Determining the optimal number of stages, in the sense that the computational cost is minimized can be a difficult task. Fortunately, the Filter Design Toolbox provides tools to automate such designs. We will look at such tools in a moment, but before that, let's look at a different line of thought that also results in multistage/multirate filter designs.

We know that the filter order is increased as the transition width of a filter is decreased. When we talk about decreasing the transition width, we are usually referring to the width in terms of normalized frequency. In terms of absolute frequency, one could ask what is a narrow transition width: 1 kHz? 100 Hz? maybe 10 Hz? The answer is: *it depends*. A transition width is narrow relative to the sampling frequency that is involved. For example 10 Hz is probably quite narrow relative to a sampling frequency in the gigahertz range, but it is not narrow relative to a sampling frequency of say 50 Hz. One should assess whether a transition region is narrow by considering the normalized transition width $\Delta f / f_s$ rather than

the absolute transition width Δf.

In many applications, we cannot change the transition width required for a filter design just because it makes the filter order too large. The application sets the requirement for the transition width and this is typically something that cannot be changed. What we can do however, is change the sampling frequency so that the transition width is not narrow relative to such sampling frequency. This is another way of looking at the idea behind multistage/multirate designs. If a very narrow transition width is required relative to the sampling frequency involved, the multistage approach is to not try to achieve the narrow transition in one shot. Instead, we lower the sampling rate in stages by using simple lowpass filters until we achieve a rate at which the required transition width is not large relative to such rate. Then we design a filter that provides the required transition.

The computational savings come from two factors. On the one hand, the normalized transition width $\Delta f / f_s$ will be much larger since f_s is smaller, resulting in a filter of lesser order. On the other hand, because the sampling rate is lower at this point, the number of input samples that have to be processed by the filter is much smaller than if we were handling the original large sampling rate. Of course there are other lowpass filters we need to implement that handle the full high sampling rate and the large number of input samples associated with it. However, these other filter are implemented with large transition widths, so that their order is low thus requiring a small number of operations per input sample.

In terms of the frequency response, the simple filters operating at high sampling rates are used to remove spectral replicas from the filters operating at lower sampling rates downstream. The concepts are easily illustrated through an example.

Example 46 *Consider the design of a lowpass filter with the following specifications:*

Specifications Set 6

1. *Cutoff frequency:* 550 *Hz*
2. *Transition width:* 100 *Hz*
3. *Maximum passband ripple: 0.1 dB*

4. *Minimum stopband attenuation: 80 dB*

5. *Sampling frequency:* 10 *kHz*

The specifications imply that the band of interest extends from 0 to 500 Hz. Since the bandwidth is being reduced by about a factor of 8, we decide to design a decimation filter with a decimation factor of 8 to meet the specifications and reduce the sampling-rate after filtering accordingly.

If we were to design a single-stage equiripple decimator, the design would take 343 multipliers and about 48 multiplications per input sample. In order to design a general multistage/multirate filter and allow for the tools to determine the optimal number of stages we use the following commands:

```
Hf   = fdesign.decimator(8,'Lowpass',...
          'Fp,Fst,Ap,Ast',500,600,.1,80,1e4);
Hmulti = design(Hf,'multistage');
cost(Hmulti)
```

The number of multipliers required is 114. The number of MPIS is 18.875. By inspecting the resulting design, we can see that a 3-stage filter has resulted. In this example, each stage has a decimation factor of 2, which accounts for the overall decimation factor of 8 we required.[*]

A plot of the magnitude response of each stage helps understand the behavior of the design and the source for the computational savings. This is shown in Figure 5.10. The simplest filter of all operates at the highest rate as should usually be the case. This first stage has a very wide transition band starting at 500 Hz (therefore not affecting the band of interest) and ending at about 4450 Hz. The second stage operates at 5 kHz given the decimation by two of the first stage. Its normalized transition width is also quite wide, given its sampling frequency. The job of this filter is to eliminate spectral replicas of the third stage centered around odd multiples of the third stage's sampling frequency. Because the second stage has replicas centered about multiples of 5 kHz, it cannot eliminate the spectral replicas of the third stage centered around even multiples of the third stage's sampling frequency. The first stage takes care of those. Notice that only the last stage has the narrow 100 Hz transition width. By that time, the sampling frequency has

[*] For this example, a joint-optimized IFIR design is more efficient than a general multistage design. This can happen in some cases, due to the highly optimized interaction between the two stages as previously discussed. This joint-optimization is not carried out for general multistage designs. This is an important thing to keep in mind.

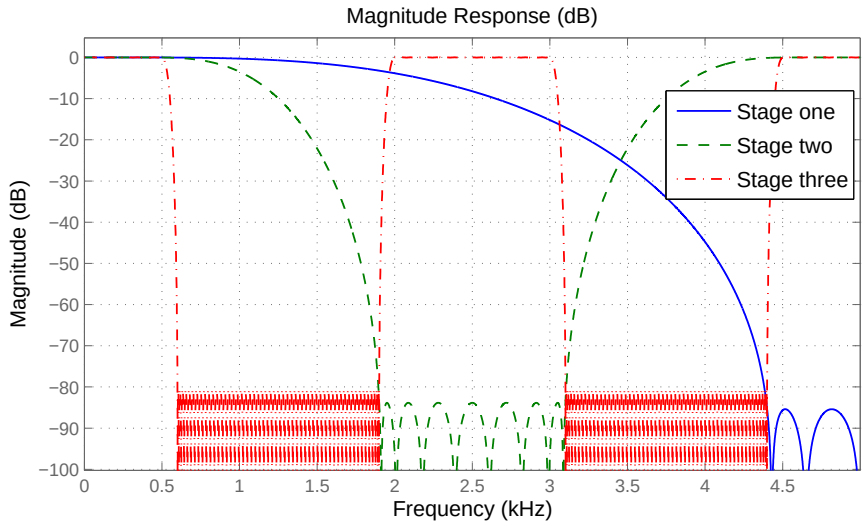


Figure 5.10: *Multistage design of decimation filter for Specifications Set 6.*

been reduced by a factor of 4, so the normalized transition width is 4 times larger (100/2500 vs. 100/10000) than it would have been if we had done a single-stage design. This accounts for most of the computational savings.

The overall magnitude dB response of the three-stage filter is obtained by adding the responses of each stage. This is shown in Figure 5.11.

5.2.1 Setting the number of stages

If so desired, the number of stages can be manually controlled up to a certain point. However, note that certain restrictions apply. The decimation factor must be a number that can be factored into as many factors as the number of stages. In some cases, it is advisable to manually try a different number of stages than the one automatically provided by the tools since similar computational costs may be found for different number of stages and other things (such as implementation complexity) may factor in to the decision of which design to use.

Example 47 *We can use the* `Nstages` *option to control the number of stages in a design. Consider the following two designs for the same design specifications:*

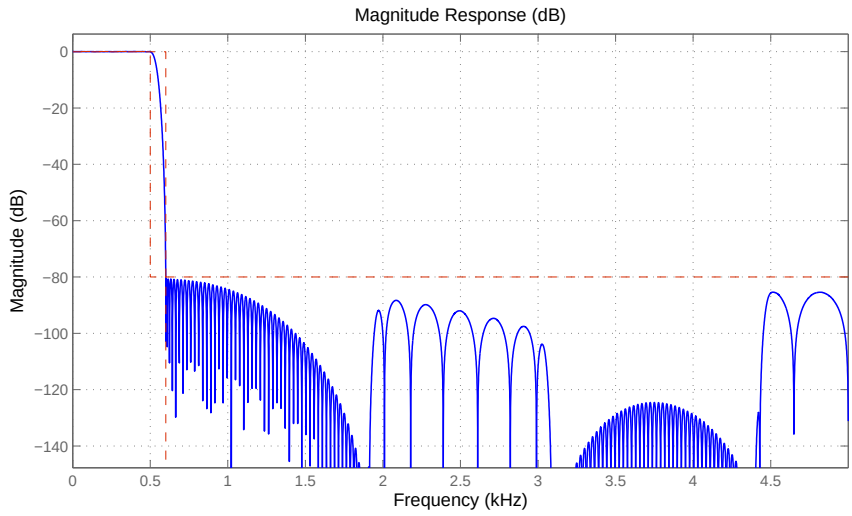


Figure 5.11: *Overall magnitude response of multistage design for Specifications Set 6.*

```
Hf    = fdesign.decimator(16,'Lowpass',...
'Fp,Fst,Ap,Ast',.05,.06,.5,70);
Hmulti = design(Hf,'multistage');
Hmulti2 = design(Hf,'multistage','Nstages',3);
```

The number of stages that is determined automatically for `Hmulti` *is 4. However, we force* `Hmulti2` *to have 3 stages. The computational cost of both designs are quite comparable. However, the complexity of implementing 3 stages in hardware may be less than that of implementing 4 stages. This could be the deciding factor between one design and the other.*

5.3 Multistage/Multirate Nyquist filters

Multirate Nyquist filters have the interesting property that if you cascade two or more of them together (all decimators or all interpolators) the resulting multistage/multirate filter is still a Nyquist filter [30]. This fact allows us to extend the reasoning behind the use of multistage/multirate designs to Nyquist designs.

We already know that Nyquist filters are well-suited for (either decimation or interpolation) multirate applications. The ability to break down a multirate Nyquist design into several multirate Nyquist stages provides very efficient designs. In many cases, halfband filters are used for the individual stages that make up the multistage design. Whenever a halfband filter is used for a stage, there is the possibility to use an extremely efficient IIR multirate halfband design as long as we allow for IIR filters as part of the design.

Example 48 *As an example, consider the design of a Nyquist filter that decimates by 8. We will compare single-stage a multistage design to illustrate the computational savings that can be afforded by using multistage Nyquist filters.*

```
Hf = fdesign.decimator(8,'nyquist',8,.016,80);
H1 = design(Hf,'kaiserwin');
H2 = design(Hf,'multistage');
H3 = design(Hf,'multistage','Nstages',2);
```

The computational costs of the three designs are summarized in the following table:

	NMult	*NAdd*	*Nstates*	*MPIS*	*APIS*	*NStages*
`H1`	*551*	*550*	*624*	*68.875*	*68.75*	*1*
`H2`	*93*	*90*	*174*	*15.625*	*14.75*	*3*
`H3`	*106*	*104*	*182*	*17.125*	*16.75*	*2*

The equivalent overall filters for all three designs are 8-band Nyquist filters. The default multistage design, `H2`*, is a 3-stage filter with each individual stage being a halfband filter (each halfband filter is different though).* `H3` *is 4-band Nyquist decimator followed by a halfband Nyquist decimator.*

5.3.1 Using IIR halfband filters

Whenever we have a halfband decimator (or interpolator) we may want to consider using either an elliptic halfband filter (if phase linearity is not an issue) or a quasi linear-phase IIR halfband filter (if phase linearity is important) to increase computational savings.

As an added bonus, all Nyquist designs tend to have a very small passband ripple. If we use IIR halfband filters, the passband ripple can be in the micro-dB range or even smaller!

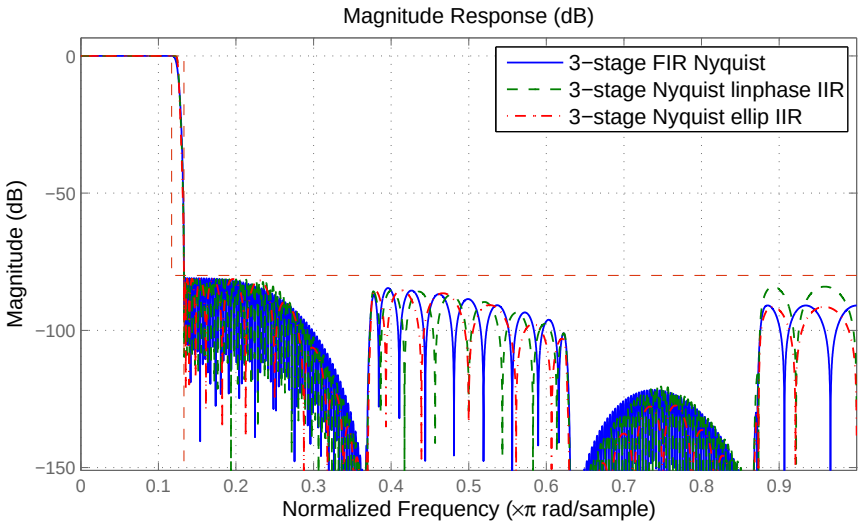


Figure 5.12: *Passband phase response for multistage Nyquist FIR and IIR filters.*

In the previous example, if we substitute the halfband filters of each stage in `H2` with quasi linear-phase IIR filters,

```
H4 = design(Hf,'multistage','HalfbandDesignMethod','iirlinphase');
```

we can get saving close to a factor of 3 in both total number of multipliers as well as MPIS.

If we can use elliptic halfbands,

```
H5 = design(Hf,'multistage','HalfbandDesignMethod','ellip');
```

the savings are about twice as good. The computational cost is shown below

	NMult	NAdd	Nstates	MPIS	APIS	NStages
`H4`	35	70	75	5.625	11.25	3
`H5`	12	24	18	2.625	5.25	3

The magnitude response of the 3-stage Nyquist FIR design along with the two 3-stage Nyquist IIR designs is shown in Figure 5.12. All three designs behave similarly overall as expected since the specifications for each halfband (each stage) are the same. However, although the 3-stage

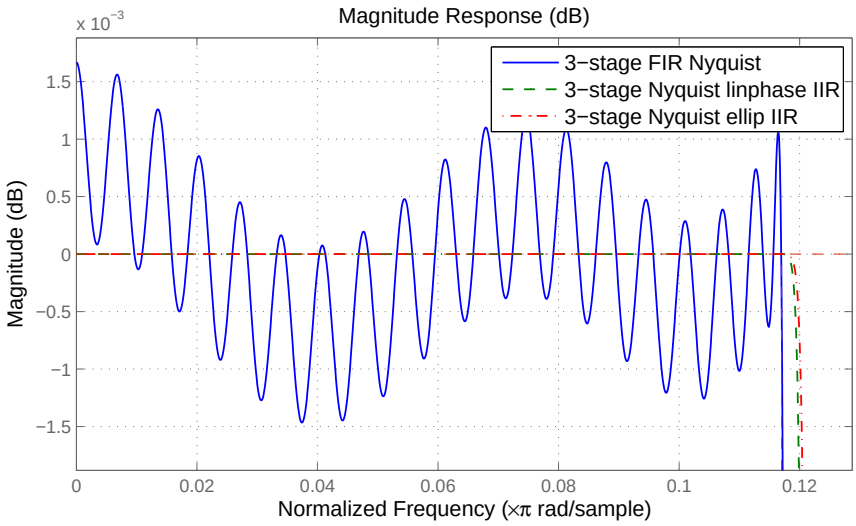


Figure 5.13: *Passband phase response for multistage Nyquist FIR and IIR filters.*

FIR Nyquist design has a small passband ripple, it is nowhere near the tiny passband ripple of the IIR designs. The passband details are shown in Figure 5.13.

As stated before, the computational savings of the elliptic design come at the expense of phase non-linearity. On the other hand, the group-delay of the elliptic design is much lower than the other two designs. The passband group-delay is shown in Figure 5.14.

We can use `realizemdl(H4)` or `realizemdl(H5)` to generate Simulink blocks that implement these multistage decimators using halfband IIR filters in efficient polyphase form.

5.4 Multistage interpolation

Just as with decimation, using multiple stages to perform interpolation can result in significant implementation savings. The reason once again has to do with the ratio of the transition width relative to the sampling frequency.

To see this, consider implementing a filter that interpolates by a factor of 8, increasing the sampling-rate from 1250 Hz to 10000 Hz with a cutoff

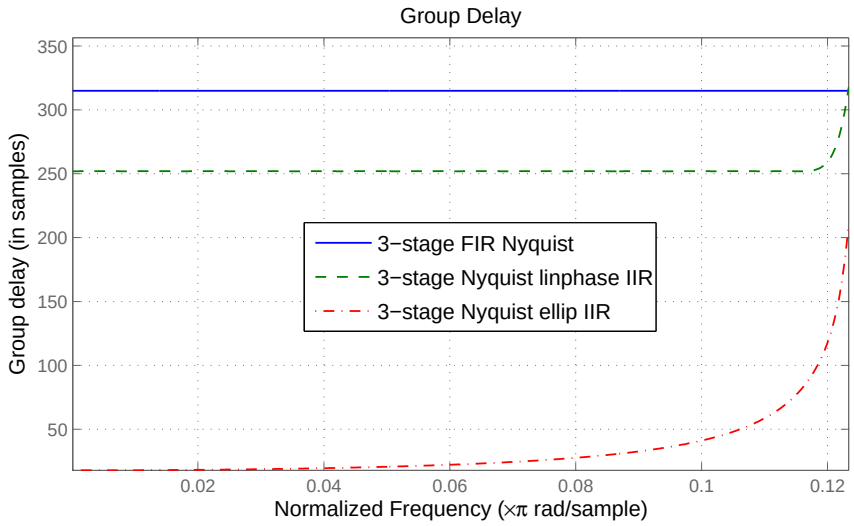


Figure 5.14: *Passband group-delay for multistage Nyquist FIR and IIR filters.*

frequency of 625 Hz and a transition width of 312.5 Hz. Let us compare the transition width, relative to the sampling frequency of a single-stage design vs. the first stage of a multistage design. Suppose the first stage of the multistage design is a halfband filter that interpolates by a factor of two (from 1250 Hz to 2500 Hz).

Figure 5.15 shows a comparison of the two filters. They both have the same transition width, yet the sampling frequency ratio is 4-to-1. Therefore the ratio $\Delta f / f_s$ is four times smaller for the single-stage filter than for the first stage of a multistage implementation. Accordingly, the number of coefficients for the single-stage design is 142 (with 135 adders), whereas the first stage of a multistage implementation has only 22 multipliers (and 21 adders). The fact that we have a savings of more than a factor of 4 in terms of number of multipliers is owed to the fact that we use a halfband filter for the latter design which yields further savings. Note that in general for polyphase interpolators, the number of coefficients equals the number of multiplications per input sample (MPIS).

Of course, as is apparent from Figure 5.15, the two filters are not performing the same job, so the comparison is unfair. While the single-stage design removes all spectral replicas in the new Nyquist interval (with $f_s = 10000$), the halfband filter only removes some spectral replicas. There

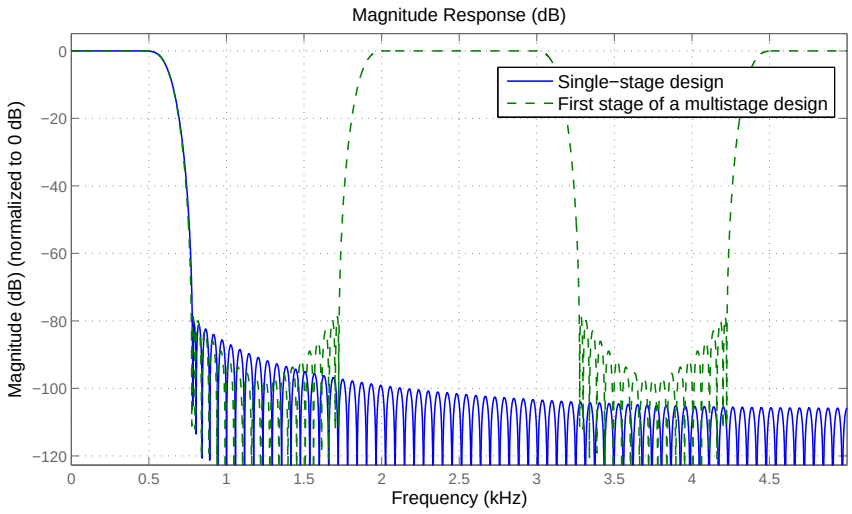


Figure 5.15: *Comparison of a single-stage design with the first stage of a multistage design.*

is clean-up left to be done by subsequent interpolation filters operating at higher rates (after all, we have only interpolated to 2500 Hz with the halfband filter).

Figure 5.16 shows a full 3-stage design (each stage being a halfband interpolator) compared to a single-stage design. Both designs satisfy the same transition width and stopband attenuation requirements. Overall the multistage design requires 40 coefficients and 72 MPIS *. The code that was used for these two designs follows:

```
f = fdesign.interpolator(8,'Nyquist',8,'TW,Ast',312.5,80,1e4);
h = design(f,'kaiserwin');
g = design(f,'multistage','Nstages',3);
```

As with multistage decimators, the use of IIR halfbands implemented in efficient allpass-based polyphase structures can improve the efficiency of designs beyond what can be achieved with FIR filters. For example, for the same specifications, this design:

* Since each stage produces more samples at its output than the number of samples at its input, for multistage interpolators the number of multiplications-per-input-sample to the overall implementation is larger then the number of coefficients required for all stages.

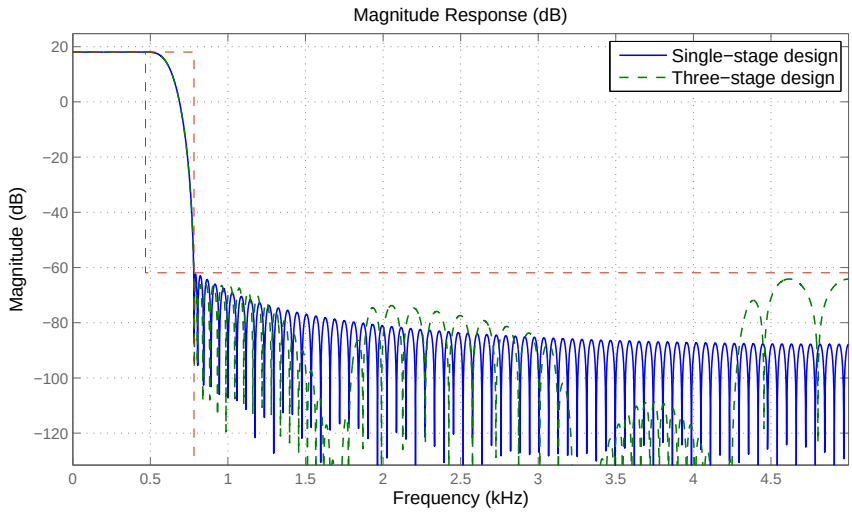


Figure 5.16: *Comparison of a single-stage design with a 3-stage design.*

```
hm = design(f,'multistage','Nstages',3,...
     'HalfbandDesignMethod','iirlinphase');
```

results in a 3-stage filter with 16 coefficients and 30 MPIS.

Figure 5.17 shows how subsequent stages remove spectral replicas that remain from the first stage. Notice how as the sampling-rate increases, the transition-width for each corresponding stage also increases so that the ratio $\Delta f / f_s$ remains large resulting in simple filters. For interpolation, the first stage is usually the most involved, but this is OK since the sampling frequency is the lowest at that point. The complexity of each subsequent stage is reduced gradually since each filter has to operate at higher and higher sampling frequencies.

5.5 Summary and look ahead

We have shown how multistage implementation of filters can result in significant computational savings. The savings are increased when multistage implementation are combined with multirate filters. There is an optimal number of stages which provide maximum computational savings.

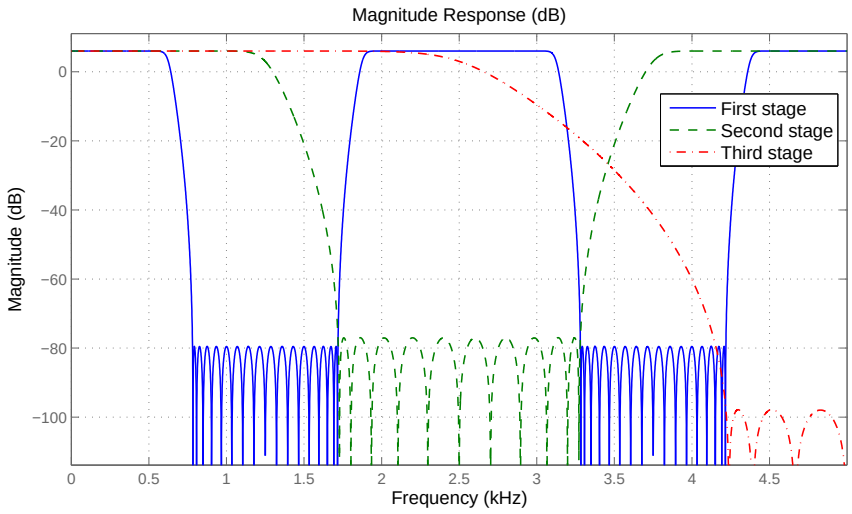


Figure 5.17: *Removal of spectral replicas performed by each stage of a 3-stage design.*

This optimal number can be determined automatically by the design tools discussed.

Multistage Nyquist filters provide very efficient implementations. This is particularly true for halfband filters, especially IIR halfbands. These multistage filters are equally adequate for efficient decimation or interpolation purposes.

In the next chapter we will discuss special multirate filters that are well-suited for multistage implementations. These include hold and linear interpolator filters as well as CIC interpolation/decimation filters. We will also discuss the use of Farrow filters for fractional decimation/interpolation. These filters can be used to provide fractional sampling-rate changes by an arbitrary factor (not necessarily a ratio of two integers). Moreover, they can do so while using a small number of multipliers.

Chapter 6

Special Multirate Filters

Overview

In this chapter we look at some special filters that come in handy for multirate applications.

We first look at hold interpolators which perform basic interpolation by simply repeating input samples L times (L being the interpolation factor).

Next, we look at linear interpolators and show how they can be thought of as two hold interpolators "put together".

Both linear and hold interpolators are attractive because of their simplicity. We see that they are very crude approximations to an ideal lowpass filter used for interpolation. Usually, because of their simplicity, they are used at the last stage of a multistage implementation, operating at high sampling rates.

We then move on to CIC interpolators and show how these are just generalizations of hold/linear interpolators. By using multiple (more than two) sections, CIC interpolators can obtain better attenuation than hold or linear interpolators. The nice thing about CIC interpolators is that they can be implemented without using multiplication. This is an attractive feature for certain hardware such as FPGAs and ASICs because multipliers take up quite a bit of area and are difficult to make to operate at very high clock rates.

CIC filters can also be used for decimation. Unlike the interpolation case, in decimation, CIC filters are usually used at the first stage of a multistage design. This is because at that stage the sampling-rate is the highest,

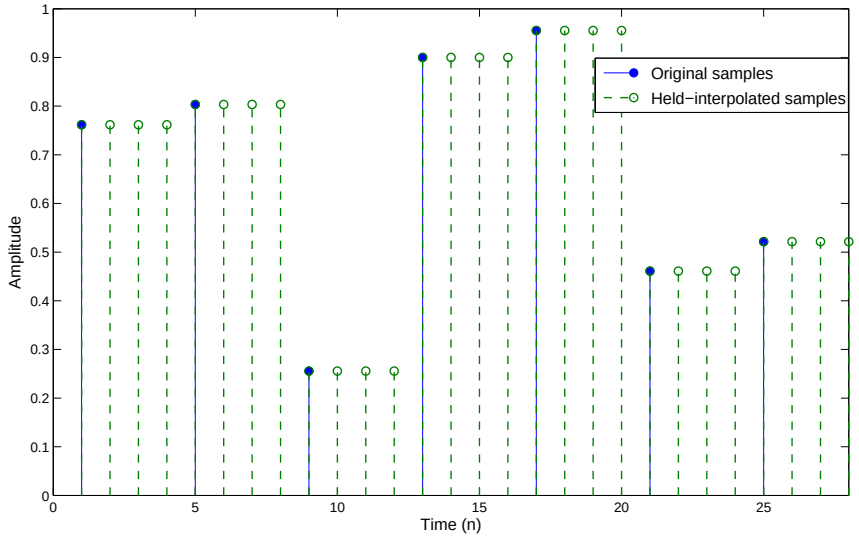


Figure 6.1: *Interpolated samples using a hold interpolator with a factor* $L = 4$.

making the multiplierless feature of CIC filters very attractive.

We then show how to design CIC compensators for either interpolation or decimation. These filters are lowpass filters whose passband is shaped to compensate for the droop in the passband of CIC filters. Of course, since linear and hold interpolators are special cases of CIC interpolators, CIC compensators can be used for those linear and hold cases as well.

Finally, Farrow filters are shown both for single-rate fractional-delay applications as well as for changing the sampling-rate of signals (by integer or fractional factors). The main allure of Farrow implementations are the low number of fixed multipliers that are required and the tunability of the fractional delay while the filter is running without having to re-design.

6.1 Hold interpolators

The simplest interpolator that one can build is the one that performs a (zero-order) hold of each sample, inserting as many repeated samples as desired. For example, Figure 6.1 shows the result of a hold interpolator, interpolating by a factor of 4.

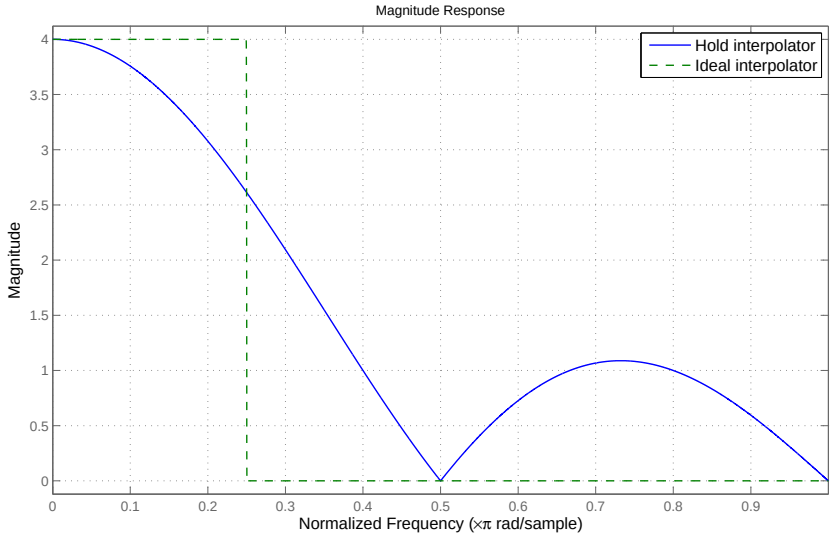


Figure 6.2: *Magnitude response of a hold interpolator with a factor $L = 4$. The ideal interpolator is shown for comparison.*

Hold interpolators are attractive because they require no arithmetic in order to be implemented. The hold interpolator is an FIR interpolator whose impulse response is simply a series of L 1's.

```
Hm = mfilt.holdinterp(4);
Hm.Numerator
```

As we know, the ideal interpolator is a lowpass filter that removes $L - 1$ adjacent spectral replicas of the signal being interpolated. The hold interpolator is a very crude approximation to the ideal interpolator as can be seen with `fvtool(Hm)`. Figure 6.2 shows the magnitude response of the hold interpolator compared to the ideal interpolator for the case $L = 4$. The hold interpolator has a very wide transition band, allows significant high-frequency content to pass through, and distorts the original data in the band of interest. Note that the fact that high frequency remains after hold interpolation is obvious from the time-domain plot shown in Figure 6.1. The fast transitions between the last interpolated sample for a given original sample and the first interpolated sample for the next original sample are indicative of high frequency content.

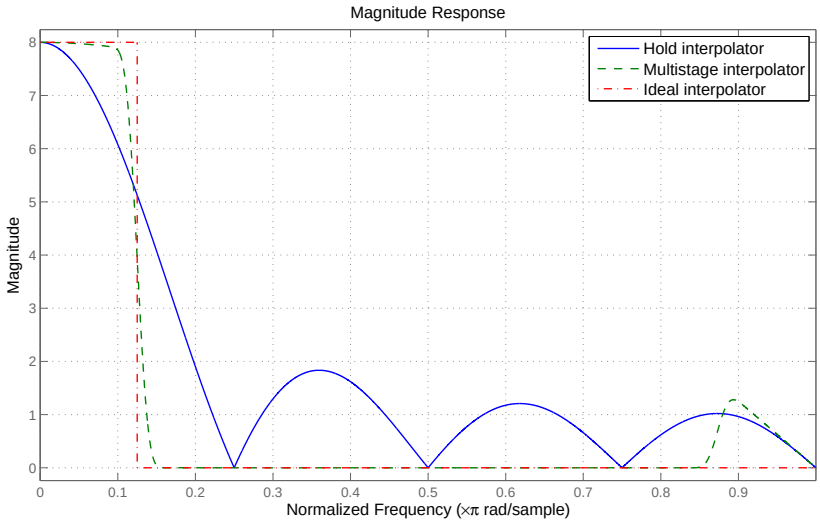


Figure 6.3: *Hold interpolator compared with multistage interpolator for $L = 8$. For reference, the ideal interpolator is shown.*

For these reasons, hold interpolators are usually not used in isolation, but rather are used as part of a cascade of interpolators (typically at the last stage, which operates at the fastest rate and therefore benefits the most of the fact that no multiplications are required). When used in such cascade configurations, most of the high frequency content has been removed by previous stages. Moreover, the distortion of the signal in the band of interest is much less severe since the band of interest will occupy a much smaller portion of the passband of the hold interpolator.

Example 49 *Let us compare the effect of using a hold interpolator for a full interpolation by a factor $L = 8$ vs. using a hold interpolator for a factor $L_3 = 2$ in conjunction with two halfband filters, each one interpolating by two.*

Figure 6.3 shows a comparison of a possible multistage design including two halfband filters followed by a hold interpolator vs. a single-stage interpolate-by-8 hold interpolator design. For reference, the ideal interpolate-by-8 filters is also shown. While the multistage interpolator does allow some high-frequency content through, it is nowhere as bad as the hold interpolator acting alone. Also, assuming the band of interest extends to 0.1π, the multistage interpolation introduces a maximum distortion of about 0.15 dB while the hold interpolator introduces a distortion of almost 2 dB.

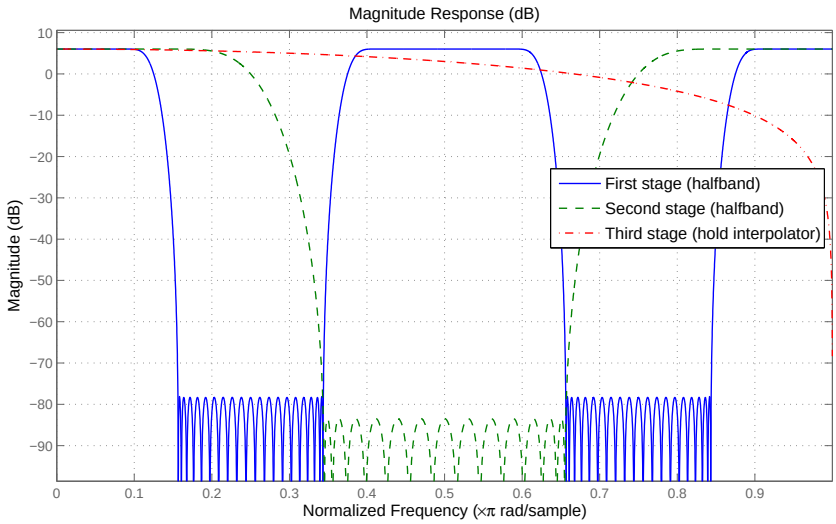


Figure 6.4: *Magnitude response of each of the three stages of the multistage design.*

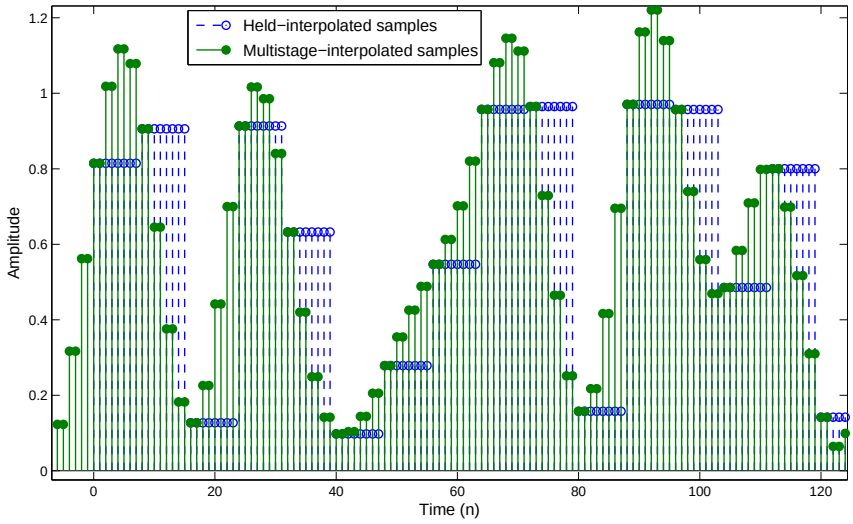


Figure 6.5: *Comparison of interpolated data using a hold interpolator and a multistage interpolator with* $L = 8$.

One thing to keep in mind that since interpolation is usually used right before D/A conversion, the remnant high frequency content can be removed by an analog (anti-image) post-filter. This will ensure a smooth analog waveform. As is apparent from Figure 6.3, because of the multistage interpolation, the remaining high frequency content will be far away from the band of interest (the passband of the filter), making it easy for a low-order analog lowpass post-filter (with a wide transition band) to remove said high frequencies.

Figure 6.4 shows the magnitude response of each of the three stages of the multistage design. Note that because of the prior interpolation by 4 (provided by the first two stages) the band of interest is only a small fraction of the passband of the hold interpolator. For this reason, the passband distortion is minimal.

Figure 6.5 shows the result of filtering the same data with the hold interpolator compared to using the multistage interpolator. While it is obvious that some high frequency content remains in either case, it is also obvious that overall the multistage-interpolated data is much smoother and therefore has much less high-frequency content.

A hold interpolator can be thought of in polyphase terms. As usual, the polyphase components are formed by taking every Lth value from the impulse response. In this case, since the impulse response is a sequence of 1's of length L, each polyphase branch consists trivially of a single coefficient equal to one. This of course is consistent with the notion that for every input sample, each polyphase branch computes one of the interpolated values. In this case, each interpolated value is compute by "multiplying" the input sample by one.

6.2 Linear interpolators

After hold interpolators, the next simplest interpolation filter is the linear interpolator. While the hold interpolator only uses one sample in order to compute the interpolated values (and therefore has only one coefficient per polyphase branch), in the case of linear interpolators, two adjacent samples are used in order to compute the interpolated values. This of course means that each polyphase branch will have two coefficients as we will soon see.

Linear interpolation is very simple to understand. Consider the plot in Figure 6.6 which depicts the case $L = 5$ (it is the same idea for other

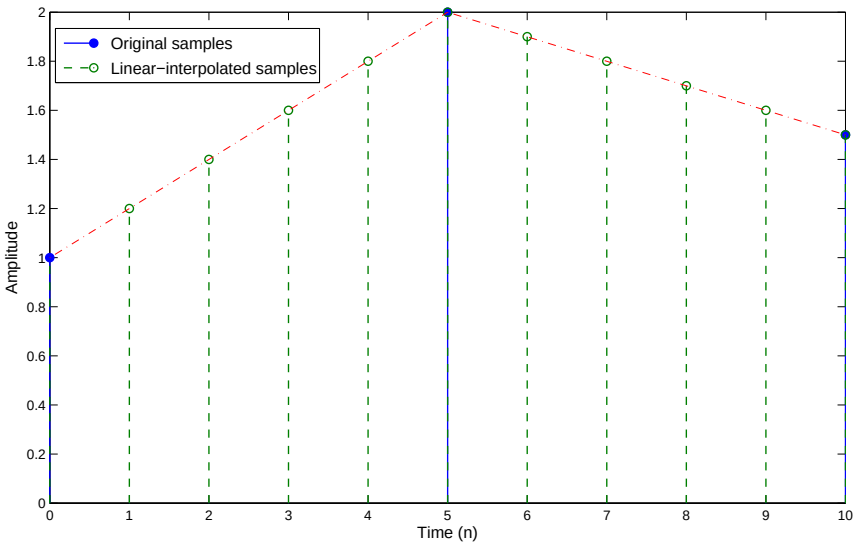


Figure 6.6: *Computing linearly-interpolated samples for* $L = 5$.

values of L). For every two existing samples, the interpolated values are computed by forming a weighted average of the existing samples. The weights are determined by the proximity of the interpolated values to each of the existing samples. In this case, the interpolated sample closest to an existing sample is four times closer to that existing sample than it is to the existing sample on its other side. Therefore, when we form the weighted average in order to compute the interpolated sample, we weight one of the existing samples four times more than the other. This results in coefficients equal to 0.2 and 0.8 (they always add to 1) for that polyphase branch. The coefficients for other polyphase branches are similarly computed based on the ratio of the distance between the new interpolated sample and the existing surrounding two samples.

This means that for the case $L = 5$, the 5 polyphase branches can trivially be computed as:

```
L = 5;
Hl = mfilt.linearinterp(L);
p = polyphase(Hl)
```

Notice that the last polyphase branch is trivial (its coefficients are 1 and 0).

This allows for the existing samples to be contained unchanged within the interpolated samples. That is, linear interpolators (and hold interpolators) are special cases of Nyquist filters.

As usual, the overall interpolation filter is a lowpass filter whose coefficients can be formed from the polyphase branches.

```
Hl.Numerator
ans =
  Columns 1 through 6
    0.2000    0.4000    0.6000    0.8000    1.0000    0.8000
  Columns 7 through 9
    0.6000    0.4000    0.2000
```

A linear interpolator can be thought of as the convolution of two hold interpolators. Indeed, for the case $L = 5$, the coefficients of the FIR hold interpolation filter are:

```
Hm = mfilt.holdinterp(5);
Hm.Numerator
ans =
     1     1     1     1     1
```

And of course

```
1/5*conv(Hm.Numerator,Hm.Numerator)
ans =
  Columns 1 through 6
    0.2000    0.4000    0.6000    0.8000    1.0000    0.8000
  Columns 7 through 9
    0.6000    0.4000    0.2000
```

is exactly equal to `Hl.Numerator`.

As we know, convolution in time is equal to multiplication in frequency. Therefore, the magnitude response of a linear interpolator is given by multiplying (or adding when working with dB) the magnitude response of two hold interpolators. This can be easily verified with `fvtool(Hm,Hl)`. The plot comparing the magnitude response of both interpolators is shown (in dB) in Figure 6.7 (we have normalized the passband gain to 0 dB to ease the comparison).

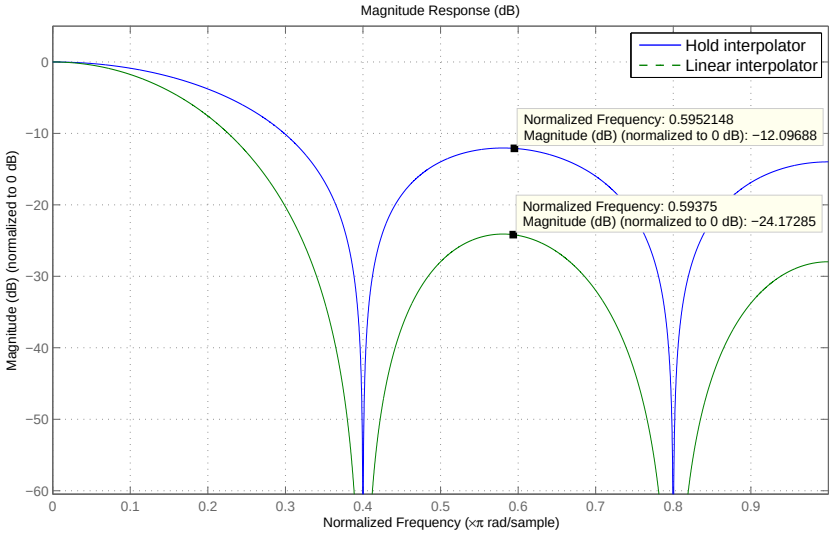


Figure 6.7: *Comparison of linear and hold interpolators for* $L = 5$.

Because the magnitude response of the linear interpolator is twice that of the hold interpolator (in dB), the linear interpolator provides more stopband attenuation and therefore better lowpass filtering than a hold interpolator. This should be obvious to see if you think of the situation in the time domain. While linearly-interpolated samples still exhibit some abrupt transitions which indicate some high frequency components remain, it is not nearly as abrupt as taking the same samples and interpolating with a hold interpolator.

On the other hand, the passband distortion of the linear interpolator is also twice that of the hold interpolator. However, as with hold interpolators, linear interpolators are usually used as the last stage of a multistage interpolation scheme. Therefore, the passband distortion due to linear interpolation can be minimal in such configurations. Although the concept of closeness is a relative notion, one can intuitively think that if samples have been interpolated already so that they are very close to each other, the error added by using linear interpolation rather than a higher-order filter is much smaller than if we try to directly linearly interpolate samples that are far apart in one shot.

The behavior is very similar to the comparison shown above between

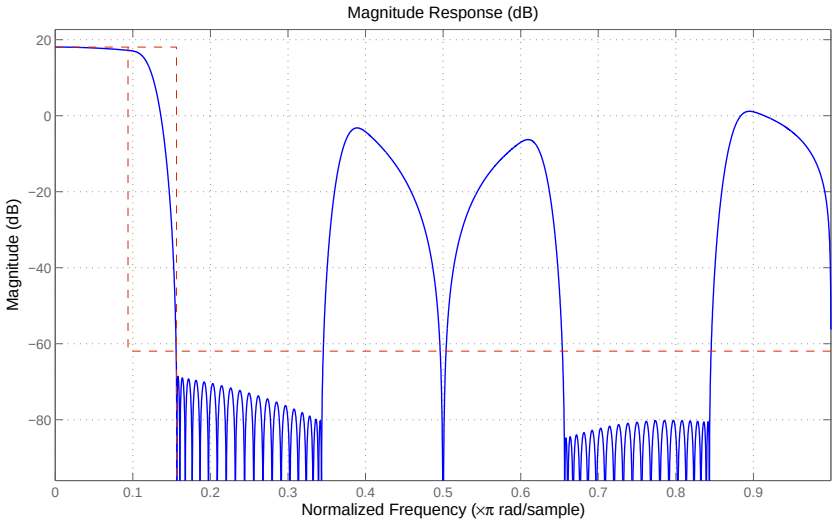


Figure 6.8: *Magnitude response of a multistage interpolator consisting of a halfband filter, followed by an interpolate-by-2 linear interpolator followed by an interpolate-by-2 hold interpolator.*

using a single-stage interpolate-by-8 hold interpolator and using a multistage interpolator with a simplified (hold or linear) final stage. In fact, one perfectly reasonable possibility is to have a linear interpolator in the next-to-last stage of a multistage configuration and a hold interpolator in the last stage.

Figure 6.8 shows just that configuration. In this case, we have obtained it by replacing the halfband filter in the second stage of the multistage filter used above, with a linear interpolate-by-two filter. Once again, if the interpolation is being done just prior to D/A conversion, it is assumed that the analog anti-image post-filter will remove the remaining high-frequency content*

The idea of convolving hold interpolators in order to obtain better stopband attenuation can be generalized to more than two such interpolators. CIC interpolation filters [31] can be seen as just that. The number of section

* We should note that part of the high-frequency content is attenuated by the staircase reconstructor that is typically used in D/A conversion in order to make the signal analog. A post-filter is used after the staircase reconstructor to smooth-out the analog waveform by removing remaining high-frequency content. For more on this, see [3].

of the CIC interpolators will correspond to the number of hold interpolators convolved. As with linear interpolators, multi-section CIC interpolators attain better stopband attenuation at the expense of larger passband distortion. In the next section we will take a look at CIC interpolators. After that, we will look at the design of CIC compensators which pre-equalize the passband* in order to compensate for the passband distortion introduced by the CIC filters. Of course, if deemed necessary, this compensators can also be used to compensate for the distortion introduced by linear or hold interpolators.

6.3 CIC interpolators

Let's perform some simple math to derive the CIC filters starting from a hold interpolator. We have seen that a hold interpolator has an impulse response of the form

$$h[n] = \{1,1,1,\ldots,1\}$$

where the number of 1's is equal to the interpolation factor. The transfer function of such a filter is given simply by

$$H(z) = 1 + z^{-1} + z^{-2} + \ldots + z^{-(L-1)} \tag{6.1}$$

If we multiply both sides of (6.1) by z^{-1} we get

$$z^{-1}H(z) = z^{-1} + z^{-2} + \ldots + z^{-L} \tag{6.2}$$

Subtracting (6.2) from (6.1) we get

$$H(z)(1 - z^{-1}) = 1 - z^{-L}$$

which means we can re-write $H(z)$ as

$$H(z) = \frac{1 - z^{-L}}{1 - z^{-1}}$$

If we think of this as two filters in cascade, $H(z) = H_1(z)H_2(z)$, with $H_1(z) = 1 - z^{-L}$ (the "comb" due to its magnitude response) and $H_2(z) =$

* Post-equalize in the case of decimation.

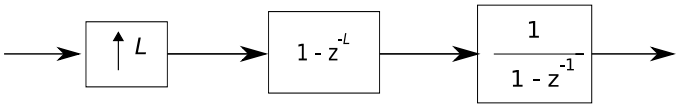


Figure 6.9: *Conceptual implementation of a CIC interpolator. As with all interpolators, conceptually it is just an upsampler followed by a lowpass filter (the combination of the last two filters).*

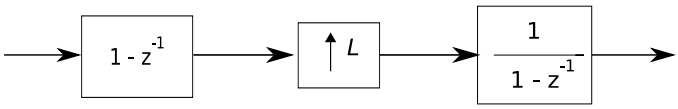


Figure 6.10: *Actual implementation of a CIC interpolator obtained by use of Noble identities.*

$1/(1-z^{-1})$ (the integrator), we have derived a cascaded integrator-comb (CIC) filter.

When used for interpolation*, the CIC interpolator in conceptual form operates as the block diagram shown in Figure 6.9. Using the Noble identities, we can interchange the upsampler and the comb filter to obtain the configuration shown in Figure 6.10.

Of course so far we haven't really done much other than show a rather complicated way of implementing a hold interpolator. In reality, if we wanted to implement a hold interpolator, we would skip the comb and integrator filters and simply repeat the input sample as many times as we wish to interpolate.

The real use of CIC filters comes from a generalization of the fact we already noted relating linear interpolators to hold interpolators. Recall that we stated that a linear interpolator's impulse response can be obtained by convolving two impulse responses of hold interpolators. With CIC filters, we convolve K such impulse responses. Since convolution in time equates to multiplication in frequency, this means that the transfer function of a CIC filter with K sections is given by elevating $H(z)$ to the power of K,

$$H_{\text{CIC}}(z) = (H(z))^K = \left(\frac{1-z^{-L}}{1-z^{-1}}\right)^K$$

If we once again use the Noble identities, we can easily verify that an

* Keeping in mind that interpolation conceptually consists of upsampling followed by lowpass filtering.

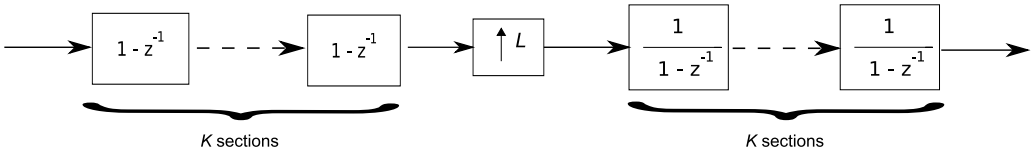


Figure 6.11: *Implementation of a multi-section CIC interpolator.*

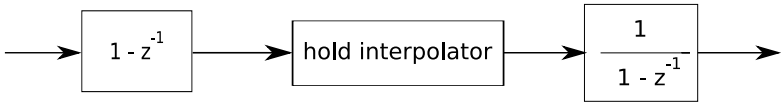


Figure 6.12: *Multiplierless linear interpolator.*

implementation of an K-section CIC interpolator can be done as shown in Figure 6.11.

Note that the following commands can be used to quickly build a Simulink model of a CIC interpolator:

```
L = 4; % Interpolation factor
K = 3; % Number of sections
Hm = mfilt.cicinterp(L,1,K);
realizemdl(Hm)
```

Of course, a multi-section CIC interpolator multiplies the magnitude response of the hold interpolator K times. Because CIC filters are multiplierless, the allure of these filters is that we can obtain good stopband attenuation by using only adders and delays if we use several sections. For this reason, CIC filters are often implemented at the last stage of a multistage interpolation filter, operating at the fastest rate.

The cases $K = 1$ and $K = 2$ are special cases of CIC interpolators that result in hold and linear interpolators. In fact, the inner-most combination of comb/upsampler/integrator, can be readily replaced with a hold interpolator (saving a couple of adders and delays [32]. This technique allows us, for instance, to implement a multiplierless linear interpolator using only 2 adders and 2 delays as shown in Figure 6.12.

6.3.1 Design of CIC interpolators

When designing CIC interpolators, we should think in the context of multistage designs. The design basically consists of determining how many

sections are needed in order to obtain a certain stopband attenuation. Unlike conventional lowpass filters, this stopband attenuation is *not* attained for the entire stopband. Instead, the attenuation will be attained only where needed to suppress remnant high-frequency content left behind by previous stages of interpolation.

Example 50 *As if often the case, an example is the best way to understand this. Suppose we want to interpolate by a large factor, say 64, in order to make our life easy when designing an analog post-filter for D/A conversion. The following is a possible set of specifications for our interpolator:*

```
f = fdesign.interpolator(64,'Nyquist',64,'TW,Ast',0.0078125,80);
```

These specifications state the the cutoff frequency for the interpolator is $\pi/64$ *and that the passband-edge frequency is* $\pi/64 - TW/2 = 0.0117\pi$*. The minimum stopband attenuation for the* entire *stopband region is 80 dB.*

First, let's design this filter using conventional Nyquist filters and let us constrain the design to be done in 3 stages:

```
Hc = design(f,'multistage','Nstages',3);
cost(Hc)
```

The result of this design is a interpolate-by-4 Nyquist filter, followed by a interpolate-by-2 Nyquist (halfband) filter, followed by an interpolate-by-8 Nyquist filter. Overall, the interpolation factor is 64 as desired. The cost of implementing this filter is 110 coefficients and 408 MPIS.

Now, let's try to replace the last stage with an interpolate-by-8 CIC filter. The desired stopband attenuation is 80 dB. However, in order to set the passband-edge frequency, we should use the overall desired passband-edge frequency, i.e. 0.0117π*:*

```
f2 = fdesign.interpolator(8,'CIC',1,'Fp,Ast',0.0117,80);
Hcic = design(f2); % Results in 4 sections
```

The following commands can be used to replace the 3rd stage with a CIC and compare the two multistage designs:

```
Hc2          = copy(Hc);
Hc2.stage(3) = Hcic;
fvtool(Hc,Hc2)
```

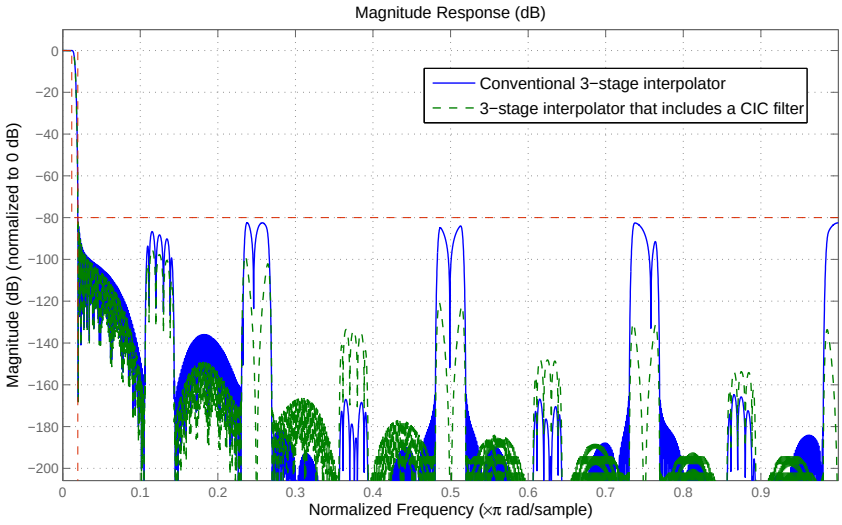


Figure 6.13: *Comparison of a conventional 3-stage interpolation filter and one which uses a CIC interpolator in its last stage.*

The magnitude response of the two implementations is shown in Figure 6.13 (we have normalized the passband gain to 0 dB in order to ease the comparison). Clearly, either filter meets the specs. However, the multistage filter that uses the CIC in its last stage can be implemented with only 70 coefficients and 88 MPIS.

Figure 6.14 shows the magnitude of each of the 3 stages comprising the filter `Hc2`*. The last stage is the CIC filter. Note that the CIC filter does not provide 80 dB attenuation over the entire stopband. However, it does provide 80 dB or more attenuation where needed, i.e. at the points* $2k\pi/L \pm 0.0117\pi$ *where* $k = 1, \ldots, L-1$ *and* $L = 8$.

6.3.2 Gain of CIC interpolators

In the magnitude response plots we have shown so far, we have normalized the passband gain of the filters to 0 dB in order to simplify visualization of the designs.

However, CIC interpolation filters have a very large gain. This goes beyond the usual gain of L that all interpolators have. To see why, recall first the linear interpolator case.

We stated that a linear interpolator can be though of as two hold in-

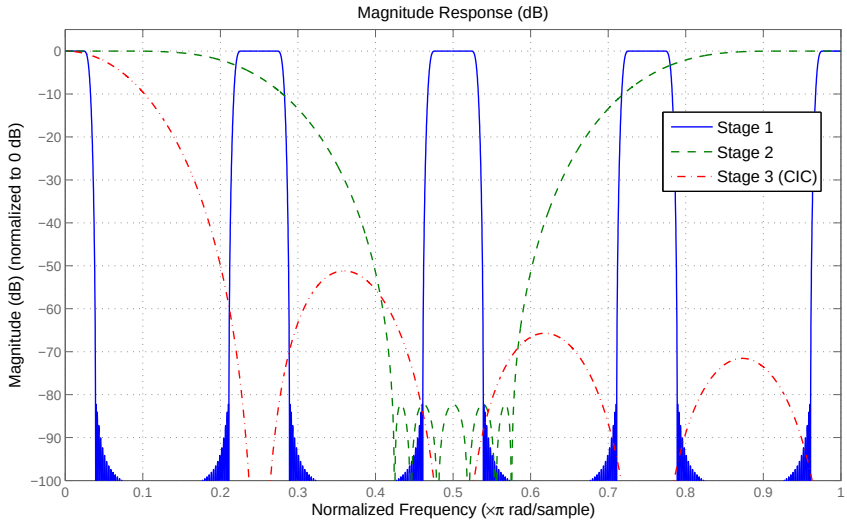


Figure 6.14: *Magnitude response of each stage of a 3-stage interpolation filter using a CIC interpolator in its last stage.*

terpolators convolved. However, in order to get the impulse response of the linear interpolator correctly scaled, when we convolved the two hold interpolators, we had to scale the result by a factor of L as well.

For example, take the case $L = 3$. A hold interpolator has impulse response given by $h[n] = 1,1,1$. A plot of the magnitude response (not in dB) shows that the gain of the filter at frequency 0, is 3. This can easily be verified with the following commands:

```
Hh = mfilt.holdinterp(3);
fvtool(Hh,'MagnitudeDisplay','Magnitude')
```

If we convolve two such impulse responses, we get an impulse response which is 3 times larger than that of the corresponding linear interpolator.

```
conv([1 1 1],[1 1 1])
Hl = mfilt.linearinterp(3);
Hl.Numerator
```

However, the linear interpolator itself has a passband gain of 3:

```
Hl = mfilt.linearinterp(3);
fvtool(Hl,'MagnitudeDisplay','Magnitude')
```

Therefore the convolution of two hold interpolators has a gain of $L^2 = 9$.

Since CIC filters are obtained by (unscaled) convolutions of hold interpolators, their gain is given by L^K. If we account for the fact that we expect any interpolator to have a gain of L, the *extra* gain introduced by a CIC interpolator is L^{K-1}.* For example, for $L = 3$ and $K = 4$, we can easily see that the passband gain is 81:

```
Hcic = mfilt.cicinterp(3,1,4);
fvtool(Hcic,'MagnitudeDisplay','Magnitude')
```

The extra gain, that is L^{K-1}, can be computed simply by using the `gain` command,

```
gain(Hcic)
ans =
    27
```

6.3.3 Further details of CIC filters

Although CIC filters seem pretty nifty, it is worth noting that CIC filters are unstable. The factor $1 - z^{-1}$ in the denominator means that the filter has a pole on the unit circle (at $z = 1$). In fact, the only way for CIC filters to work (meaning that their output will not grow without bound) is by using fixed-point arithmetic (with overflows wrapping).

Also, note that CIC filters have been constructed by adding a pole and a zero at $z = 1$. This pole/zero pair should cancel, yielding the traditional FIR transfer function. However, for implementation, we do not cancel the pole and instead implement the filter with recursion. The filter is still FIR even though it has feedback†

* In our treatment of CIC interpolators so far, we have ignored the differential delay (we have assumed it is one). The differential delay is the order of the delays in the difference (comb) part of the filter. Typical values for the differential delay are 1 and 2. We will assume it is always 1 here. A value different to one will also affect the passband gain. † The strict definition of FIR is a filter whose *minimal realization* has a nil-potent state-transition matrix in its state-space representation (see [26] Ch. 13). The CIC implementation is not minimal since it allows for a pole/zero cancellation.

6.4 CIC decimators

Consider a signal with two-sided bandwidth equal to f_s and sampled at a sampling rate f_s. If we need to decimate by a factor M in order to reduce the rate to a sampling frequency $f_d = f_s/M$, we need to first reduce the two-sided bandwidth of the signal to f_d as well.

CIC filters are attractive for decimation purposes due to their low computational cost given that they do not require multipliers. However, compared to an ideal brick-wall lowpass filter, the response of CIC filter provides a poor lowpass characteristic. Nevertheless, this can be compensated for by the use of subsequent filters that operate at reduced sampling rates and therefore do not incur a high computational cost.

Let us illustrate typical CIC behavior for a decimation problem by a factor $M = 4$. Let us examine Figure 6.15. For this example, we have assumed $f_s = 100$ MHz.

The rectangular areas shown with a dashed line illustrate how the various shifted replicas would overlay had they been filtered with a perfect brick-wall filter. Notice that in this case the various images would be just adjacent to each other but there would be no overlap, i.e. no aliasing.

In contrast, the decimated version of the CIC filtered signal shows significant overlap between spectral replicas. Note that the aliasing is maximum at the edges of the low-rate Nyquist interval, $[-f_d/2, f_d/2]$. On the other hand, the aliasing is minimal at the center of the Nyquist interval (at DC).

For this reason, coupled with the fact that the droop in the CIC filter is not as accentuated in the vicinity of DC, decimating CIC filters are usually used when the ultimate band of interest is not the entire interval $[-f_d/2, f_d/2]$ but rather a (perhaps small) subset of it, say $[-f_p, f_p]$ where $f_p \ll f_d/2$.

It is assumed of course that the energy of the signal outside of the final band of interest, that is, in the band $[-f_d/2, -f_p] \cup [f_p, f_d/2]$ will be removed with either a single-rate lowpass filter, or, more likely, with a subsequent lowpass decimation filter that will reduce the sampling rate to the vicinity of $2f_p$ (this can be done of course with more than one subsequent decimation filter resulting in an overall implementation with perhaps 3 or 4 stages). This subsequent filter will typically compensate for the notable droop in the passband of the CIC filter as well.

It is clear from Figure 6.15, that the maximum amount of aliasing in

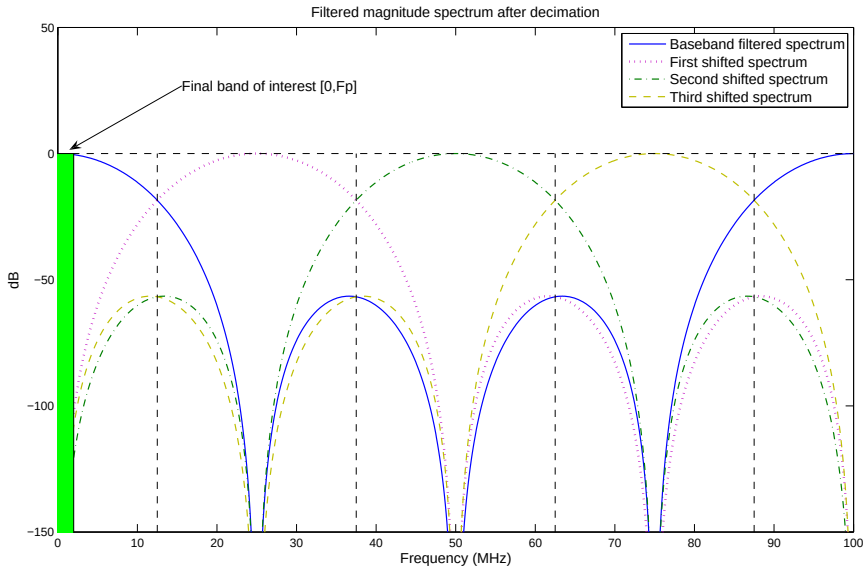


Figure 6.15: *Final band of interest of signal decimated with a CIC filter.*

the final band of interest will occur at f_p. The actual maximum amount of aliasing at f_p is due to the adjacent replica. For a CIC filter, the larger the number of sections K, the smaller the gain for the replica at such frequency, i.e. in order to limit the amount of aliasing, we would like to have a large number of sections. However, the more sections we have, the larger the droop of the filter in the band of interest that we need to compensate for.

In order to design a CIC filter, we reverse the previous statement. Namely, given an amount of aliasing that can be tolerated, determine the minimum number of sections K that are required. The full design parameters are listed next.

6.4.1 Design parameters

At this point we can enumerate the design parameters for the filter. The design parameters are:

- the edge frequency f_p

- the amount of aliasing tolerated at f_p: A_{st}
- the decimation factor M
- optionally, the sampling frequency prior to decimation f_s

As with the design of CIC interpolators, the use of stopband attenuation is slightly different than with conventional lowpass filters. In this case, it is not that the filter must provide a minimum attenuation of A_{st} throughout the entire stopband. Instead, A_{st} is the minimum amount of attenuation permitted for frequencies that will alias back into the band of interest. Of course, at the end of the day, this last interpretation is the same for any decimation filter.

Example 51 *Given these parameters, a CIC decimation filter with the necessary number of sections K such that the amount of aliasing tolerated at $f = f_p$ is not exceeded can be designed as follows:*

```
M    = 8;     % Decimation factor
D    = 1;     % Differential delay
Fp   = 2e6;   % 2 MHz
Ast  = 80;    % 80 dB
Fs   = 100e6; % 100 MHz
Hf   = fdesign.decimator(M,'CIC',D,'Fp,Ast',Fp,Ast,Fs);
Hcic = design(Hf);
```

The resulting design that meets the required attenuation consists of 6 sections.

6.5 CIC compensators

CIC compensators are single-rate or multirate filters that are used to compensate for the passband droop in CIC filters. In the case of CIC interpolation, what is usually done is to pre-equalize for the droop in a prior stage of a multistage design. In the case of decimation, we post-equalize in a subsequent stage to the CIC decimator.

As we have already stated, the amount of droop will depend on the number of sections K in the CIC filter. For the special case of linear and hold interpolators, the number of sections is 2 and 1 respectively.

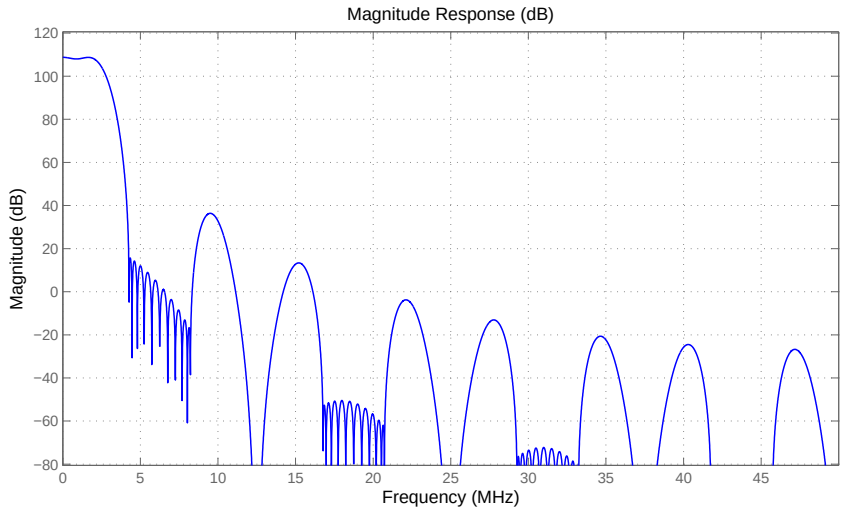


Figure 6.16: *Overall two-stage design consisting of CIC decimator and compensator.*

In the case of decimation, the larger the ratio of the final passband of interest f_p to the original sampling rate f_s, the smaller the amount of droop since only a very small portion of the passband of the CIC filter is involved. In some cases, we can do without the compensation altogether, since the small droop may not warrant equalization.

Similarly, in the case of interpolation, the larger the overall interpolation factor, the smaller the droop will be in the baseband spectrum. Again, it may be so that we choose not to equalize if the overall multistage interpolation factor is large enough.

Example 52 *As an example consider a possible compensator design for the CIC decimator design of the previous section. Suppose we want to further decimate by a factor of 4 for an overall decimation factor of 32 for the two stages.*

```
K   = Hcic.NumberOfSections; % Determine from previous design
M2  = 4;      % Decimation factor for this stage
Fst = 4.25e6; % 4.25 MHz, TW = 2.25 MHz
Ap  = 1;      % 1 dB peak-topeak passband ripple
Hf2 = fdesign.decimator(M2,'ciccomp',D,K,'Fp,Fst,Ap,Ast',Fp,Fst,Ap,Ast,Fs/M);
Hd  = design(Hf2);
```

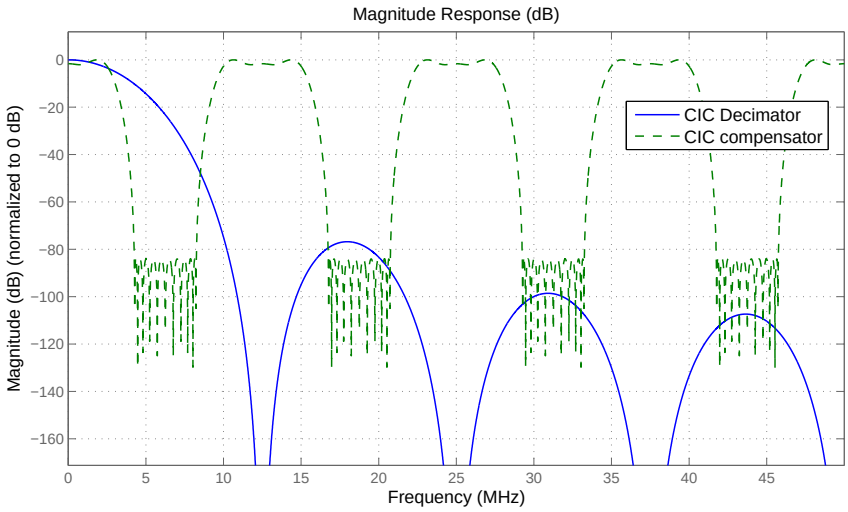


Figure 6.17: *Magnitude response of each of the two stages in the design.*

We can obtain the resulting overall decimator of cascading the two filters from:

```
Hcas = cascade(Hcic,Hd);
fvtool(Hcas) % Show overall response
```

Figure 6.16 shows the resulting filter after cascading the CIC decimator and its compensator. Figure 6.17 shows how the CIC decimator attenuates the spectral replicas from the compensator. Figure 6.18 shows the passband of the overall two-stage decimator. Notice how the droop in the passband has been equalized. As with CIC interpolators, CIC decimators have a large passband gain (for the same reason). The gain can be found from `gain(Hcic)` *and is once again given by* M^K.

In Appendix C, we will compare a multistage decimator design including a CIC decimator and a CIC compensator with various other single- and multistage designs.

6.6 Farrow Filters

We will now derive a filter that can be very effectively used for both fractional advances/delays and changing the sampling rate of signals by arbitrary factors (not necessarily rational).

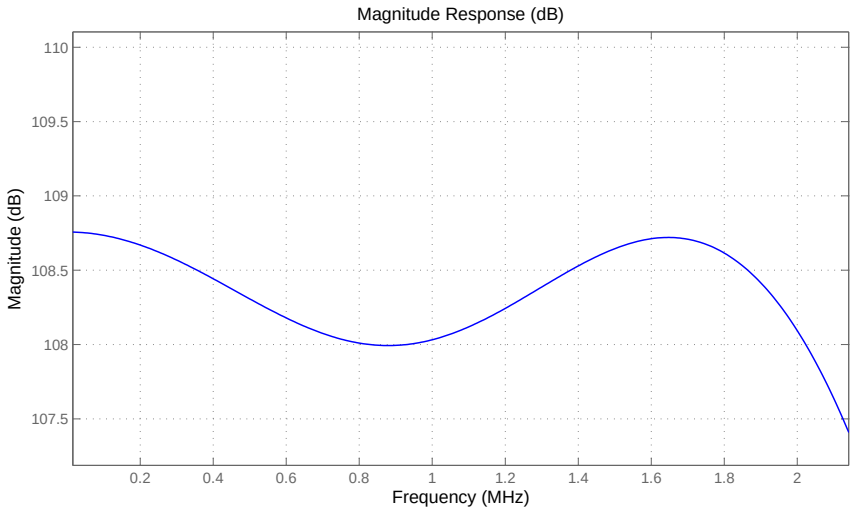


Figure 6.18: *Equalized passband of two-stage design showing the effect of the CIC compensator on the droop.*

Farrow filters are basically obtained by (piece-wise) polynomial interpolation (curve fitting) on the input samples to the filter. Once a polynomial has been fitted, it can be evaluated at any point. This allows for interpolation at arbitrary locations between samples. Moreover, the implementation of Farrow filters is essentially done using Horner's method of polynomial evaluation. The advantage of doing this is that it allows for tuning of the location at which we wish to interpolate by changing only one parameter in the filter. All filter coefficients remain constant. Therefore, Farrow filters are suitable for instance to implement variable fractional advance/delay filters.

Theoretically, polynomials of any order can be used to fit to the existing samples. However, since large order polynomials tend to oscillate a lot, typically polynomials of order 1, 2, or 3 are used in practice.

The easiest way to understand the derivation of Farrow filters is to start with linear interpolators. For the moment, consider the case where we want to interpolate solely for the purpose of introducing a fractional advance/delay (i.e. without changing the sampling rate of the signal).

A linearly interpolated fractional advance/delay can be implemented with two multipliers as we have already seen (this is just what each polyphase

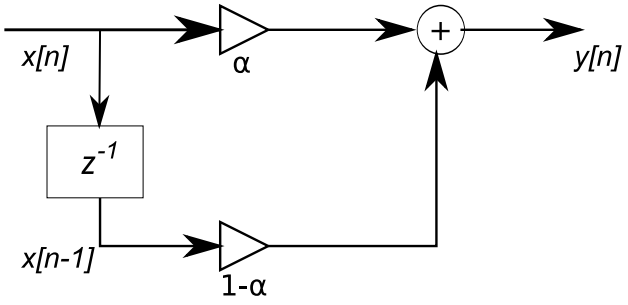


Figure 6.19: *Two-tap filter that can be used for fractional delay by linear interpolation.*

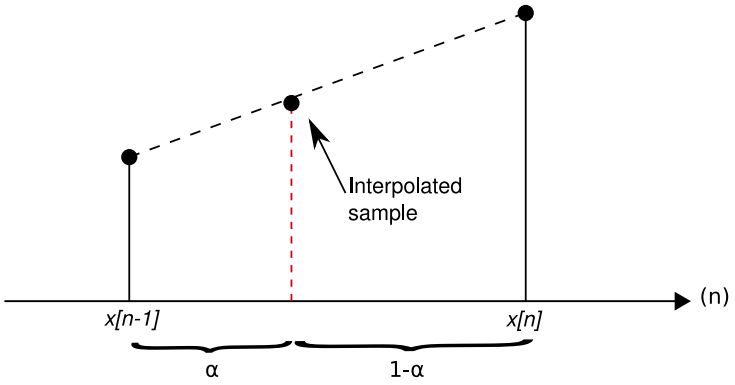


Figure 6.20: *Computing a fractionally advanced/delayed interpolated sample from its two surrounding samples via linear interpolation.*

branch of a linear interpolator does). Consider the two-tap filter shown in Figure 6.19. In order to compute the interpolated values between a previous sample, $x[n-1]$, and a current sample, $x[n]$, we weigh each sample relative to the distance between it and the desired interpolated value. Figure 6.20 shows the resulting interpolated sample.

The filter as shown, performs a fractional delay of $1-\alpha$ (as mentioned before, this is equivalent to a fractional advance of α if we allow for a full sample delay for the sake of causality). The output is computed as

$$y[n] = (1-\alpha)x[n-1] + \alpha x[n]$$

Now, by simply re-writing the previous expression as

$$y[n] = x[n-1] + \alpha(x[n] - x[n-1])$$

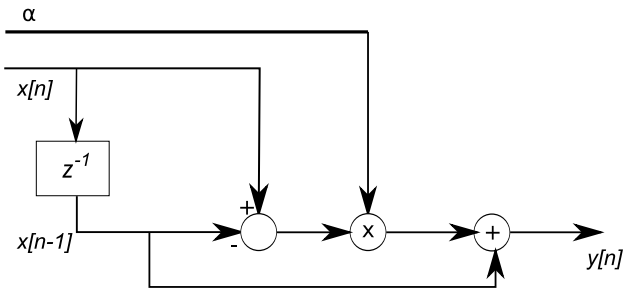


Figure 6.21: *Fractional advance filter.*

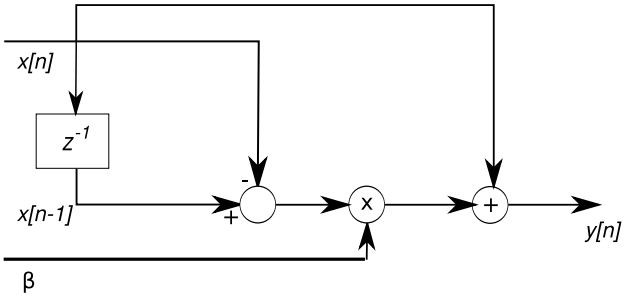


Figure 6.22: *Fractional delay filter.*

we see we can implement the filter as shown in Figure 6.21.

The advantage of this implementation is that there are no fixed multipliers. The desired fractional advance, α, can be thought of as a second input to the filter. It can be tuned at any time, enabling a variable fractional advance/delay.

It is of course trivial to re-wire the filter so that the input is the fractional delay $\beta = 1 - \alpha$ rather than the fractional advance α:

$$y[n] = \beta(x[n-1] - x[n]) + x[n]$$

This is shown in Figure 6.22.

6.6.1 Higher-order polynomials

As usual, better interpolation can be obtained at the expense of more computation. Quadratic and cubic polynomial interpolation are very common cases. Consider the cubic case. For this case, we use two samples to the

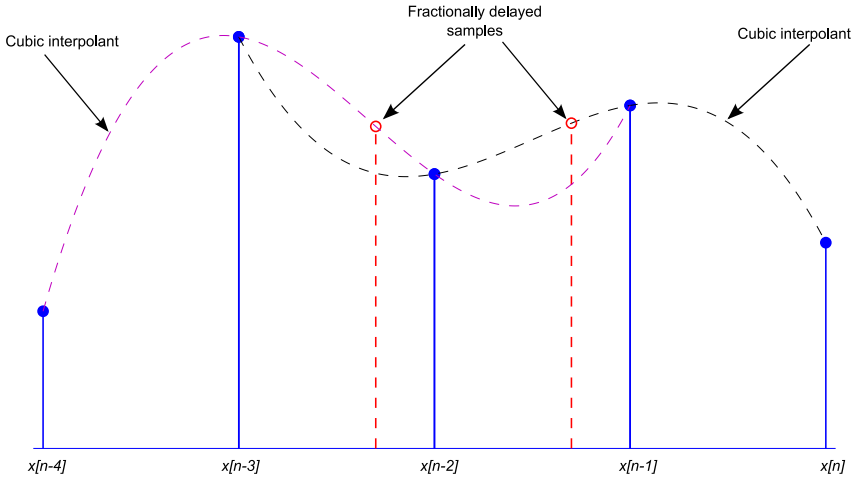


Figure 6.23: *Fractional delay using cubic polynomials.*

left and two samples to the right of where we wish to interpolate.* The situation is depicted in Figure 6.23. Once we have four samples (such as the first four input samples), we can interpolate, but we compute values that lie between the two inner-most samples only (between the second and third from the left). To interpolate, we fit a 3rd-order polynomial to the four samples (the polynomial is unique) and we evaluate the polynomial at the point we wish to interpolate. Then we advance one sample, the leftmost sample is discarded and the right-most sample comes into play. Once again, we fit the unique (new) 3rd-order polynomial to these four samples. Once we have the polynomial, we interpolate by computing the value of the polynomial between the two inner-most samples. After this, we would advance another sample and so on.

As in the linear case, for higher-order polynomials, the Farrow structure makes use of Horner's rule so that all coefficients in the filter are constant while the fractional delay is an input that is tunable at run-time.

* As always, we must allow for sufficient delay in practice in order to make things causal.

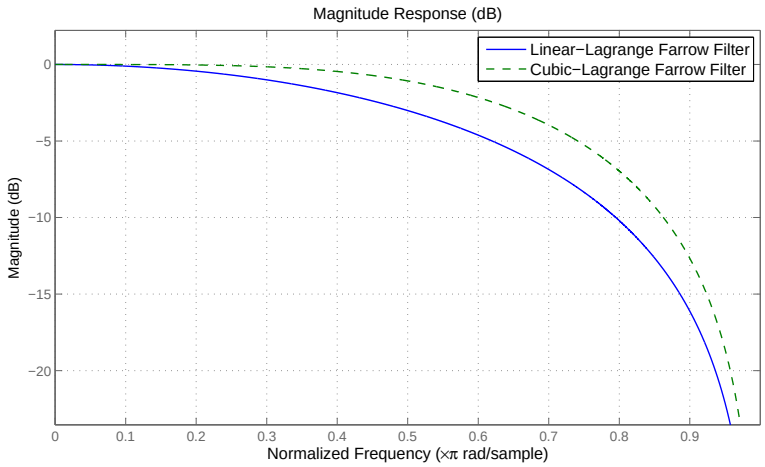


Figure 6.24: *Magnitude response of linear and cubic fractional delays designed via the Lagrange interpolation formula.*

6.6.2 Design of Farrow fractional delays

The design of variable fractional delays using the Farrow structure and Lagrange interpolation boils down to selecting the filter order (the polynomial order). As we have already stated, the linear, quadratic, and cubic cases are the most typical.

It is worth noting though, that for this structure, as the filter order N increases, the number of coefficients increases as N^2. The fractional delay is specified in the design as a starting point. However, as we have already said, the value can be changed at any time while the filter is running.

Example 53 *Let us compare linear and cubic fractional delays using the Lagrange interpolation formula:*

```
del = 0.5; % Desired fractional delay
Hf_lin = fdesign.fracdelay(del,1);
Hf_cub = fdesign.fracdelay(del,3);
Hlin   = design(Hf_lin,'Lagrange');
Hcub   = design(Hf_cub,'Lagrange');
```

As we know, ideal fractional delays should have an allpass magnitude response, and a flat group-delay. Figure 6.24 compares the magnitude responses.

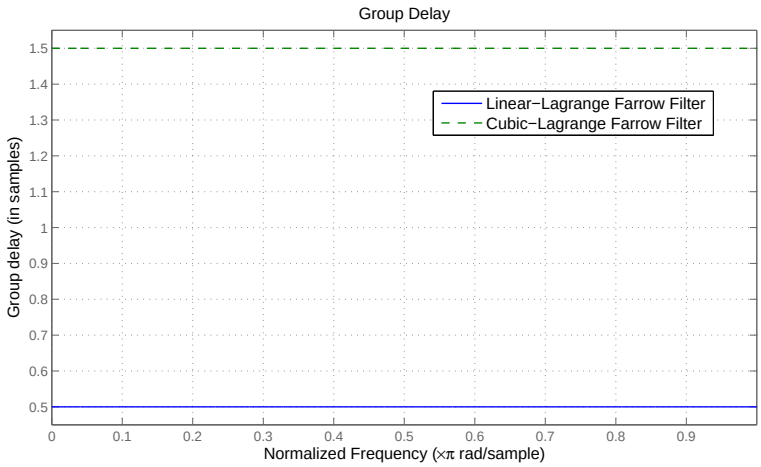


Figure 6.25: *Group delay of linear and cubic fractional delays designed via the Lagrange interpolation formula.*

As can be seen, neither is a particularly good approximation to an allpass filter, however, the cubic design fairs a little better. The group-delays are plotted in Figure 6.25. Both of them are ideal relative group-delays of 0.5. In the cubic case, there is an extra one sample delay introduced by the higher order.

Simulink models for the Farrow filters resulting from these designs can be obtained as usual by using the `realizemdl()` command (e.g. `realizemdl(Hlin)` or `realizemdl(Hcub)` in the Example above).

6.6.3 Multirate Farrow filters

So far we have seen how to implement single-rate fractional delays with Farrow filters. Farrow filters can be used for both increasing and decreasing the sampling-rate of a signal (by either integer or fractional factors). One of the key advantages of Farrow filters is that they can be used to change the sampling-rate of a signal by an irrational factor without adding complexity in terms of number of coefficients. In contrast, the previous polyphase designs we had looked at could only change the rate by a factor given by the ratio of two integers. The best that could be done for irrational factors, was to form an approximation to a rational number of the form L/M. In order to approximate an irrational factor adequately, it

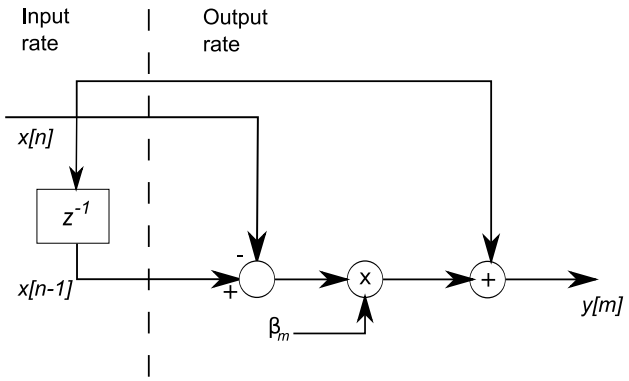


Figure 6.26: *Multirate Farrow filter.*

may be necessary for L and M to be very large integers. As a consequence of this, the number of coefficients required may be quite large.

Nevertheless, it is worth keeping in mind that while the number of coefficients may be kept low by using Farrow structures, the actual number of MPIS will depend on the sampling-rate conversion factor. To see this, let's describe how a multirate Farrow filter works.

Consider the modified diagram for the linear case of a Farrow fractional delay shown in Figure 6.26. The dashed line separates the filter into a section running at the input signal's sampling-rate and a section running at the output sampling-rate. Note that we have re-labeled the output to be $y[m]$ rather than $y[n]$. This is due to different input and output rates. Notably, the fractional delay, now denoted β_m will now *change* at every instant an output sample occurs.

Let's walk through a simple case to get a feel for how this operates. Let's say that we are increasing the sampling-rate by a factor of 2. Since for every input there are two outputs, the value held in the delay register will be used twice. The first time an input is used, β_m will take on the value 0.5 and the output will be computed as

$$y[m] = 0.5(x[n-1] - x[n]) + x[n] = 0.5x[n-1] + 0.5x[n]$$

Before the input sample changes, one more output sample will be computed. β_m will take the value 0 and the output will simply be

$$y[m+1] = x[n];$$

Subsequently, the input sample will change, β_m will be once again set to 0.5 and so forth.

In summary, when increasing the sampling-rate by a factor of two, β_m will cycle between the values $\{0.5, 0\}$ twice as fast as the input, producing an output each time it changes.

In the general case, it is simply a matter of determining which values β must take. The formula is simply

$$\beta_m = \left(\frac{m f_s}{f_s'} \right) \bmod 1$$

where f_s is the input sampling rate and f_s' is the output sampling rate.

Example 54 *Let's say we want* $f_s' = \frac{3f_s}{5}$. *Then* β_m *will cycle through the values* $\{0, 2/3, 1/3\}$. *The following code designs such a multirate filter (for the first-order case) and filters some random data:*

```
% Design first-order multirate Farrow filter
f = fdesign.polysrc(3,5);
f.PolynomialOrder = 1;
H = design(f,'lagrange');
% Filter some random data
x = randn(100,1);
y = filter(H,x);
```

Figure 6.27 shows partial plots of the input and output signals assuming $f_s = 1$. *Notice that the delay follows the values of* β_m *that we have indicated.*

Figure 6.28 shows partial plots for the cubic case rather than the linear case. The improved smoothness of using a cubic polynomial is apparent from the figure.

6.6.4 Polynomial interpolation and maximally flat filtering

In this section we show that using polynomials as we have for piece-wise curve fitting by interpolating $N + 1$ samples using an Nth-order polynomial is equivalent to "traditional" interpolation from the signal processing point of view if we use a maximaly-flat FIR filter as our lowpass filter.

To see this, consider interpolation by $L = 4$. Let's first design a cubic-polynomial Farrow filter to perform such interpolation.

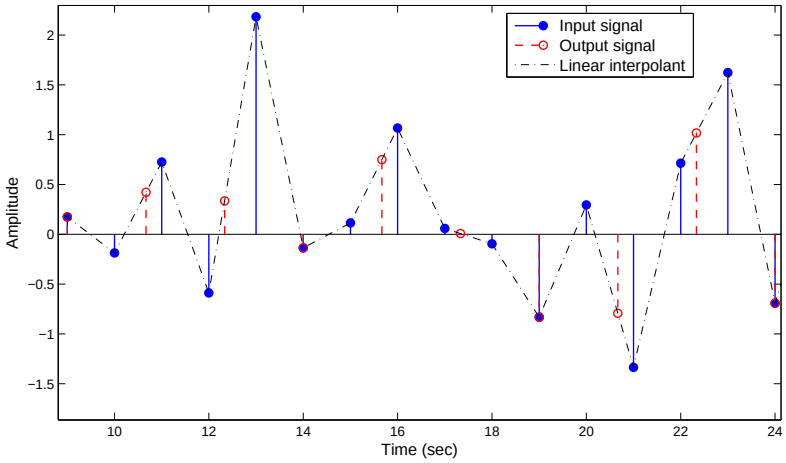


Figure 6.27: *Multirate Farrow filtering using a linear polynomial.*

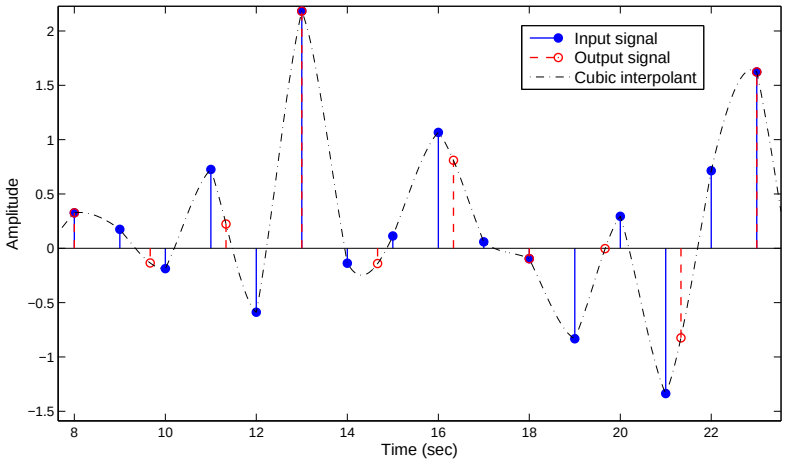


Figure 6.28: *Multirate Farrow filtering using a cubic polynomial.*

```
% Design first-order multirate Farrow filter
L = 4;
M = 1;
N = 3; % Polynomial order
f = fdesign.polysrc(L,M);
f.PolynomialOrder = N;
H = design(f,'lagrange');
```

To interpolate by 4, the fractional delay will take on values of $\{3/4, 1/2, 1/4, 0\}$ in a cyclic manner. If we set the fractional delay of the filter to each of those four values and in each case compute the equivalent transfer function:

```
Hf = fdesign.fracdelay(0.75)
H = design(Hf,'lagrange');
[b0,a0] = tf(H)
H.FracDelay=0.5;
[b1,a1] = tf(H)
H.FracDelay=0.25;
[b2,a2] = tf(H)
H.FracDelay=0;
[b3,a3] = tf(H)
```

It is trivial to verify that `b0,b1,b2,b3` correspond to the polyphase components of the FIR filter returned by the `intfilt()` command:

```
bint = intfilt(L,N,'lagrange');
```

The magnitude response of the filter `bint` is shown in Figure 6.29. We also show the ideal interpolation filter for reference. As previously noted, the maximally flat characteristic has the undesirable effect of a wide transition band as a result. This tells us that polynomial interpolation produces only acceptable filters for sampling-rate conversion applications (and fractional delays). We had already seen in this in a different form in Figure 6.24, where we noted how far from an ideal allpass the fractional delays produced by polynomial interpolation were.

In numerical analysis, better piece-wise curve-fitting is obtained by using techniques such as hermite polynomial interpolation and cubic splines. In signal processing, typically we improve the interpolation by using filters with better transition-width than maximally-flat filters.

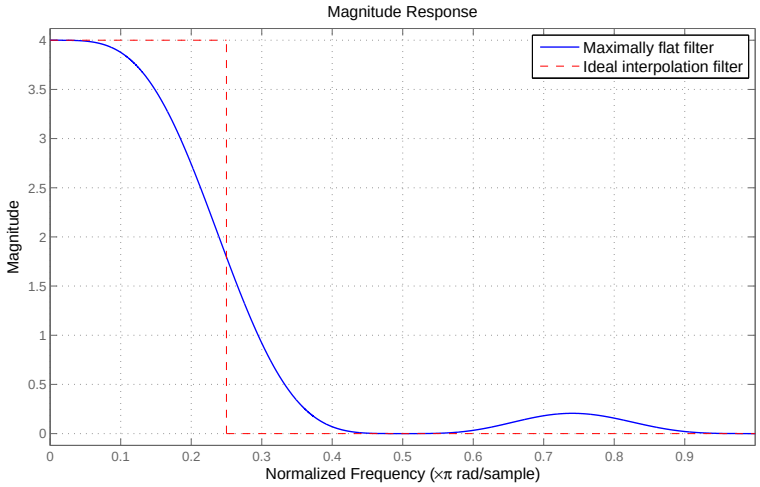


Figure 6.29: *Magnitude response of a maximally-flat interpolate-by-4 filter corresponding to cubic polynomial interpolation.*

6.6.5 Using Farrow sample-rate converters in multistage designs

We have seen how multistage/multirate designs can result in significantly reduced implementation cost when compared to single-stage cases. However, multistage techniques are applicable only when the rate change factor is not a prime number. However, if we need to change the rate by a prime number, we need not be "stuck" with a single-stage design. One possibility, is to change the rate by a non-prime number near the prime number we are looking for, and use a Farrow filter to change the final rate so as to obtain the desired overall rate-change.

Example 55 *Suppose we have a signal sampled at 680 kHz, but the band of interest extends to 17 kHz only. We may have oversampled as signal as part of analog to digital conversion for the purpose of simplifying the anti-aliasing analog filter. We wish to reduce the bandwidth and the rate by decimating. The ideal decimation factor would be $M = 17$. However, since 17 is a prime number, we can only design a single-stage decimator such as:*

```
Fs  = 680e3; % 680 kHz
Fp  = 17e3;  % 17 kHz
```

```
TW  = 6e3;   % 6 kHz
Ast = 80;    % 80 dB
M   = 17;    % Decimation factor
Hf  = fdesign.decimator(M,'Nyquist',M,'TW,Ast',TW,Ast,Fs);
Hd  = design(Hf,'kaiserwin');
cost(Hd)
```

The cost of this design is of about 543 coefficients and 32 MPIS.

An alternative strategy may be to design a multistage filter that decimates by 16. In fact, due to the wide transition band of the Farrow filter that follows, we may choose to have a smaller transition band here, say half the transition width:

```
Hf = fdesign.decimator(16,'Nyquist',16,'TW,Ast',TW/2,Ast,Fs);
Hmult = design(Hf,'multistage')
cost(Hmult)
```

The result is a multistage design with 4 halfbands in cascade. The cost is of about 92 coefficients and 10 MPIS.

In order to reach the overall decimate-by-17 factor, we can design a cubic-polynomial Farrow filter to perform a rate-change by a factor of 16/17:

```
f = fdesign.polysrc(16,17);
H = design(f,'lagrange');
```

We can cascade this filter after the multistage decimator to achieve the goal we want:

```
Hcas = cascade(Hmult,H);
```

Part II

Filter Implementation

Chapter 7

Implementing FIR Filters

7.1 Some basics on implementing FIR filters

There are several factors that influence the selection of the filter structure used to implement an FIR filter. If the filter length is very large, it may be preferable to use a frequency-domain implementation based on the FFT. If it is to be implemented in the time domain, the target hardware is an important factor.

From a design perspective, there is an important distinction between minimum-phase and linear phase filters. Minimum-phase FIR filters do not have any symmetry in their coefficients while linear phase FIR filters have either symmetric or antisymmetric coefficients. Depending on the target hardware, it may be possible to implement a linear-phase FIR filter using less multipliers than the minimum-phase filter by taking advantage of the symmetry even if the filter length of the linear-phase is larger. In other words the implementation advantages of linear-phase filters may offset the gains of a minimum-phase design making it preferable to use a linear-phase filter even when linearity of phase is not critical for the application at hand. Of course there are other reasons to use minimum-phase filters that may give a compelling reason to do so. For instance, as we have seen, minimum-phase filter introduce a minimal transient delay. This may be important enough to stick with a minimum-phase design.

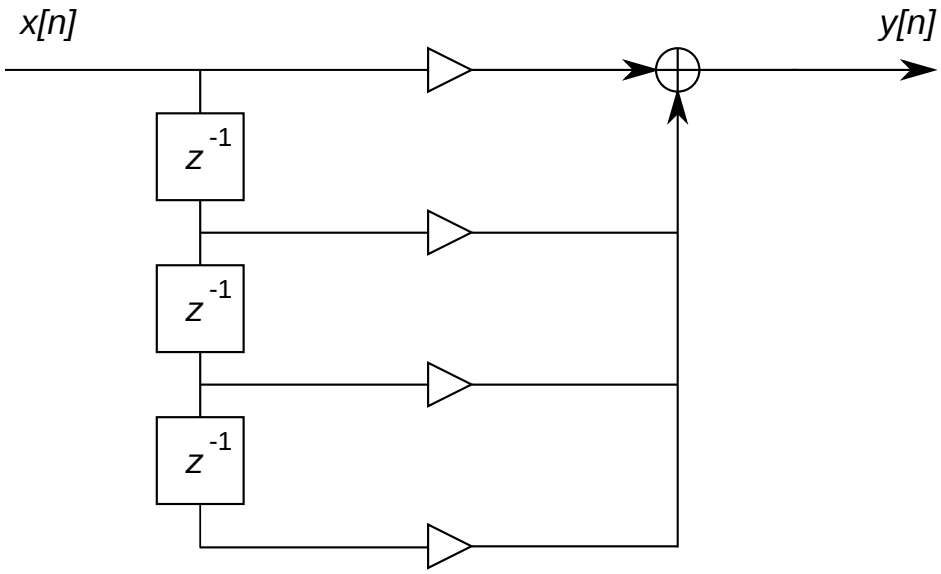


Figure 7.1: *A 4-tap FIR filter implemented in direct-form.*

7.1.1 Direct-form filter structure

Consider the design of an FIR filter,

```
Hf  = fdesign.lowpass('N,Fp,Fst',3,.3,.8);
Heq = design(Hf,'equiripple','FilterStructure','dffir');
```

The filter order is very low for illustration purposes. The filter structure that is used to implement the filter is specified by the string `'dffir'` (which is the default) and corresponds to the so-called direct-form structure (also called a tapped delay line). To visualize the structure we can create a Simulink block for the resulting filter using the `realizemdl` command,

```
realizemdl(Heq);
```

The resulting block is a subsystem. If we look inside the subsystem, we will see the structure shown in Figure 7.1.

The number of delays is equal to the filter order and the number of coefficients (number of taps), which is one more than the number of delays, determines the filter length.

The structure has some regularity in that a sample is read from memory (a delay is simply a register), multiplied with a filter coefficient, and accumulated to form the output. DSP processors have historically been built with this multiply-accumulate (MAC) instruction in mind. The structure requires a shift of the input data throughout all delays for each sample. To minimize the amount of data copies, circular buffers are commonly used [3].

A downside of the direct-form structure is that is does not take advantage of the symmetry (or antisymmetry) in the coefficients of linear-phase filters. So the cost of using this filter structure is maximum in terms of the number of multipliers required as can be seen using `cost(Heq)`. Nevertheless, many DSP processors have been architectured to implement this structure efficiently, so that it should be used even for linear-phase filters on that hardware. Of course the structure is well-suited for filters that do not present symmetry in their coefficients such as minimum-phase FIR filters.

7.1.2 Symmetric direct-form filter structure

The design shown in the previous section was a linear-phase design. Since the number of coefficients is four, there are two unique multipliers. The filter can be implemented using only two multipliers by using the symmetric direct-form structure.

```
Hsym = design(Hf,'equiripple','FilterStructure','dfsymfir');
realizemdl(Hsym);
cost(Hsym)
```

The filter structure obtained in Simulink by using the `realizemdl` command is shown in Figure 7.2. In general, the number of multipliers required is half* of the number required for the direct-form structure. Notice that even though there are only two multipliers, there are still three delays (same as for the direct-form structure) required since the number of delays corresponds to the order of the filter.

* Rounded up if the number of coefficients is odd.

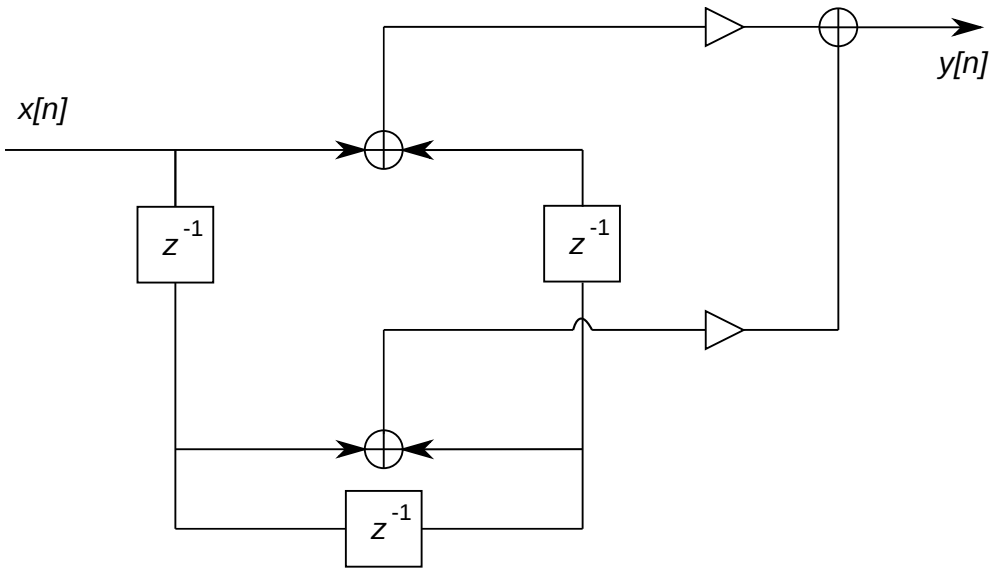


Figure 7.2: *A 4-tap symmetric FIR filter implemented using the symmetric direct-form filter structure.*

7.1.3 Transposed direct-form filter structure

The direct-form structure has the disadvantage that each adder has to wait for the previous adder to finish before it can compute its result. For high-speed hardware such as FPGAs/ASICs, this introduces latency which limits how fast the filter can be clocked.

A solution to this is to use the transposed direct-form structure instead. With this structure, the delays between the adders can be used for pipelining purposes and therefore all additions/multiplications can be performed in fully parallel fashion. This allows real-time handling of data with with very high sampling frequencies.

The transposed structure for a 4-tap filter is shown in Figure 7.3. It can be generated using the `realizemdl` command on a filter whose filter structure has been selected to be transposed direct-form.

```
Htran = design(Hf,'equiripple','FilterStructure','dffirt');
realizemdl(Htran)
```

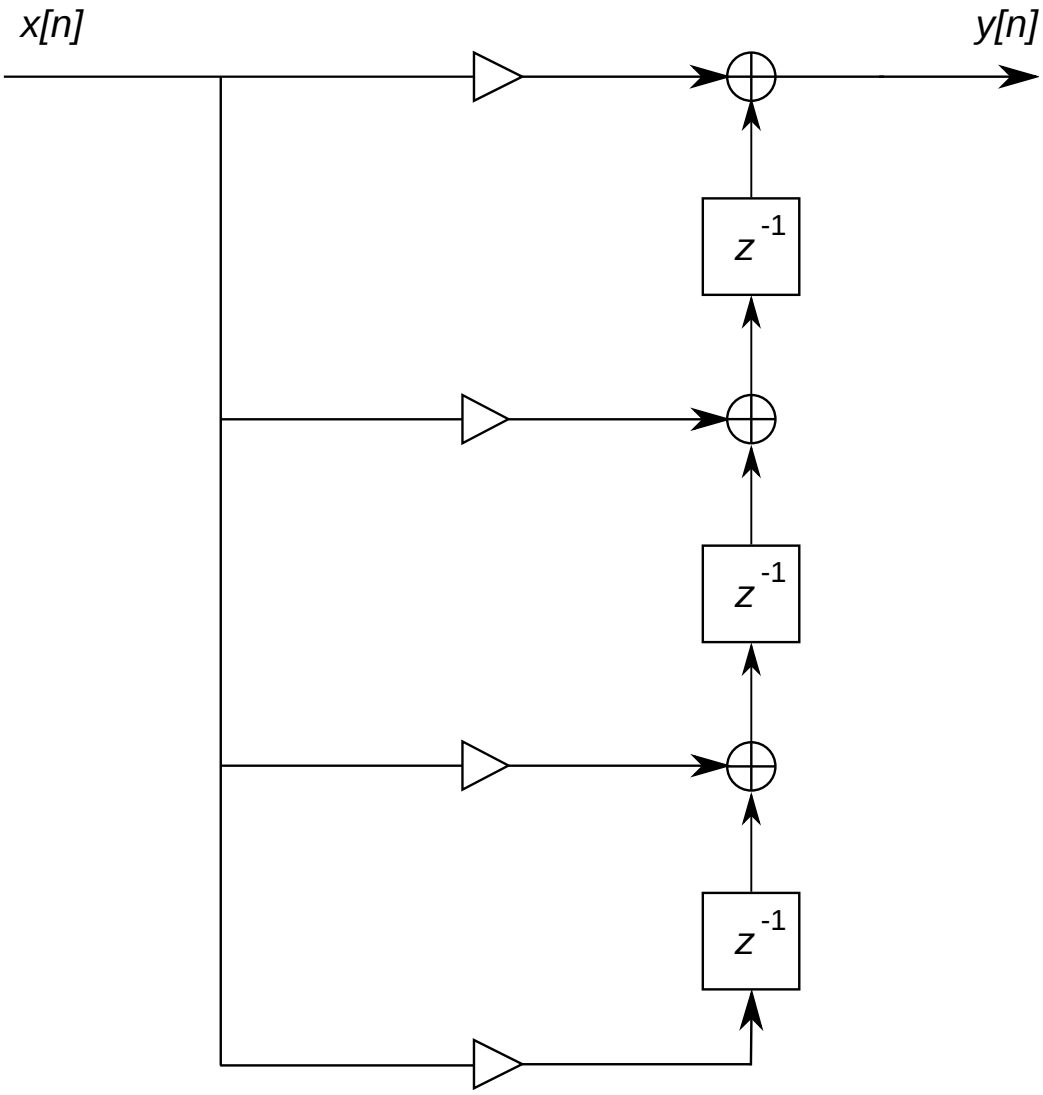


Figure 7.3: *A 4-tap FIR filter implemented in transposed direct-form.*

7.2 Fixed-point implementation

Implementing an FIR filter using fixed-point arithmetic is generally a rather simple task. Some care must be taken when quantizing the coefficients so that we use enough bits to achieve the desired stopband attenuation of

the filter and we scale the bits in the most beneficial manner (more on this below).

Once the coefficients have been quantized, and given a fixed-point input signal, it is easy to figure out what it would take to perform all multiplications/additions within the filter in a such a way that no roundoff error is introduced (so-called full-precision mode).

With all arithmetic being performed with full precision, there will be bit growth associated with filtering. In most cases, the output signal cannot have the full wordlength resulting from this bit growth. We must throw out bits at the output, usually to match the wordlength of the input signal. To throw away bits, we simply remove LSBs. This will introduce some quantization error at the output as we will see below. No overflow will occur by throwing out LSBs.

Appendix D provides a brief review of fixed-point concepts that may be useful prior to reading through the rest of this Chapter.

7.2.1 Quantizing the filter's coefficients

The filter coefficients have to be quantized from double-precision floating point in which they are designed into fixed-point representation with usually a smaller number of bits. We must make sure we make the most of the limited number of bits we have.

The first thing to do is check if there are enough bits available to cover the required dynamic range. When we quantize the impulse response of an FIR filter, we should not expect to achieve the full 6 dB/bit*. This is not surprising given the sinc-like shape of most FIR impulse responses. Most of the values of the impulse response are small, so that the average signal strength does not cover the available range.

Example 56 *Let's compute the SNR when quantizing the impulse response of an FIR filter using 16 bits to represent the coefficients:*

```
Hf    = fdesign.lowpass(0.4,0.5,0.5,80);
Hd    = design(Hf,'equiripple');
himp  = Hd.Numerator; % Non-quantized impulse response
Hd.Arithmetic = 'fixed'; % Uses 16-bit coefficients by default
himpq = Hd.Numerator; % Quantized impulse response
```

* See Appendix D.

```
SNR   = 10*log10(var(himp)/var(himp-himpq))
SNR =
   84.3067
```

When we quantize the impulse response of an FIR filter, we may use the additive noise model to get an idea of how the frequency response is affected. If the impulse response is given by $h[n]$, the quantized impulse response is

$$h_q[n] = h[n] + e[n]$$

The additive noise, $e[n]$, will affect the frequency response of the quantized impulse response by adding a noise-floor of intensity $\varepsilon^2/12$, distorting both the passband and the stopband of the filter. In most cases, the effect on the stopband is most critical, reducing the attainable stopband attenuation of the filter.

A good rule-of-thumb is that we can expect about 5 dB/bit when quantizing an FIR filter, as long as we use the bits wisely. By this we mean, scale them so that impulse response is representable with the bits we have without wasting any bits.

The range of values that can be represented with a given wordlength/fraction length combination falls between a negative and a positive power of two (for instance the intervals [-0.125,0.125), [-0.5,0.5), [-1,1),[-8,8), etc)*. We want to choose the smallest interval that contains the largest (absolute) value of the impulse response. If we choose an interval larger than the minimum, we are just wasting bits. If the largest value of the impulse response is positive and a power of two, then it is somewhat of a judgement call whether or not we want to allow for a small overflow error for that one coefficient.

Example 57 *Consider the following equiripple Nyquist design:*

```
f = fdesign.nyquist(4,'TW,Ast',0.2,80);
h = design(f,'equiripple');
```

The filter has a minimum attenuation of 80 dB. Its largest coefficient is 0.25.

Using the 5 dB/bit rule, we need at least 16 bits in order to provide the 80 dB of attenuation. Since the largest coefficient is 0.25, we can choose a fractional

* Note the open interval on the right. That's because we cannot represent exactly that value. How close we get will depend on the number of bits available.

length of 17 to scale the bits so that all values in the interval [-0.25,0.25) are representable. In this example we choose to live with small overflow in the quantization of the largest coefficient (we quantize the value to $0.25 - 2^{-17}$*). Since the value is a power of two, depending on the hardware, another option would be not to implement this coefficient as a multiplier at all (instead simply perform a bit shift). This is a nice property of all Nyquist filters, the largest coefficient is always a power of two.*

In order to set a 16-bit wordlength and a 17-bit fraction length, we perform the following steps:

```
h.Arithmetic     = 'fixed'; % Uses 16-bit wordlength by default
h.CoeffAutoScale = false;
h.NumFracLength  = 17;
```

Note that there is automatic scaling of the coefficient bits by default. It is designed to avoid overflow in the quantization while minimizing the interval so that the bits are used as best possible. As we have said, strictly speaking the quantization of the coefficient equal to 0,25 overflows with a fraction length of 17, so a fraction length of 16 is used by default (which along with the wordlength, means any value in the interval [-0.5,0.5) is representable without overflow).

The magnitude response of the quantized filter is shown in Figure 7.4. * *The 16 bits are adequate in order to preserve quite faithfully the intended magnitude response of the filter.*

To emphasize the point regarding the need to use both the right number of bits and use them wisely, consider what would have happened in the previous example if instead of a fraction length of 17, we used a fraction length of 15. The magnitude response for this case is shown in Figure 7.5. Notice that the quantized filter no longer meets the required 80 dB stopband attenuation (the passband also has greater error than in the previous case).

The reason we can no longer meet the stopband attenuation is that we are essentially wasting two bits by using a fraction length of 15. The maximum roundoff error (assuming we round to the nearest quantized value) can be as a large as 2^{-16} as opposed to 2^{-18} if we had used a fraction

* Note that this analysis along with most others (including the impulse response) only takes into account the quantization of the coefficients. It does not take into account the fact that there may be roundoff/overflow introduced by the multiplications/additions when actually filtering data.

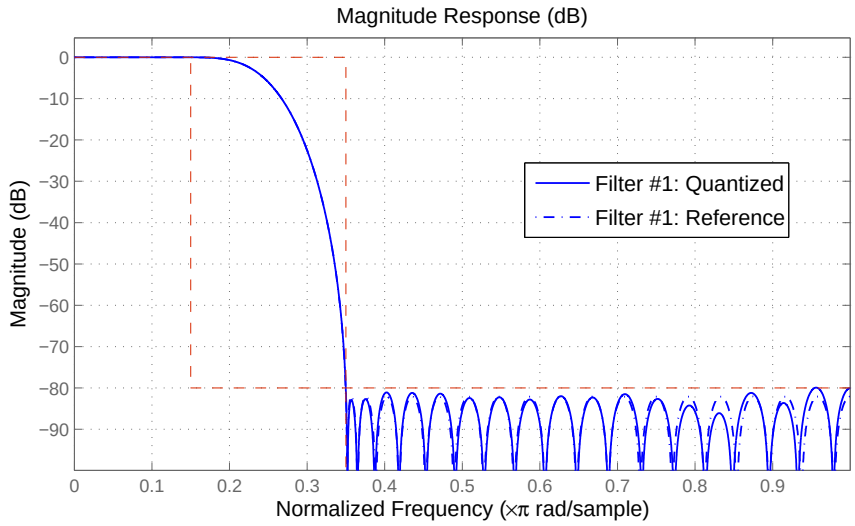


Figure 7.4: *Magnitude response of equiripple Nyquist filter quantized with 16-bit wordlength and a fraction length of 17.*

length of 17. This increase in roundoff error results in raising the quantization noise floor in the frequency domain, reducing the attainable minimum stopband attenuation.

Redundant bits in smaller coefficients

So far, we have assumed that the same number of bits are used to represent each coefficient of an FIR filter. Because usually many of the coefficients are quite smaller than the largest coefficient, many of the bits used for these small coefficients are redundant and, depending on the hardware used to implement the filter, could be removed without affecting the performance.

Example 58 *Let's look again at what happens when we quantize the following filter:*

```
Hf = fdesign.lowpass(0.4,0.5,0.5,80);
Hd = design(Hf,'equiripple');
Hd.Arithmetic = 'fixed';
B = fi(Hd.Numerator,true,16,16);
```

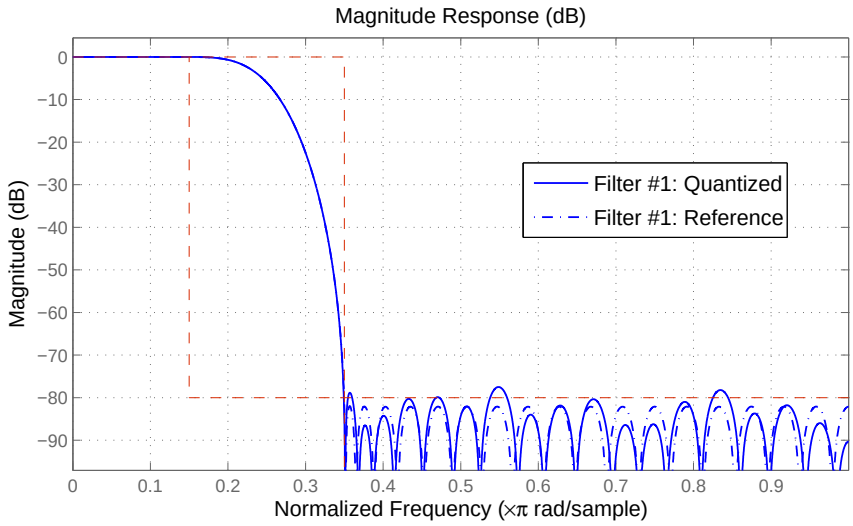


Figure 7.5: *Magnitude response of equiripple Nyquist filter quantized with 16-bit wordlength and a fraction length of 15.*

The variable `B` *contains the fixed-point numbers corresponding to the filter's coefficients. Because the largest value of the coefficients is about 0.434, they have been quantized with a fractional length of 16 (the default wordlength is in turn 16 bits which is in-line with the requirement for this filter following the 5dB/bit rule).*

Now let's look at the binary representation of say the first three coefficients:

```
B.bin
ans =
1111111111011100   1111111110100110   1111111111010100   ...
```

The repeated 1's towards the left (the MSB) are all unnecessary in order to represent the value. Indeed, we could represent the first value simply as 1011100, the second value as 10100110, and so forth. So in reality we need only 7 bits to represent the first coefficient, 8 bits to represent the second coefficient, etc.

Indeed, note that 1111111111011100 with a fractional length of 16 corresponds to the value -0.00054931640625. But this number can be represented without loss of precision using only 7 bits:

```
C1 = fi(-0.00054931640625,true,7,16)
C1.bin
```

The full 16 bits are used for the largest coefficients. For instance for the middle one (the largest one):

```
Hd.Numerator(30)
ans =
    0.4339599609375
C30 = fi(0.4339599609375,true,16,16);
C30.bin
ans =
    0110111100011000
```

which of course has no redundant bits.

The fact that we have redundant bits in many coefficients is another way of understanding why we cannot achieve the full 6 dB-per-bit when quantizing FIR filter coefficients. Effectively we don't use all bits available to represent most of the coefficients, thus the SNR decreases.

With certain hardware, it is possible to take advantage of the fact that many coefficients have redundant bits by using smaller multipliers (in terms of the number of bits) for these coefficients.

7.2.2 Fixed-point filtering: Direct-form structure

Consider an FIR filter implemented in direct form. Assume the input signal has been quantized already and the filter coefficients have been quantized as well. It is possible to easily perform all additions and multiplications within the filter structure in such a way that no roundoff error is introduced.

If the number of bits used to represent the output is the same as the number of bits used for the additions (the accumulations) then no roundoff error is introduced throughout the filter. This so-called full precision mode represents the best possible result we can achieve when fixed-point filtering with an FIR filter implemented in direct form. The only quantization error is due to the coefficient quantization (the quantization of the input signal is considered separately as it is not affected by what happens within the filter itself.

No overflows will occur in full precision mode because we assume we will grow enough bits when adding (see below) to accommodate for signal levels increasing.

Full precision products

In Appendix D, we saw that given two fixed-point numbers, one with wordlength/fraclength given by $\{B_1, F_1\}$ and the other with wordlength/fraclength given by $\{B_2, F_2\}$, the product of the two numbers can be represented with no roundoff by using a wordlength/fraclength of $\{B_1 + B_2, F_1 + F_2\}$.

For a direct-form FIR filter to be implemented with full precision, this means that there will be bit growth once the input signal is multiplied with the filter's coefficients (see Figure 7.1).

Full precision additions

In Appendix D, we saw that in order to perform a series of N additions without introducing round-off error, it is necessary to allow for a bit-growth of $\lfloor \log_2(N) \rfloor + 1$ bits. In an FIR filter implemented in direct form, there are a series of additions that occur after multiplying the input signal with the coefficients (see Figure 7.1).

Example 59 *Consider once again the following design:*

```
Hf = fdesign.lowpass(0.4,0.5,0.5,80);
Hd = design(Hf,'equiripple');
Hd.Arithmetic = 'fixed';
```

The coefficients are represented with 16 bits. If the input is represented with 16 bits as well, full precision multiplications would mean that the values to be added are represented with 32 bits. Since there are 58 additions to be performed, the number of bits we need in order to perform the additions with full precision is

$$\lfloor \log_2(58) \rfloor + 1 = 6$$

As it turns out, it is possible to perform the additions with full-precision by using even less bits than what it is specified by the formula $\lfloor \log_2(N) \rfloor + 1$. We will look at how to do so next.

Full precision multiplications/additions given the actual coefficient values

Let's look again at the previous example to see if we actually need 32 bits for the full-precision multiplications. The largest coefficient of the filter `Hd`

is 0.434. The coefficients are represented with $\{B_1, F_1\} = \{16, 16\}$ and the input is assumed to be represented with $\{B_2, F_2\} = \{16, 15\}$, i.e., the input signal is assumed to fall within the interval $[-1, 1)$.

The fraclength for the product is given by $F_1 + F_2 = 31$. Since the largest possible value that any input value multiplied by any coefficient can produce is about 0.434, the interval used to represent the products must be $[-0.5, 0.5)$. This interval is covered by making the wordlength equal to the fraclength. Since the fraclength is 31, the wordlength needed for full-precision multiplications is also 31, not 32.

Now let's move to full-precision additions. Let's take a look at the convolution equation for a length $N + 1$ FIR filter,

$$y[n] = \sum_{m=0}^{N} h[m]x[n-m]$$

Given that we should know what are the maximum values that the input can take, we want to determine what are the maximum numbers the output can take. To do so, we use some simple math. We start by taking the absolute value at both sides of the convolution equation,

$$|y[n]| = \left| \sum_{m=0}^{N} h[m]x[n-m] \right|$$

Using the triangle inequality for the 1-norm, we have

$$\begin{aligned} |y[n]| &\leq \sum_{m=0}^{N} |h[m]x[n-m]| \\ &= \sum_{m=0}^{N} |h[m]|\,|x[n-m]| \end{aligned}$$

Let's say that the input covers the range $[-R/2, R/2)$. This means that in the worst case,

$$|y[n]| = \frac{R}{2} \sum_{m=0}^{N} |h[m]| = \frac{R}{2} \|h[n]\|_1$$

Thus the 1-norm of the impulse response provides a gain factor for the output relative to the input. For example, if $R/2 = 1$, the maximum, value

the output can take is given by $\|h[n]\|_1$. This tells us how many bits we need to grow in order to perform full-precision arithmetic throughout the filter.

Example 60 *For the 59-tap filter* `Hd` *we have been using, we can compute the 1-norm of its impulse response as follows:*

```
norm(Hd,'l1')
ans =
    1.9904
```

This means that we need to ensure the additions can represent the range $[-2,2)$. *Since the multiplications fall in the interval* $[-0.5,0.5)$, *it is necessary to add two bits in order to implement a full-precision filter. If we look at the values for* `AccumWordLength` *and* `OutputWordLength` *when we write* `get(Hd)`, *we see than indeed these values have been set to 33 bits, i.e. two more bits than what is used for the multiplications.*

We have seen how we determine the number of bits required for full-precision implementations. As a matter a fact, this value is still somewhat conservative because it assumes the worst possible case for the input. If we were to actually take into account typical values we think the input can assume, we may even be able to reduce the required number of bits further.

7.2.3 Fixed-point filtering: Transposed direct-form structure

In order to perform all operations with full precision, the settings for the transposed direct-form structure shown in Figure 7.3 are almost identical to the non-transposed direct-form case.

The only difference has to do with the wordlength necessary for the state registers. In the direct-form case, the states needed to have the same wordlength as the input signal in order to avoid roundoff. In the transposed structure, the states must have the same wordlength as the wordlength used for addition. This is typically more than twice the wordlength of the input, so that the transposed structure clearly has higher memory requirements for a given filter.

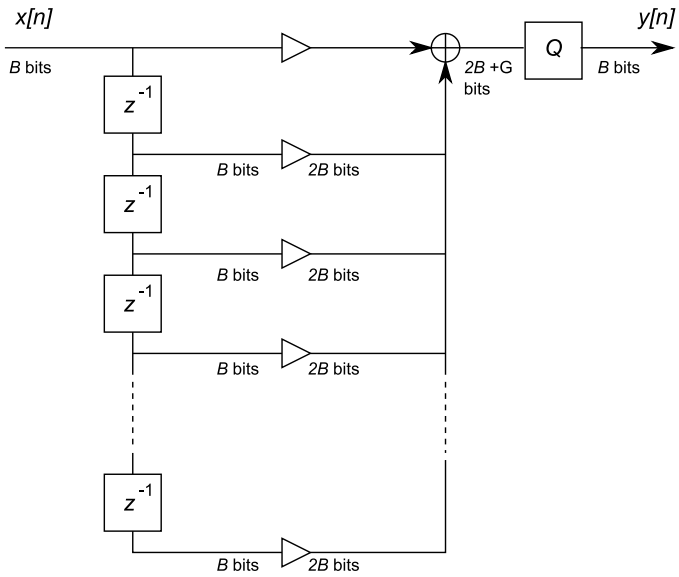


Figure 7.6: *Direct-form FIR filter with quantization at the output.*

7.2.4 Quantization of the output signal

In many cases, we need to reduce the number of bits at the output of the filter for downstream processing. A common case is to use the same number of bits for the output as we did for the input. In the example above, the input was represented with 16 bits, but the full-precision output required 33 bits. This means we may need to discard* 17 LSBs at the output of the filter. The situation is depicted in Figure 7.6 for the direct-form structure. The idea is the same for other structures. In that figure, we assumed that the coefficients are represented with the same number of bits as the input and all multiplications/additions are performed with full precision. G is the number of bits that need to be added for the additions to be performed without round-off as discussed above.

As discussed in Appendix D, the quantization can be modeled as additive white noise with variance $\varepsilon^2/12$ as long the number of bits discarded is relative large (more than 3). For the direct-form structure, this means

* Note that by discarding, we do not necessarily mean simple truncation. It is possible that when we discard, we perform some sort of rounding such as round to nearest. This is controlled with the RoundMode setting once the filter internals have been set to specify the precision.

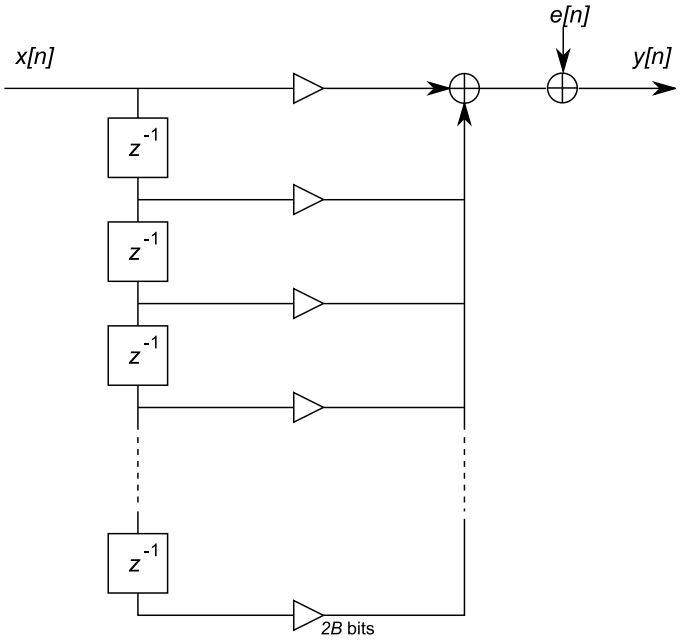


Figure 7.7: *Direct-form FIR filter with quantization at the output.*

we can replace the quantizer with an adder as shown in Figure 7.7.

In the example above, the full-precision version of `Hd` had 33 bits at the output and a fractional length of 31. If we discard 17 LSBs, we will have 16 bits at the output and a fractional length of 14:

```
Hd.FilterInternals  = 'specifyPrecision';
Hd.OutputWordLength = 16; % 33-16 = 17
Hd.OutputFracLength = Hd.OutputFracLength-17; % 14
```

The variance of the quantization noise that we introduced will be given by:

```
2^(-14)^2/12
ans =
   3.1044e-10
```

We can also find this value by integrating the power-spectral density of the output signal due to the noise signal $e[n]$ by using (D.4). Note that the

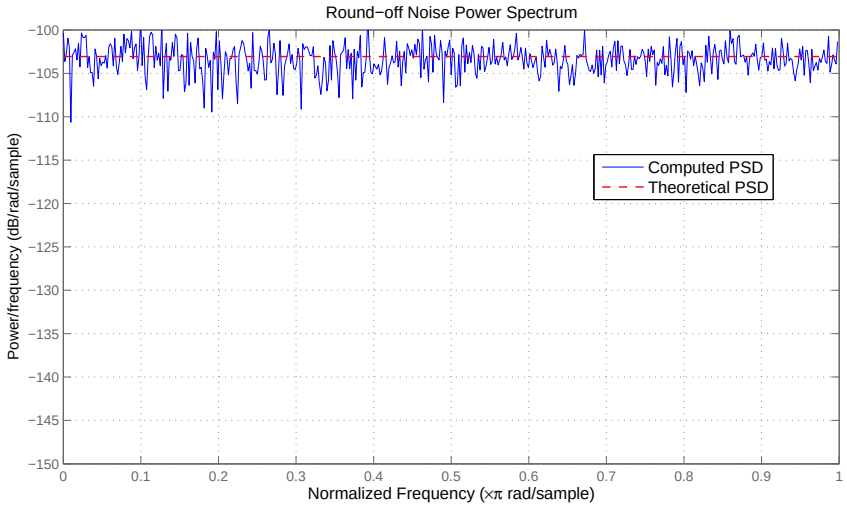


Figure 7.8: *Power spectral density of the quantization noise at the output of an FIR filter.*

transfer function between the noise input, $e[n]$, and the output is simply $H_e(z) = 1$. The integral of the PSD gives the average power or variance:

```
P = noisepsd(Hd);
avgpower(P)
ans =
   3.0592e-10
```

This value is close to the theoretical value we expect from the additive noise model as computed above.* The PSD is plotted in Figure 7.8. Note that the PSD is approximately flat as it should be for white noise (the plot can be obtained by calling the `noisepsd` command with no outputs).

7.2.5 Evaluating the performance of the fixed-point filter

In order to evaluate how well the fixed-point filter compares to the reference floating-point filter, we can filter some test data and compare the

* The PSD here includes a factor of $1/2\pi$, therefore the intensity of the PSD is given by the variance divided by 2π. When we integrate in order to compute the average power, we should keep in mind that the variance has already been scaled by 2π.

results. In order to isolate the behavior of the filter, it is important that the same quantized input test data is used in all cases.

We would like to distinguish between three different cases:

- *What we ideally would want to compute,* $y_r[n]$. This is the result of using the non-quantized original filter coefficients and performing all multiplications/additions with infinite precision (or at least double-precision floating-point arithmetic).
- *The best we can hope to compute,* $y_d[n]$. This is the result of using the quantized coefficients, but without allowing any further round-off noise within the filter or at its output. All multiplications/additions are performed with full-precision.
- *What we actually compute,* $y[n]$. This is the result we get with all final fixed-point settings in our filter.

Clearly, what we actually compute can at best be the same as the best we can hope to compute. In order to make the best we can hope to compute closer to what we ideally would compute, we would have to use more bits to represent the filter coefficients (and throughout the filter).

Example 61 *Consider once again the following design:*

```
Hf = fdesign.lowpass(0.4,0.5,0.5,80);
Hd = design(Hf,'equiripple');
Hd.Arithmetic = 'fixed';
```

As we have said, the quantized filter uses 16-bits to represent the coefficients by default. A plot of the magnitude response of `Hd` *shows that 16-bits are not quite enough in this case to get the full 80 dB attenuation throughout the stopband. Nevertheless, if we choose to use 16 bits for the coefficients, the best we can hope to achieve can be computed for some test data as follows:*

```
rand('state',0);
x    = fi(2*(rand(1000,1)-.5),true,16,15);
Href = reffilter(Hd); % Compute reference filter
yr   = filter(Href,x);
yd16 = filter(Hd,x); % Uses full-precision
var(yr-double(yd16))
ans =
   4.8591e-10
```

The filter `Hd` *performs all multiplications/additions using full precision because the* `FilterInternals` *property is set by default to* `'FullPrecision'`.

If we decide to use say 18 bits for the coefficients instead of 16, we can get closer to the result obtained with the reference filter,

```
Hd18 = copy(Hd);
Hd18.CoeffWordLength = 18;
yd18 = filter(Hd18,x); % Uses full-precision
var(yr-double(yd18))
ans =
   2.4946e-11
```

Once we have settled the number of bits for the coefficients, we have established a baseline for the best we can hope to achieve if we can't use full-precision. We then measure how well our actual results compare to the best we can hope for.

Example 62 *Continuing with our previous example. Suppose we can only keep 16 bits at the output and use 16 bit coefficients. In this example, this means we need to remove 17 LSBs at the output as was shown in Figure 7.6. Once again, this can be done as follows:*

```
Hd.FilterInternals  = 'specifyPrecision';
Hd.OutputWordLength = 16; % 33-16 = 17
Hd.OutputFracLength = Hd.OutputFracLength-17;
```

If we now filter the same data and compute the energy in the error between the best we can hope for and what we actually compute:

```
y = filter(Hd,x);
var(double(y)-double(yd16))
ans =
   3.1004e-10
```

We of course obtain the same result we had previously computed for the variance of the noise, i.e., `2^(-14)^2/12`.

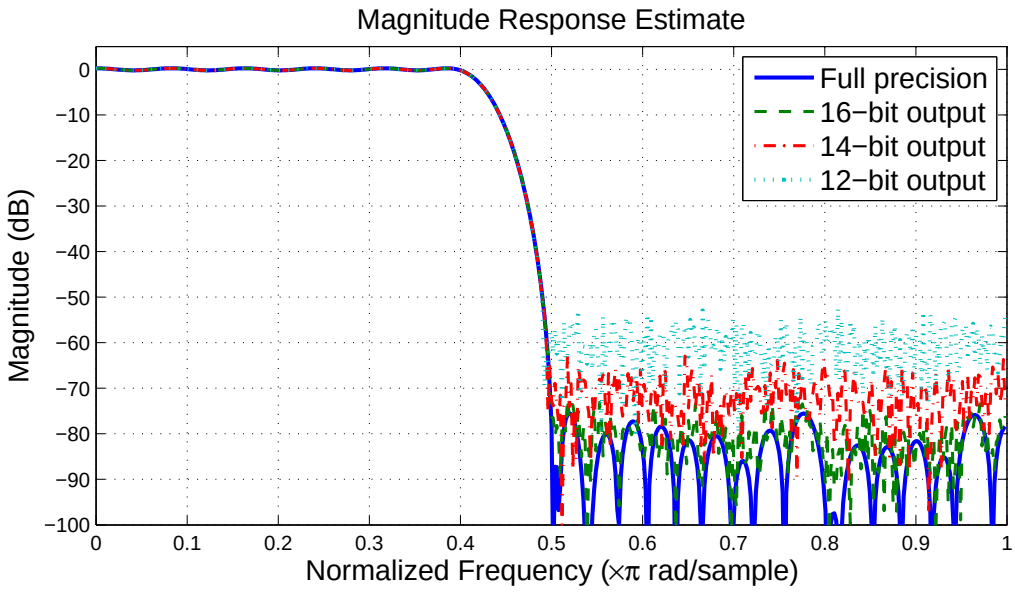


Figure 7.9: *Magnitude response estimates for various output wordlengths.*

Using the magnitude response estimate to evaluate performance

Most of the analysis commands we perform on fixed-point filters only take into account the coefficient quantization. This is true of the magnitude response, impulse response, group-delay, and pole/zero computation.

The magnitude response estimate however, computes an estimate of the magnitude response of the filter by actually filtering data through it. The fixed-point filtering means that roundoff error may be introduced because of discarding bits at the output or other roundoff if the additions/multiplications are not performed with full precision.

The closer the magnitude response estimate resembles the magnitude response computed solely based on the quantization of the coefficients, the more the actual filtering approaches the best we can hope to compute. In this way, we can use the magnitude response estimate to evaluate the performance of the fixed-point filter.

Example 63 *Let us continue with the same design we have been using the last several examples. If we start with full precision, the magnitude response estimate is almost indistinguishable from the magnitude response:*

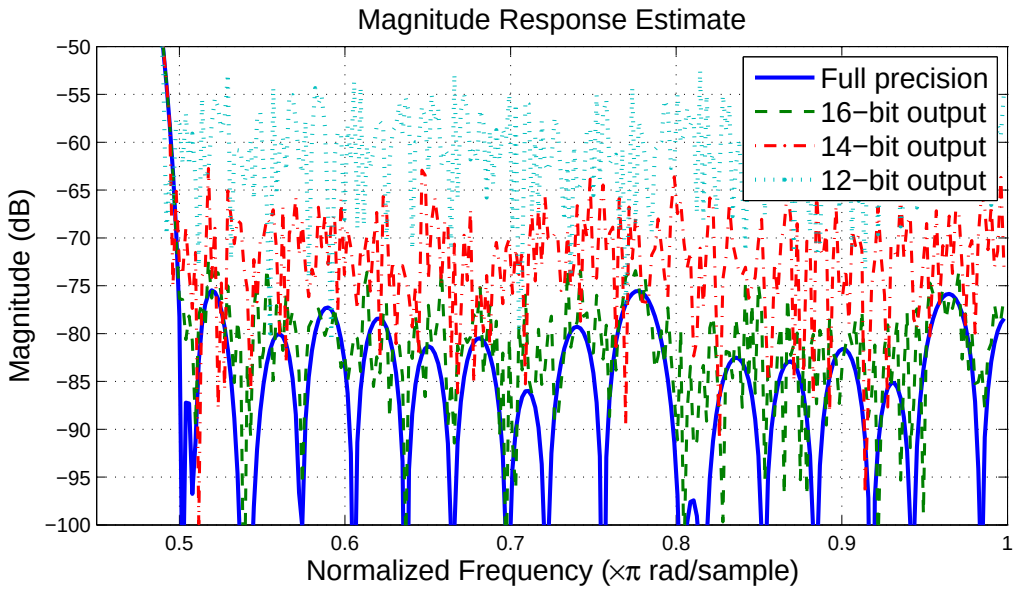


Figure 7.10: *Stopband details of magnitude response estimates for various output wordlengths.*

```
Hf = fdesign.lowpass(0.4,0.5,0.5,80);
Hd = design(Hf,'equiripple');
Hd.Arithmetic = 'fixed'; % 16-bit coefficients
H  = freqz(Hd);
He = freqrespest(Hd,1,'NFFT',512);
norm(H-He)
ans =
     0.0023
```

Now, we will compute the magnitude response estimate using always the same 16-bit coefficients, but in each case discarding a different number of bits at the output. Notice that we now use a relatively large number of trials, 50, in order to get a reliable average from the estimate (the computation takes a while). We'll start by discarding 17 bits,

```
Hd.FilterInternals  = 'specifyPrecision';
Hd.OutputWordLength = 16; % 33-16 = 17
Hd.OutputFracLength = Hd.OutputFracLength-17;
```

```
He16 = freqrespest(Hd,50,'NFFT',512);
```

We'll repeat this for 14-bit and 12-bit outputs,

```
Hd.OutputWordLength = 14;
Hd.OutputFracLength = 12;
He14 = freqrespest(Hd,50,'NFFT',512);
Hd.OutputWordLength = 12;
Hd.OutputFracLength = 10;
He12 = freqrespest(Hd,50,'NFFT',512);
```

The plot of the various magnitude response estimates is shown in Figure 7.9. Note the quantization at the output basically results in reduced stopband attenuation. The stopband is shown in greater detail in Figure 7.10. Note that we can once again observe about a 5 dB/bit behavior as we change the output wordlength.

Chapter 8

Implementing IIR Filters

8.1 Some basics of IIR implementation

8.1.1 The use of second-order sections

All of the IIR designs we have discussed in Chapter 2 are performed using second-order sections by default. The full polynomials of the transfer function are never formed. The main reason for this has to do with hardware implementations of IIR filters. However, even from the design perspective it is good practice to always keep the design in second-order section and avoid computing the transfer function explicitly.

The reason is that forming the transfer function involves polynomial multiplication, or equivalently convolution of the polynomial coefficients. Convolution can introduce significant round-off error even with double-precision floating-point arithmetic. This is mainly due to additions of large numbers with small numbers (that become negligible when using finite precision arithmetic).

Of course the effect gets worse as the filter order increases since we are performing more and more convolutions. For low-order designs, it is not absolutely necessary to use second-order sections, but there still may be good reasons to do so from an implementation perspective.

Example 64 *As a rather extreme example consider the following design:*

```
Hf    = fdesign.lowpass('Fp,Fst,Ap,Ast',.47,.48,.05,120);
Hc    = design(Hf,'cheby1');
[b,a] = tf(Hc);
```

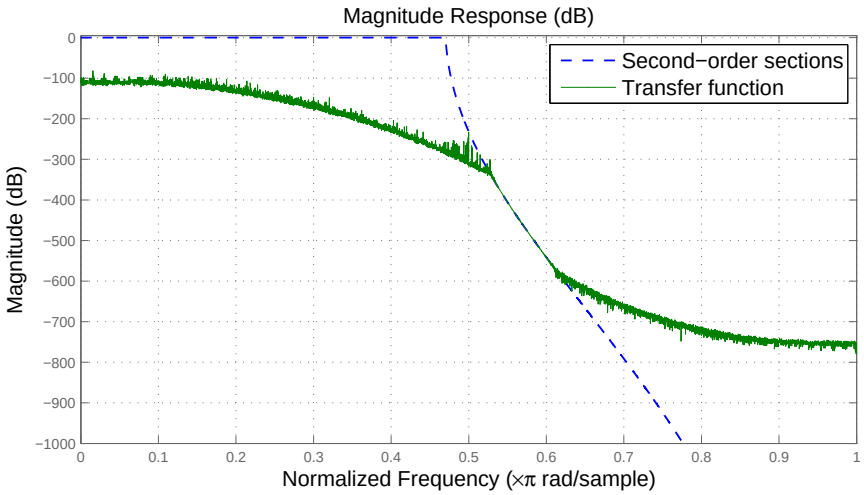


Figure 8.1: *A Chebyshev type I filter implemented in second-order sections and with the transfer function.*

The resulting filter order is very high (67). By forming the transfer function using the `tf` *command, we have completely distorted the response of the filter (see Figure 8.1). Moreover, the filter has become unstable! The transfer function of the polynomials is completely wrong as evidenced by the pole/zero plot (Figure 8.2). Compare to the pole/zero plot of the second-order sections to see what it should look like (Figure 8.3).*

The most common second-order section structures are the direct-form I and direct-form II. A two-section direct-form I filter is shown in Figure 8.4. The structure uses more delays than the direct-form II structure because the latter shares the delays between the numerator and denominator of each section while the former does not. However, if we do not use the scale value between sections, we can share the delays from one section to the next, so that the direct-form I structure would only require two more delays than the direct-form II rather than twice as many.

8.1.2 Allpass-based implementations

Second-order sections have been used for many years to implement IIR filters. However, another possibility is to use allpass-based implementa-

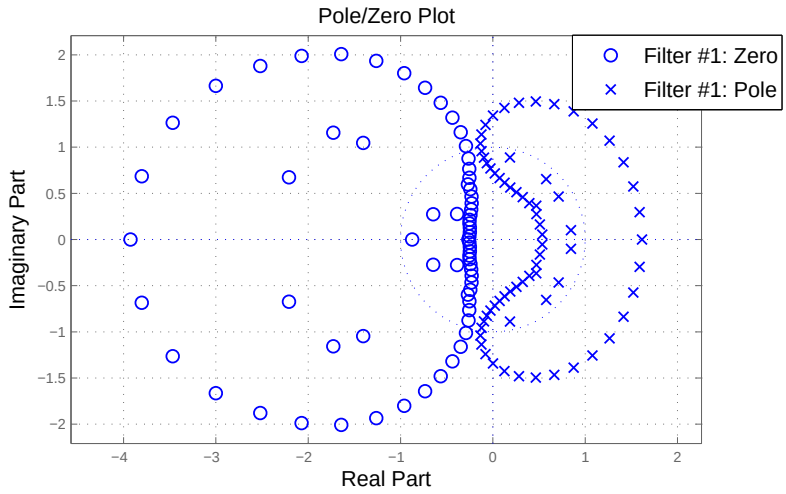


Figure 8.2: *Pole/zero plot of a Chebyshev type I filter implemented with the transfer function.*

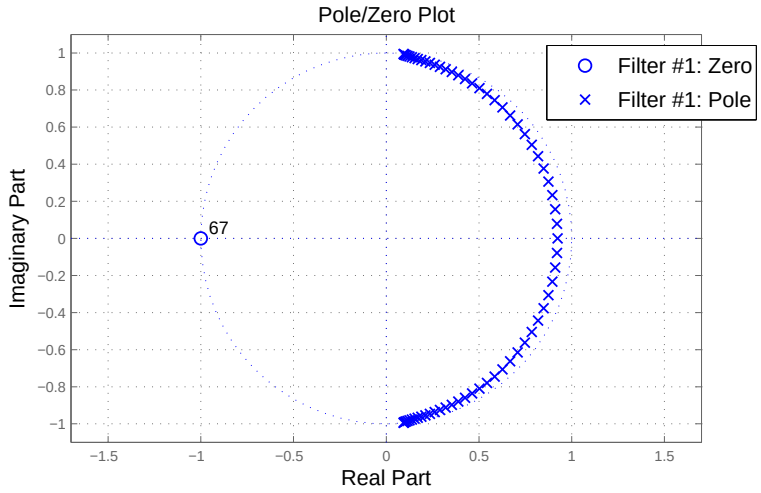


Figure 8.3: *Pole/zero plot of a Chebyshev type I filter implemented with second-order sections.*

tions. These have the advantage that they require less multipliers than corresponding second-order sections implementations.

The idea is to implement the filter $H(z)$ as the sum of two allpass filters

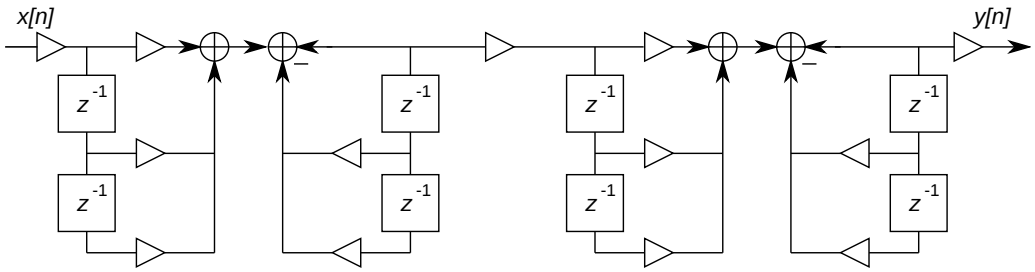


Figure 8.4: *Direct-form I second-order sections structure.*

$A_0(z)$ and $A_1(z)$,

$$H(z) = \frac{1}{2}(A_0(z) + A_1(z)) \tag{8.1}$$

This decomposition, known as parallel allpass or coupled-allpass, can be performed under certain conditions on $H(z)$ [15], [16], [17]. It is know that Butterworth, Chebyshev, and elliptic filters satisfy such conditions and therefore can be implemented in such manner.

The allpass filters themselves can be implemented in various ways. Lattice filters are one possibility. In the Filter Design Toolbox, the allpass filters are implemented using a minimal number of multipliers and adders. In keeping with the good practices of not forming the full transfer function, the allpass filters are implemented as cascades of low-order allpass sections. The delays are shared between sections whenever possible in order to minimize the number of them required.

Example 65 *To illustrate, consider the elliptic design of Example 25. When using the direct-form II second-order sections implementation we found that the number of multipliers required was 20, the number of adders was also 20 and the number of delays was 10. If we use cascaded low-order allpass sections,*

```
He2 = design(Hf,'ellip','FilterStructure','cascadeallpass');
```

we can find through the `cost` *function that the number of multipliers reduces to 12. The number of adders increases to 23, and the number of delays is 15.*

To visualize the structure, we use the `realizemdl` command to generate a Simulink model of the filter. Even though the number of adders is higher than with second-order sections, the saving in multipliers usually more than compensates. Ultimately, this depends on the hardware used

to implement the filter. On a programmable DSP processor for instance, an adder and a multiplier have the same cost. This is not the case with an FPGA or an ASIC.

Example 66 *For illustration purposes, we can easily get to the transfer function* of the allpass filters as follows:*

```
Hf = fdesign.lowpass('N,F3dB,Ap,Ast',5,.3,1,60);
He = design(Hf,'ellip','FilterStructure','cascadeallpass');
[b1,a1] = tf(He.stage(1).stage(1)); % A0(z)
[b2,a2] = tf(He.stage(1).stage(2)); % A1(z)
```

Note that `He.stage(1)` *contains the term* $A_0(z) + A_1(z)$ *while* `He.stage(2)` *is simply the 1/2 factor. If we look at the numerators* `b1` *and* `b2`*, we can see that they are reversed versions of* `a1` *and* `a2` *respectively. This is an indication of their allpass characteristic.*

Lattice wave digital filters

A variation of the cascaded allpass implementation is to use lattice wave digital filters. The term lattice in this case refers to the fact that there are two parallel allpass branches $A_0(z)$ and $A_1(z)$. It does not refer to the lattice structure proposed by Gray and Markel [18].

To use these structures for the elliptic design of Example 25, it is simply a matter of specifying the appropriate filter structure,

```
He3 = design(Hf,'ellip','FilterStructure','cascadewdfallpass');
```

This implementation requires the same 12 multipliers as the cascade allpass does. However, the number of adders increases to 34. The number of states is only 11, but the savings in number of states may not make up for the additional adders required.

In terms of computational cost, the cascade allpass implementation seems to be superior. However, the lattice wave digital filters are well-scaled for fixed-point implementations, a fact that may make their use compelling.

More about lattice wave digital filters can be found in [18] and [19] for example.

* This is not needed for implementation purposes.

8.2 Fixed-point implementation

Implementing IIR filters with fixed-point arithmetic is more challenging than the FIR case. We have already seen reasons not to use the transfer function of IIR filters even before thinking of fixed-point implementation. The case to avoid the transfer function becomes much stronger once fixed-point is taken into account. The following example showcases this.

Example 67 *Consider the following design:*

```
Hf = fdesign.lowpass(0.45,0.5,0.5,80);
He = design(Hf,'ellip'); % Already in SOS form
```

Let's construct a direct-form II filter using the transfer function equivalent to the SOS we have:

```
[b,a] = tf(He);
Htf = dfilt.df2(b,a);
```

Now let's quantize the coefficients using 16 bits and compare the resulting magnitude responses.

```
He.Arithmetic  = 'fixed';
Htf.Arithmetic = 'fixed';
```

The magnitude responses are shown in Figure 8.5. Note that in this case, forming the double-precision floating-point transfer function does not significantly alter the magnitude response (try `fvtool(b,a)`*). The deviation that we observe for the transfer function in Figure 8.5 has to do with the quantization.*

It is worth noting that the magnitude responses shown take into account only the quantization of coefficients. Any other round-off error introduced by the fixed-point settings is not reflected in the magnitude response analysis shown in `fvtool`.

8.2.1 Fixed-point filtering

In general, for fixed-point implementation, we want to stay away from the transfer function because quantizing large polynomials introduces significant errors when compared to quantizing second-order polynomials.

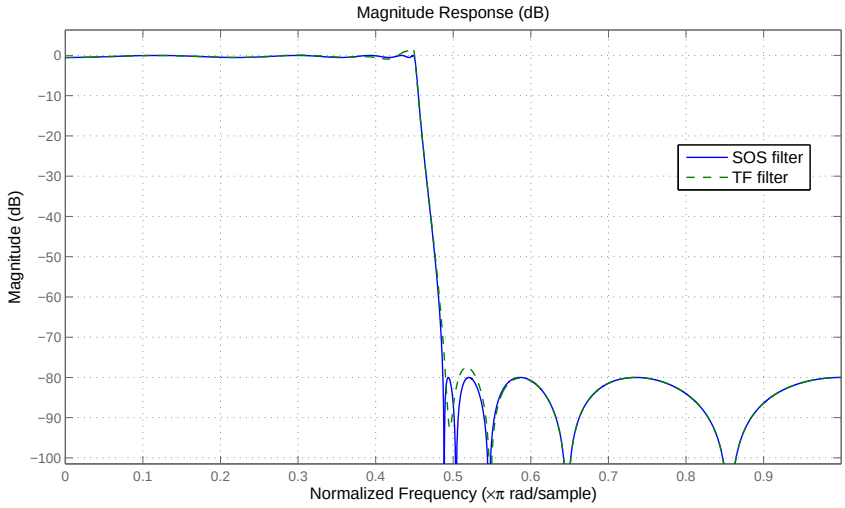


Figure 8.5: *Comparison of magnitude response of the same filter quantized to 16-bit coefficients. In one case, we use second-order sections (SOS) while in the other case we use the transfer function (TF).*

Nevertheless, with IIR filters, it is not enough to obtain a nice magnitude response with the quantized coefficients.

Unlike FIR filters, it is not possible to implement IIR filters using full-precision arithmetic since this would result in infinite bit growth due to the feedback. Therefore, it is necessary to introduce quantization at the feedback points within a filter structure. This quantization will of course introduce roundoff error. In the worst case, the quantization produces an overflow condition.

Nevertheless, as with FIR filters, we can consider the following computations for the output:

- $y_r[n]$, *what we ideally would want to compute.* This is the result of using the non-quantized original filter coefficients and performing all multiplications/additions with infinite precision (or at least double-precision floating-point arithmetic).

- $y_d[n]$, *the best we can hope to compute.* Since full-precision is not an option with IIR filters, in order to obtain a baseline, we can use the `double()` command on the filter to obtain a filter that has quan-

tized coefficients, but performs all multiplications/additions using double-precision floating-point arithmetic. *

- $y[n]$, *what we actually compute,.* This is the result we get with all final fixed-point settings in our filter.

Let us consider the direct-form II second-order sections structure. For each section, the feedback part comes before the shared states. Since we need to quantize the result from the feedback loop prior to the values entering the states, we want to ensure that round-off error at this point is minimized.

Example 68 *Let us repeat the previous design with a slight change. By default, we scale and re-order the second-order sections in order to help obtaining good fixed-point filtering results from the start. However, for now, let's remove the scaling/re-ordering to show how a good magnitude response is necessary but not sufficient in order to obtain satisfactory results.*

```
Hf = fdesign.lowpass(0.45,0.5,0.5,80);
He = design(Hf,'ellip','SOSScaleNorm',''); % Don't scale
He.Arithmetic = 'fixed';
```

We'll compute the ideal output, $y_r[n]$, and compare to the best we can hope to compute, $y_d[n]$. The difference between the two is due solely to coefficient quantization.

```
rand('state',0);
x  = fi(2*(rand(1000,1)-.5),true,16,15);
Hr = reffilter(He);
yr = filter(Hr,x); % Ideal output
Hd = double(He);
yd = filter(Hd,x); % This is the best we can hope for
var(yr-yd)
ans =
   1.5734e-08
```

Now, let's enable min/max/overflow logging and filter the same data with the fixed-point filter:

* Note that the commands reffilter() and double() differ in that the former uses the original floating-point coefficients, while the later uses the quantized coefficients.

```
fipref('LoggingMode','on')
ye = filter(He,x);
qreport(He)
```

If we look at the results from `qreport`*, we can see that the states have overflown 37 out of 10000 times. Because of the overflow, it is not worth comparing* `ye` *to* `yd`*.*

In order to avoid overflow, we can use scaling to control the signal growth at critical points inside the filter. The most stringent scaling, ℓ_1-norm scaling, will ensure that no overflow can occur.

Example 69 *Let us design the elliptic filter again, but using ℓ_1-norm scaling this time. For now, we will disable re-ordering of the second-order sections:*

```
s = fdopts.sosscaling;
s.sosReorder = 'none';
Hl1 = design(Hf,'ellip','SOSScaleNorm','l1',...
    'SOSScaleOpts',s);
Hl1.Arithmetic = 'fixed';
```

Now let's filter with this scaled structure,

```
yl1 = filter(Hl1,x);
qreport(Hl1)
```

Notice from the report that we have no overflows. Therefore we can compare the output to our baseline in order to evaluate performance,

```
var(yd-double(yl1))
ans =
   1.8339e-04
```

When doing fixed-point filtering, one needs to compromise between maximizing the SNR and minimizing the possibility of overflows. The ℓ_1-norm scaling we have used is good for avoiding overflows, but reduces the signal levels too much, adversely affecting the SNR. A more commonly used scaling (the default) is to use $\mathcal{L}_\infty$-norm scaling. However, this scaling may result in overflow as the next example shows.

Example 70 *Let's design the filter and scale using $\mathcal{L}_\infty$-norm scaling. For now, we still do not re-order the sections.*

```
HLinf = design(Hf,'ellip','SOSScaleNorm','Linf',...
     'SOSScaleOpts',s);
HLinf.Arithmetic = 'fixed';
```

Now let's filter and look at the report.

```
yLinf = filter(HLinf,x);
qreport(HLinf)
```

The report shows that 8 overflows in the states have occurred.

So far, we have not re-ordered the second-order sections in the examples we have explored. Re-ordering can help improve the SNR, and, as the next example shows, sometimes even help avoid overflow.

Example 71 *Let's once again use $\mathcal{L}_\infty$-norm scaling, but this time use automatic re-ordering of the sections.*

```
s = fdopts.sosscaling;
s.sosReorder = 'auto';
HLinf = design(Hf,'ellip','SOSScaleNorm','Linf',...
     'SOSScaleOpts',s);
HLinf.Arithmetic = 'fixed';
```

Once again, let's filter the same data, and look at the report.

```
yLinf = filter(HLinf,x);
qreport(HLinf)
```

Since we did not overflow, it is meaningful to compare the output to our baseline to evaluate performance.

```
var(yd-double(yLinf))
ans =
   1.4809e-05
```

As we can see, the result is more than an order of magnitude better than what we got with ℓ_1-norm scaling.

Yet a less stringent scaling is to use $\mathcal{L}_2$-norm scaling.* Yet the chances of overflowing when using that type of scaling are even larger.

It is worth noting that in some applications, it is preferable to allow for the occasional overflow in order to increase the SNR overall. The assumption is that an overflow once in a while is not critical for the overall performance of the system.

8.2.2 Autoscaling

The scaling we have seen so far is data agnostic. As a result sometimes overflow occurs.

We can use the `autoscale` command in order to optimize the fixed-point fractional length settings in a filter based on specific input data that is given. Doing so, we can get even better results than if we use data-agnostic scaling.

Example 72 *Let's try once again to design the filter without data-agnostic scaling.*

```
Hf = fdesign.lowpass(0.45,0.5,0.5,80);
He = design(Hf,'ellip','SOSScaleNorm',''); % Don't scale
He.Arithmetic = 'fixed';
```

We know that this filter will overflow with the input data we have. However, if we use this input data to autoscale,

```
Hs = autoscale(He,x);
```

and we filter the same data,

```
ys = filter(Hs,x);
qreport(Hs)
```

we can see from the report that now overflow has been avoided. We now evaluate the output relative to our baseline,

```
var(yd-double(ys))
ans =
   1.4144e-07
```

* This norm can be shown to be the same as ℓ_2-norm. This is Parseval's theorem.

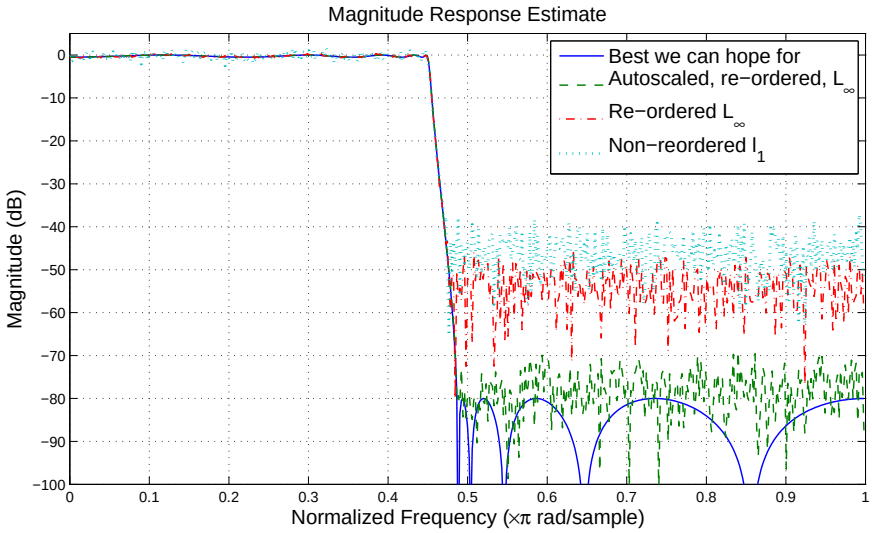


Figure 8.6: *Magnitude response estimates for various scaling cases.*

Which is of course the best result we have obtained so far.

Finally, if we combine $\mathcal{L}_\infty$-norm scaling, automatic re-ordering, and autoscaling, we can easily verify that we can get even better results:

```
var(yd-double(ys2))
ans =
   2.3167e-08
```

8.2.3 Evaluating filter performance using the magnitude response estimate

As we saw for FIR filters, one way to evaluate the performance of the fixed-point IIR filter is to compute an estimate of the magnitude response by filtering data.

Because the filtering is performed with fixed-point arithmetic, the magnitude response estimate takes into account not only the quantization of the coefficients, but also any roundoff error introduced within the filter.

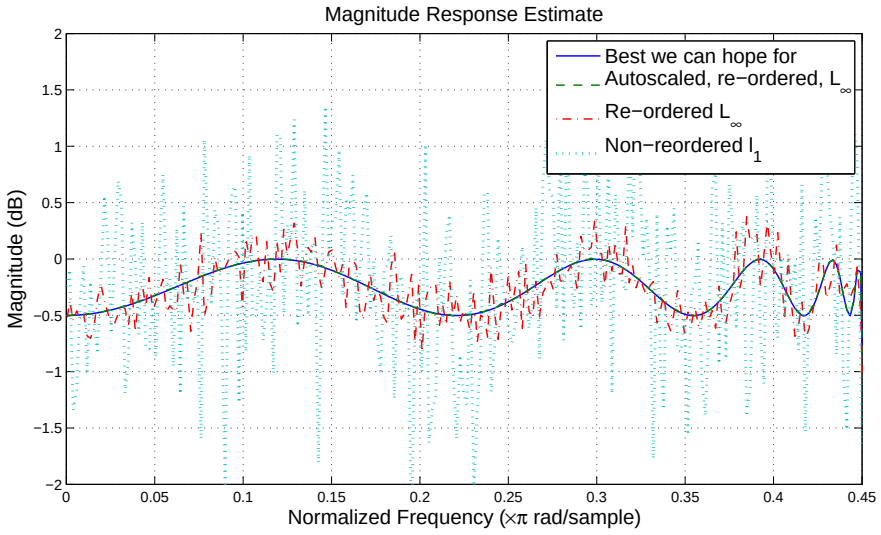


Figure 8.7: *Passband details of magnitude response estimates for various scaling cases.*

The magnitude response estimate is particularly useful when the fixed-point filter behaves in a weekly nonlinear manner.* This means that it is most useful when no overflow occurs given that overflows introduce significant nonlinear behavior.

As an example, Figure 8.6 shows the magnitude response estimate for four cases corresponding to examples above. In all four cases shown, overflow is avoided. Once again, the roundoff error in the filtering has a net effect of reducing the stopband attenuation. Moreover, unlike the FIR case, the roundoff can also affect the passband response of the filter. The passband details are shown in Figure 8.7.

* Quantization is a nonlinear operation. But a linear model that approximates the behavior as additive white noise is adequate in many cases as we have seen.

Part III

Appendices

Appendix A

Summary of relevant filter design commands

A.1 Filter Design (fdesign)

A.1.1 Setup design specifications

Use `fdesign` objects to store design specifications. This will not design the filter, only set the specs. In order to design you need to invoke the `design` command (see below).

```
f = fdesign.lowpass('Fp,Fst,Ap,Ast',0.3,0.4,0.5,75);
```

To view all available frequency response designs (lowpass, highpass, etc) type:

```
help fdesign/responses
```

To change specification set, it can be done a step at a time,

```
f.Specification = 'N,Fc,Ap,Ast';
f.FilterOrder   = 60;
...
```

but it can also be done in one shot,

```
setspecs(f,'N,Fc,Ap,Ast',60,0.3,0.5,75);
```

By default, frequency specifications are assumed to be normalized with an implied scaling by π, so for example a value of 0.3 for `Fc` means that the cutoff frequency is being set to 0.3π radians/sample. In order to specify frequencies in Hertz, the sampling frequency must be appended,

```
setspecs(f,'N,Fc,Ap,Ast',60,30,0.5,75,200); % Fc = 30 Hz, Fs = 200 Hz
```

For a list of all available specification sets for a particular response,

```
set(f,'Specification')
```

A.1.2 Design options

To view available design methods for a particular response/specification combination:

```
designmethods(f)       % List all design methods
designmethods(f,'iir') % List only IIR designs
designmethods(f,'fir') % List only FIR designs
```

To design with default design options

```
h = design(f) % Use default design method
% For the following commands, h will be an array
h = design(f,'all') % Design for all design methods
h = design(f,'fir') % Design for all FIR methods
h = design(f,'iir') % Design for all IIR methods
```

To view design options and defaults for a particular design method:

```
designoptions(f,'equiripple')
```

For context-sensitive (response sensitive) help type:

```
help(f,'equiripple') % Specific help for lowpass/equiripple combo
```

A.1.3 Design analysis/validation

To view various frequency-domain, time-domain, or pole-zero responses:

```
f = fdesign.lowpass('Fp,Fst,Ap,Ast',1.3e3,1.4e3,0.5,75);
h = design(f,'equiripple','StopbandShape','1/f','StopbandDecay',3);
fvtool(h)
```

In order to measure critical frequencies and/or ripple/attenuations:

```
measure(h)
```

To view a high-level implementation cost:

```
f = fdesign.halfband('TW,Ast',0.01,80);
h = design(f,'equiripple');
cost(h)
ans =
Number of Multipliers : 463   % Number of non-zero and non-one coefficients
Number of Adders      : 462   % Total number of additions
Number of States      : 922   % Total number of states
MultPerInputSample    : 463   % Non-zero/non-one multiplications per input samp
AddPerInputSample     : 462   % Additions per input sample
```

For some general information about the design:

```
info(h)
```

A.2 Selecting filter structure

At design time, different filter structures can be specified. For FIR filters, the default structure is direct-form, `dffir`. For symmetric filters (linear phase), half the multipliers can be saved by using a symmetric FIR structure:

```
f = fdesign.lowpass('N,Fp,Fst',40,0.4,0.5);
h1 = design(f,'Wpass',1,'Wstop',10,'FilterStructure','dffir');   % default
h2 = design(f,'Wpass',1,'Wstop',10,'FilterStructure','dfsymfir');
cost(h1)
cost(h2)
```

For IIR filters, the default structure is direct-form II. IIR filters are designed by default as cascaded second-order sections (SOS). Direct-form I SOS structures use more states, but may be advantageous for fixed-point implementations,

```
f = fdesign.lowpass('N,F3dB,Ap,Ast',5,0.45,1,60);
h1 = design(f,'FilterStructure','df1sos');
h2 = design(f,'FilterStructure','df2sos');
cost(h1)
cost(h2)
```

A.3 Scaling IIR SOS structures

By default, sos IIR designs using structures such as `df1sos` and `df2sos` are scaled. The scaling uses L_∞-norm scaling, so that the cumulative magnitude response of the second-order sections (first section alone, first and second sections, first-second-and-third, etc.) does not exceed one (0 dB):

```
f = fdesign.lowpass;
h1 = design(f,'ellip','FilterStructure','df1sos');
h2 = design(f,'ellip','FilterStructure','df2sos');
s = dspopts.sosview;
s.view='Cumulative'; % Set view to cumulative sections
fvtool(h1,'SOSViewSettings',s)
s.secondaryScaling = true; % Use secondary scaling point for df2sos
fvtool(h2,'SOSViewSettings',s)
```

This scaling helps avoid overflows when the filter is implemented with fixed-point arithmetic, but doesn't totally eliminate the possibility of them. To eliminate overflows, the more restrictive l_1-norm scaling can be used (at the expense of reduced SNR due to sub-utilization of the dynamic range):

```
f = fdesign.lowpass;
h1 = design(f,'ellip','FilterStructure','df1sos','SOSScaleNorm','l1');
s = dspopts.sosview;
s.view = 'Cumulative';
fvtool(h1,'SOSViewSettings',s)
```

A.4 Designing multirate filters

To design multirate filters, select the multirate type, then the interpolation/decimation factor(s), then any `fdesign` single-rate response,

```
f = fdesign.interpolator(4,'lowpass','N,Fc,Ap,Ast',60,1/4,0.5,75); % Interp by
f = fdesign.decimator(2,'halfband','TW,Ast',0.1,95);               % Decim  by
f = fdesign.rsrc(3,5,'nyquist',5,'TW,Ast',0.05,80);                % Up=3 Down
```

After the specifications are set, the usual commands are available: `design`, `designmethods`, `designoptions`, etc.

The `cost` command takes into account the polyphase savings of multirate filters:

```
f1 = fdesign.interpolator(15,'lowpass','Fp,Fst,Ap,Ast',...
     1/15-0.01,1/15+0.01,1,80);
h1 = design(f1,'multistage'); % Results in 2 stages
cost(h1) % Compare to a single-stage design: design(f1,'equiripple');
f2 = fdesign.decimator(8,'nyquist',8,'TW,Ast',0.03,80);
h2 = design(f2,'multistage'); % Result in 3 stages
cost(h2) % Compare to a single-stage design: design(f1,'kaiserwin');
% When applicable, the use of IIR halfbands results in huge implementation savi
% The implementation uses allpass-based polyphase branches
h3 = design(f2,'multistage','HalfbandDesignMethod','iirlinphase');
cost(h3) % If linear phase is not an issue, ellip halfbands would be even bette
```

A.5 Converting to fixed point

In order to convert a design to fixed point, set the `Arithmetic` property,

```
h.Arithmetic = 'fixed';
```

or

```
set(h,'Arithmetic','fixed');
```

By default, the coefficients are quantized to 16 bits with automatic scaling, and, for FIR filters, the filter internals (additions and multiplications) are set in a such way that no loss of precision occurs (so-called "full precision"). The input signal to the filter is assumed to be a signal quantized

with 16 bits and covering the range $[-1,1)$. The LSB is scaled by 2^{-15}. (Two's-complement is assumed for all fixed-point quantities.)

Coefficient quantization affects the frequency (and time) response of the filter. This is reflected in `fvtool` and `measure`,

```
f = fdesign.decimator(2,'Halfband','TW,Ast',0.01,80);
h = design(f,'equiripple');
fvtool(h)
measure(h)
% Compare to
h.Arithmetic = 'fixed';
fvtool(h)
measure(h)
```

For FIR filters, in order to change fixed-point settings for the output and/or additions/multiplications within the filter, change the filter internals first*,

```
h.FilterInternals = 'SpecifyPrecision'
```

For example to view the effect solely due to the use of a smaller number of bits for the output of a filter, one could write something like,

```
x = randn(1000,1);      % Create a random input
xn = x/norm(x,inf);     % Normalize to the range [-1,1]
xq = fi(xn,true,16,15); % Quantize to 16 bits. Cover the range [-1,1)
f = fdesign.decimator(2,'Halfband','TW,Ast',0.01,80);
h = design(f,'equiripple','StopbandShape','linear','StopbandDecay',40);
h.Arithmetic = 'fixed';
yfull = filter(h,xq);   % Full precision output (33 bits)
h.FilterInternals  = 'SpecifyPrecision';
h.OutputWordLength = 16;
h.OutputFracLength = 13;
y16 = filter(h,xq);     % 16-bit output
plot(double(yfull-y16)) % Plot the difference due to output quantization
```

* The LSB scaling for any fixed-point quantity can be changed by setting the appropriate "FracLength" parameter. Note that this quantity is not really a "length" since it can be greater than the word length or even negative. For example a word length of 4 bits and a "FracLength" of 8 means the bits (from MSB to LSB) are scaled by -2^{-5},2^{-6},2^{-7}, and 2^{-8} respectively. (The negative scaling of the MSB is due to the two's-complement representation.)

There are a couple of commands that can come in handy for analysis when converting filters from floating point to fixed point: `double`, and `reffilter`,

```
f = fdesign.decimator(2,'Halfband','TW,Ast',0.01,80);
h = design(f,'equiripple');
h.Arithmetic = 'fixed';
hd = double(h);    % Cast filter arithmetic to double. Use quantized coeffs
hr = reffilter(h); % Get reference double-precision floating-point filter
```

The difference between `hd` and `hr` lies in the filter coefficients. The reference filter, `hr`, has the original "ideal" double-precision coefficients, while the casted-to-double filter, `hd`, has a casted version of the quantized coefficients.

These functions can be used to compare what we ideally would like to compute, with the best we can hope to compute*, and with what we can actually compute. Continuing with the previous two examples:

```
yr = filter(hr,xq); % What we would ideally like to compute
yd = filter(hd,xq); % Best we can hope to compute. Same as yfull
plot(double([yr-yd,yr-y16]))
```

A.6 Generating Simulink blocks

There are two commands that can be used to generate blocks from floating-point or fixed-point filters: `block()` and `realizemdl()`:

```
f = fdesign.hilbert('N,TW',122,0.1);
h = design(f);
block(h)
f = fdesign.decimator(2,'halfband','TW,Ast',0.05,75);
h = design(f,'iirlinphase','FilterStructure','cascadeallpass');
realizemdl(h)
```

The difference between them is that `block` will use the Digital Filter block (or one of the multirate blocks) from the Signal Processing Blockset while

* For FIR filters with multiplications/accumulations computed with more than 53 bits, the best one can hope to compute should be determined with full-precision fixed-point filtering rather than casting to double-precision floating point.

`realizemdl` will build the filter from scratch using basic elements like adders/multipliers.

The use of the `block` command will generally result in faster simulation speeds and better code generation. However, `realizemdl` provides support for more filter structures and will permit customization of the resulting block diagram (say to reduce the number of bits of some of the multipliers of a filter or to rewire the filter in some way).

A.7 Graphical User Interface

Most of the commands discussed here are available through a GUI:

```
filterbuilder
```

Note that to design multirate filters, you should first select the response (`lowpass`, `halfband`, etc) and then set the Filter Type to decimator, etc (if available).

Once designed (and possibly quantized), the filter can be saved to the workspace as an object named with the variable name specified at the top.

Appendix B

Analog/Digital Sampling and Reconstruction

This appendix provides a short review of both theoretical and practical issues related to sampling of a signal and reconstruction of an analog signal from a sampled version of it. In general, understanding sampling and analog reconstruction is essential in order to understand multirate systems. In fact, as we will see, sampling an analog signal is just the extreme case of downsampling a digital signal thereby reducing its sampling rate. Similarly, analog reconstruction is just the edge case of increasing the sampling rate of a signal.

The material presented here is standard in many textbooks*, but there are a couple of practical issues that are often overlooked and that we wish to highlight.

First, it is well-known that in order to avoid aliasing when sampling it is necessary to sample at least twice as often as the highest frequency component present. What is somewhat less well-known is that in lieu of ideal brickwall anti-aliasing and reconstructing lowpass filters, we should always sample a bit higher than twice the maximum frequency component. As a rule of thumb, anywhere between 10 and 20 percent higher. This is done to accommodate for the transition bands in the various filters involved. Moreover, if we oversample by a factor of say two or four to ease the burden of the analog anti-aliasing filter, we still leave an extra 10-20 percent of room even though at this point we would be sampling

* In particular the textbooks [3] and [11] are recommended in addition to the article [35].

anywhere between 4 and 8 times higher than the highest frequency component.

Moreover, since we know that the information in the excess frequency bands is of no interest for our application, we allow for aliasing to occur in this band. This enables us to sample at a rate that, while larger than twice the frequency of interest, does not need to be as large as twice the stopband-edge of the anti-aliasing filter. Equivalently, this allows for the transition width of the anti-aliasing filter to be twice as large (resulting a simpler, lower-order filter) than what it would be if did not permit aliasing in the excess frequency bands.

As an example consider an audio signal. The frequency range of interest is 0 to 20 kHz. Ideally we would bandlimit the analog signal exactly to 20 kHz prior to sampling with an anti-alias analog filter. We would then sample at 40 kHz. Subsequently, in order to reconstruct the analog signal from the samples, we would use a perfect lowpass analog filter with a cutoff frequency of 20 kHz that removes all spectral replicas introduced by sampling. In reality, we band-limit the signal with an analog anti-aliasing filter whose passband extends to 20 kHz but whose stopband goes beyond 20 kHz by about 20 to 40 percent. Typical sampling rates for audio are 44.1 kHz or 48 kHz, meaning that we have an excess bandwidth of about 10 to 20 percent. The stopband-edge of the anti-aliasing filter will be at about 24.1 kHz or 28 kHz respectively. These would also be the maximum frequency components of the audio signal that is bandlimited with such anti-alias filter. Since we don't quite sample at twice the maximum frequency components, aliasing occurs. However, aliasing will only occur in the excess frequency band, 20 kHz to 22.05 kHz or 20 kHz to 24 kHz depending on the sampling frequency.

Other sampling rates used for audio are 88.2 kHz, 96 kHz, 176.4 kHz, and 192 kHz. These sampling rates correspond to 2x or 4x oversampling, but the important thing is that they still include an extra 10 to 20 percent cushion in addition to the 2x or 4x oversampling.

The second issue we wish to highlight is that when choosing the stopband edge frequency for a decimation filter we can in many cases allow for transition-band overlap (i.e. aliasing in the transition band). This issue is related to the excess bandwidth resulting from the 10 to 20 percent oversampling that we have just mentioned and it is the reason the extra 10 to 20 percent is still present even if there is an oversampling of 2x or 4x.

Allowing for transition-band overlap (aliasing in the excess frequency

bands) when decimating means that the cutoff frequency (not the stopband-edge frequency) for the decimation filter can be selected as π/M where M is the decimation factor. This in turn means that efficient Nyquist filters can be used for decimation. In particular, very efficient halfband filters can be used when decimating by a factor of two.

We will elaborate on all this in §B.5.

B.1 Sampling an analog signal

We will provide only a brief overview of sampling. For a complete treatment of the subject see [35]. Sampling is also covered in any signal processing textbook (see e.g. [3]).

Consider an analog signal $x_a(t)$ with Fourier transform $X_a(f)$ also called its spectrum or frequency response,

$$X_a(f) = \int_{-\infty}^{\infty} x_a(t) e^{-2\pi j f} dt.$$

If we construct a sampled signal $x_s(t)$ from the original signal $x_a(t)$ in the following way:

$$x_s(t) = \sum_{n=-\infty}^{\infty} x_a(nT)\delta(t - nT)$$

The spectrum of the sampled signal, $X_s(f)$ is related to the spectrum of the original signal by the Poisson summation formula,

$$X_s(f) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X_a(f - kf_s) \tag{B.1}$$

where $f_s = 1/T$ is the sampling frequency.

Note that the sampled signal $x_s(t)$ is still an analog signal in the sense that it is a function of the continuous-time variable t. Therefore, its spectrum is still given by the continuous-time Fourier transform,

$$X_s(f) = \int_{-\infty}^{\infty} x_s(t) e^{-2\pi j f} dt.$$

However, if we form the discrete-time sequence

$$x[n] = \{x_a(nT)\} \quad \forall n,$$

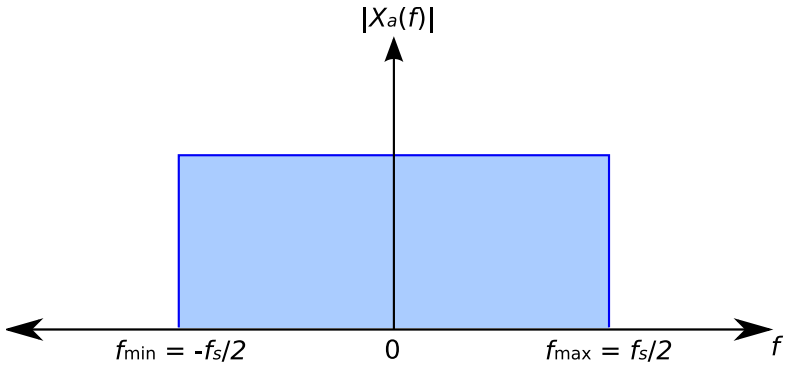


Figure B.1: *Spectrum of band-limited analog signal with maximum frequency given by* $f_s/2$.

that is, we form a sequence by recording the value of the analog signal $x_a(t)$ at each multiple of T, we can determine the continuous-time Fourier transform of $x_s(t)$ by computing the discrete-time Fourier transform of $x[n]$,

$$X_s(f) = X(f) = \sum_{n=-\infty}^{\infty} x[n]e^{-2\pi jfnT}.$$

So even though $x_s(t)$ and $x[n]$ are conceptually different, they have the same frequency response.

Eq. (B.1) is valid for any signal $x_a(t)$ whether band-limited or not [3]. The terms $X_a(f - kf_s)$ are shifted versions of the spectrum of $x_a(t)$, and are called spectral replicas. They are centered around multiples of f_s. This concept is key to understanding multirate signal processing so we'd like to emphasize it. Any sampled signal has spectral replicas centered around multiples of its sampling frequency. The higher the sampling frequency, the further apart the spectral replicas will lie.

In general, when we form the sum of all the spectral replicas as in Eq. (B.1), the replicas will interfere with each other due to frequency overlapping. This interference is called aliasing. However, if the signal is bandlimited in such a way that only one replica has a non-zero value for any given frequency, then the replicas will not overlap and they will not interfere with each other when we add them as in Eq. (B.1).

Obviously if $X_a(f) \neq 0 \, \forall f$, the replicas will invariably overlap. However, if $x_a(t)$ is bandlimited, i.e. its spectrum is equal to zero for all frequencies above a certain threshold and all frequencies below a certain

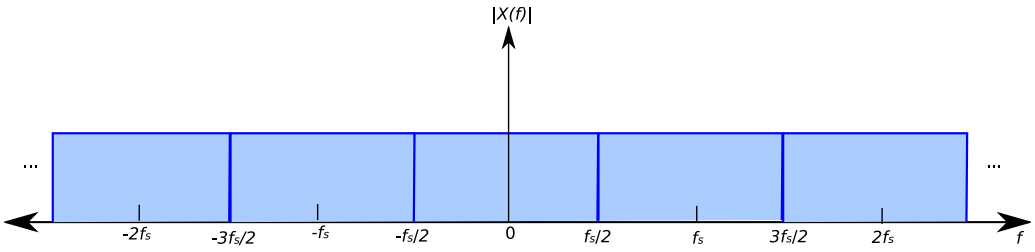


Figure B.2: *Spectrum of sampled signal with maximum frequency given by* $f_s/2$.

threshold, then by spreading the replicas enough apart from each other, i.e. choosing a large enough value for f_s, the replicas will not overlap thereby avoiding aliasing.

It is easy to see that if the analog signal's spectrum satisfies

$$X_a(f) = 0, \quad |f| > \frac{f_s}{2} \tag{B.2}$$

then the replicas will not overlap. Aliasing will thus be avoided if the sampling frequency is at least equal to the two-sided bandwidth.

Figure B.1 shows the magnitude spectrum of an example of an analog signal that satisfies Eq. (B.2). This case corresponds to the best known case, i.e. $f_{\max} = f_s/2$ and $f_{\min} = -f_s/2$. The two-sided bandwidth, B_x, is simply $B_x = f_{\max} - f_{\min} = 2f_{\max} = f_s$.

If the analog signal is not bandlimited, we must make it so by lowpass filtering the signal with an analog filter prior to sampling. Such filter is called an anti-alias filter for obvious reasons and its cutoff frequency must be chosen such that it allows all frequencies we are interested in for a given application to pass.

A critically sampled spectrum, i.e. one in which the sampling frequency is exactly equal to the two-sided bandwidth is shown in Figure B.2. The spectral replicas are adjacent to each other but do not overlap. If we were to oversample the signal, that is if we choose $f_s' > B_x$, the sampled spectrum would be something like what is depicted in Figure B.3. Notice the "white-space" between replicas due to oversampling.

The interval $[-f_s/2, f_s/2]$ is called the Nyquist interval. Since f_s determines the number of samples per second, when comparing two Nyquist intervals of different size, it is useful not only to realize that one encompasses a larger range of frequencies than the other, but also that it involves

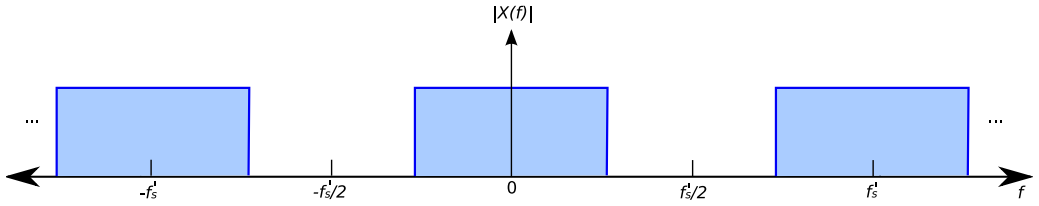


Figure B.3: *Spectrum of an oversampled signal.*

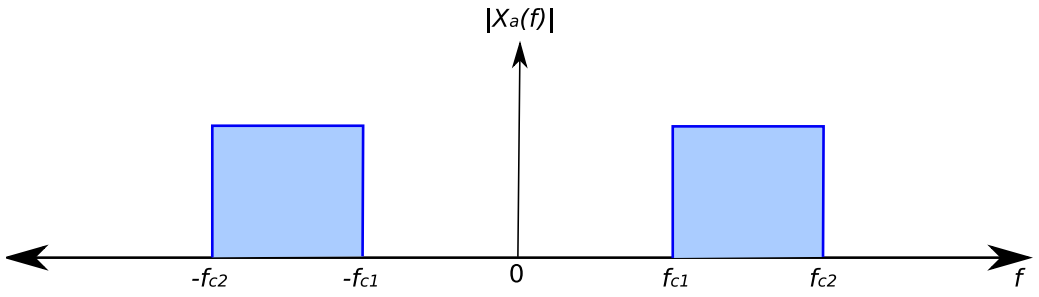


Figure B.4: *Continuous-time signal with bandpass spectrum.*

a larger amount of samples. Therefore, in general, any digital processing performed on a signal on a larger Nyquist interval requires more computation than comparable signal processing performed on a signal that occupies a smaller Nyquist interval. This means that we generally want to avoid oversampling (i.e. avoid white-space in the spectrum). However, oversampling can be useful in some particular cases.

The spectral replicas that appear in a sampled signal make the signal periodic in the frequency domain. This periodicity means that any operation performed on a signal within the Nyquist interval will automatically happen in all spectral replicas as well. We can take advantage of this fact to efficiently process analog bandpass signal as is explained next.

B.1.1 Bandpass sampling

Now consider the sampling of the bandpass signal depicted in Figure B.4. If we were to sample at $f_s = f_{\max} - f_{\min} = 2f_{c2}$ the resulting signal would have the spectrum shown in Figure B.5. The white-space between replicas tells us that the signal has been oversampled. In fact, for this signal we can sample at a much lower rate as depicted in Figure B.6 and still avoid

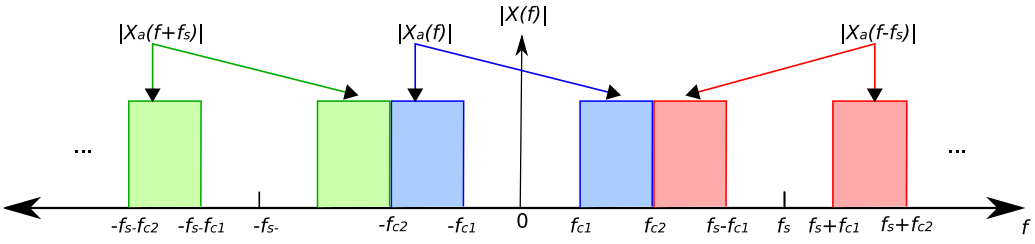


Figure B.5: *Spectrum of sampled bandpass signal with* $f_s = 2f_{c2}$.

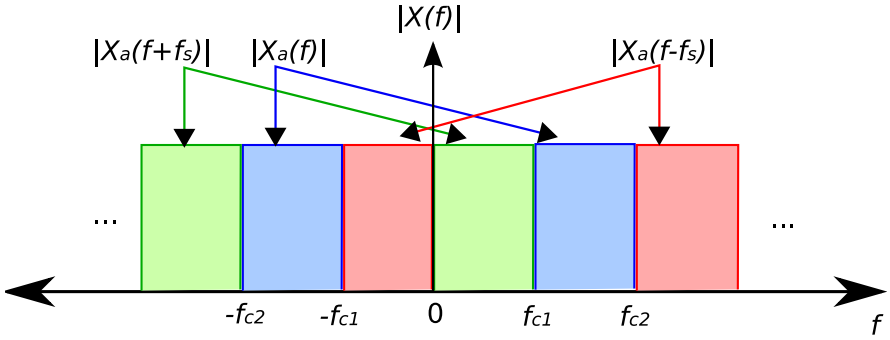


Figure B.6: *Spectrum of sampled bandpass signal with* $f_s = 2f_{c1}$.

aliasing.

Notice that by sampling at $f_s = 2f_{c1}$ instead of at $f_s = 2f_{c2}$ we have effectively moved the bandpass spectrum to baseband. The Nyquist interval being smaller, $[-f_{c1}, f_{c1}]$ rather than $[-f_{c2}, f_{c2}]$, means that if we want to modify the signal in some way through signal processing algorithms it will require less computation to do so than if we had sampled at $f_s = 2f_{c2}$. Because of the periodicity of the spectrum, the processing we performed at baseband is replicated in the bandpass replicas.

Notice that we have made some assumptions in order for bandpass sampling to work. We have assumed that $f_{c1} = f_{c2} - f_{c1}$ so that when we sample at $f_s = 2f_{c1}$, all spectral replicas align nicely without overlapping. This will happen for any f_s such that $f_s = 2k(f_{c2} - fc1)$, where k is some positive integer. Nevertheless, aliasing may occur if we choose some other sampling frequency even if it is larger than the minimal f_s that can be chosen to avoid aliasing.

B.2 Sampling a discrete-time signal: downsampling

We have seen that sampling an analog signal results in spectral replicas and possible aliasing if the sampling frequency is not large enough.

We now move on to explore what happens when we sample a signal that is already sampled, i.e. a discrete-time signal. More precisely, given a sequence $x[n]$, we downsample the sequence by a factor of M, where M is a positive integer. In other words, we form the sequence $x_d[n]$ by keeping one out of every M samples of $x[n]$,

$$x_d[n] = x[Mn].$$

We will see that just as sampling an analog signal results in spectral replication, sampling a discrete-time signal $x[n]$ results in its spectrum, $X(f)$, being replicated as well. The replicas of $X(f)$ are centered at multiples of the sampling frequency of the downsampled signal.

Just as with sampling, the spectral replicas that form the spectrum of a downsampled signal will overlap in general and generate aliasing when added together. However, once again just as with sampling, if the two-sided bandwidth is less or equal to the downsampled rate (within a Nyquist interval), the spectral replicas do not overlap and no aliasing occurs.

The fact that aliasing may occur when downsampling should be obvious. After all, if $x[n]$ was sampled at a frequency just high enough to avoid aliasing, throwing out $M-1$ out of every M samples is equivalent to having sampled the analog signal at a frequency M times lower than what is required to avoid aliasing. On the other hand, if $x[n]$ has been oversampled by at least a factor of M, then no aliasing should occur when downsampling.

In order to derive the relation between the spectra, consider two sampled versions of $x_a(t)$. In one case, we sample with a sampling frequency $f_s = 1/T$, $x_d[n] = \{x_a(nT)\}$, while in the other case we sample with a sampling frequency $f_s' = Mf_s = 1/T' = M/T$, $x[n] = \{x_a(nT')\}$.

For illustration purposes, assume (without loss of generality) that f_s corresponds to the critically sampled case so that all spectral replicas are adjacent to each other. The signal $x[n]$ is oversampled by a factor M.

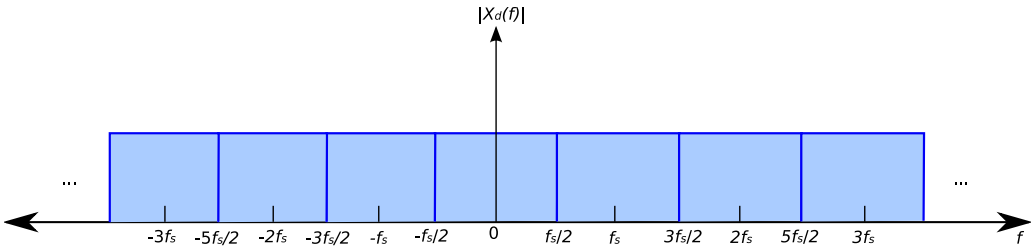


Figure B.7: *Critically sampled signal.*

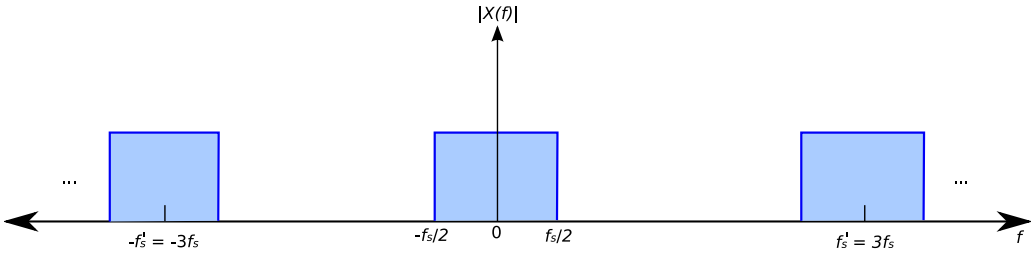


Figure B.8: *Three times oversampled signal.*

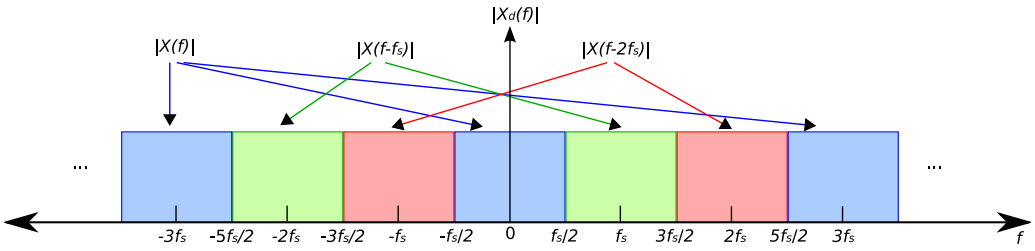


Figure B.9: *Spectrum resulting from downsampling an oversampled signal.*

As an example assume $M = 3$. The spectrum of $x_d[n]$, $X_d(f)$, would be something like that shown in Figure B.7. The spectrum of $x[n]$, $X(f)$, has the same baseband replica, but the remaining replicas are centered around multiples of f_s' as shown in Figure B.8.

In order to construct $X_d(f)$ from $X(f)$, we add $X(f)$ with two shifted versions of it, $X(f - f_s)$ and $X(f - 2f_s)$. The resulting spectrum is shown in Figure B.9.

Of course we can also show this formally for any value of M through

the use of the Poisson summation formula. We know that

$$X_d(f) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X_a(f - kf_s)$$

and

$$X(f) = \frac{1}{T'} \sum_{k=-\infty}^{\infty} X_a(f - kf_s')$$

after some algebraic manipulation, we can arrive at the discrete version of the Poisson summation formula,

$$X_d(f) = \frac{1}{M} \sum_{k=0}^{M-1} X(f - kf_s). \tag{B.3}$$

Note the scaling factor $1/M$. When we downsample a signal, the baseband replica retains the same shape (hence the same information) as long as no aliasing occurs. However, all replicas are scaled down by the downsampling factor.

B.2.1 Filtering to avoid aliasing when downsampling

If the bandwidth of the signal $x[n]$ is larger than $1/M$ times the Nyquist interval, aliasing will occur if such signal is downsampled by a factor of M. In order to avoid aliasing, a standard result in multirate signal processing is to pass the signal $x[n]$ through a lowpass filter prior to downsampling. The role of this filter is akin to that of the analog anti-alias filter used to bandlimit an analog signal prior to sampling. The cutoff frequency for the filter should be π/M.

While this result is correct, reversing the thinking on this is perhaps more useful and sometimes overlooked. By filtering a signal in a way that its bandwidth is reduced we are basically acknowledging that we no longer are interested in part of its bandwidth (more precisely in the information contained therein). Therefore, if we reduce the bandwidth of a signal by a factor of M, we should reduce its sampling rate accordingly in order to reduce the size of the Nyquist interval. The point is we want to eliminate white-space in the spectrum since we know that white-space is akin to an oversampled signal and the larger Nyquist interval that goes with an oversampled signal implies extra (unnecessary) computation.

The distinction is subtle but important. We are used to thinking: "I need to downsample therefore I should filter first". That's fine, but we should also think: "I have reduced the bandwidth of a signal therefore I should downsample". In fact, as we will see in Chapter 4, we should do this even if the factor by which we reduce the bandwidth is not an integer.

B.2.2 Downsampling bandpass signals

Just as with sampling, under certain conditions it is possible to downsample a bandpass signal and avoid aliasing. If the signal is not bandpass already, we can filter it with a bandpass filter that reduces the bandwidth to $1/M$ times the Nyquist interval and then downsample it as long as the passband of the resulting signal lies in one of the intervals

$$k\pi/M < |\omega| < (k+1)\pi/M \quad k = 0,1,\ldots,M-1.$$

The downsampling will cause the bandpass spectrum to be moved to baseband. This is a useful technique that allows a spectrum to be translated in frequency without the need for any multiplications. Without downsampling, it would be necessary to multiply the bandpass signal by a sinusoid if we wanted to translate its spectrum to baseband.

B.3 Increasing the sampling rate of a signal

We have seen that sampling the same analog signal at two different rates produces the same baseband spectrum plus replicas centered around multiples of the sampling frequency. Figure B.7 shows the spectrum of a critically sampled signal, while Figure B.8 shows the spectrum of the same signal oversampled.

In those figures, the oversampling factor is 3, but the general principle applies to any integer oversampling factor L. The question is how to increase the sampling rate of a signal *without* converting the signal back to the analog domain and resampling at a higher rate.

Increasing the sampling rate is just a digital lowpass filtering problem. The filter must keep the baseband spectrum and all replicas of Figure B.8, but remove all remaining replicas, i.e. for the case of 3x oversampling, remove all replicas in Figure B.7 that are *not* in Figure B.8.

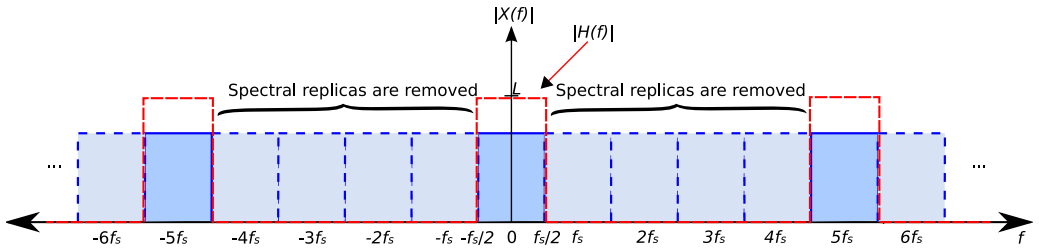


Figure B.10: *Frequency response of filter used to increase the sampling rate by a factor* $L = 5$.

In order to do so, the lowpass digital filter needs to operate at the high sampling rate so that it has replicas of its own centered around multiples of the high sampling rate that will preserve the sampled replicas at such frequencies.

It is well-know that the cutoff frequency for the lowpass filter should be π/L. However, rather than memorizing that formula it is helpful to realize that the cutoff frequency must be selected in such a way that no information is lost. This should be obvious since we are trying to reconstruct samples from the analog signal based on the samples we have. Therefore, the baseband spectrum must be left untouched. The lowpass filter must remove the $L-1$ adjacent replicas.

The (ideal) lowpass filter specifications used to increase the sampling rate by a factor L are thus,

$$H(f) = \begin{cases} L & \text{if } |f| \leq f_s/2, \\ 0 & \text{if } f_s/2 < |f| \leq Lf_s/2. \end{cases} \tag{B.4}$$

where f_s is the sampling rate before we increase the rate and Lf_s is the sampling rate that results after increasing the rate. The gain of L in the passband of the filter is necessary since as we saw the spectrum of the low-rate signal has a gain that is L times smaller than that of the high-rate signal.

The frequency response of the filter used to increase the sampling rate by a factor $L = 5$ is shown in Figure B.10. The filter removes $L-1$ spectral replicas between the replicas centered around multiples of the high sampling rate Lf_s.

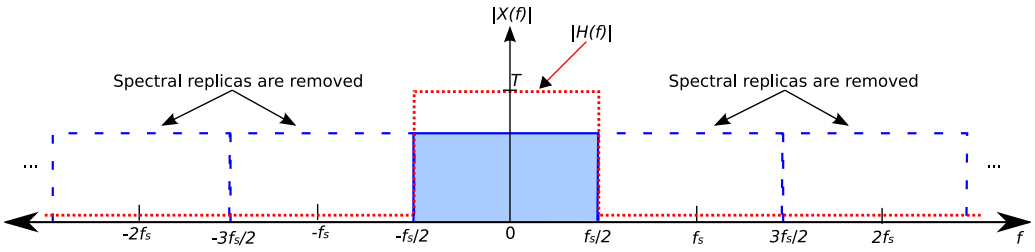


Figure B.11: *Spectral characteristics of lowpass filter used for analog reconstruction.*

B.4 Reconstructing an analog signal

Once signal processing algorithms have been performed on a sampled signal in order to modify it in some desired way, it may be necessary to reconstruct the analog signal corresponding to the samples at hand.

Just as with increasing the sampling rate of a signal, the mechanism to reconstruct an analog signal is obvious if we look at the situation in the frequency-domain. We'd like to take $X(f)$, the spectrum of the sampled signal $x[n]$, and obtain $X_a(f)$, the spectrum of the band-limited analog signal $x_a(t)$. Since

$$X(f) = X_s(f) = \frac{1}{T} \sum_{k=-\infty}^{\infty} X_a(f - kf_s).$$

In the most common case, we need to remove all the replicas, $X_a(f - kf_s)$, $k \neq 0$ and only retain the baseband spectrum $X_a(f)$. However, if we sampled a bandpass signal, and we wish to reconstruct it, we would want to retain a replica for $k \neq 0$ instead.

In order to retain the baseband spectrum, we need a lowpass filter. The lowpass filter must be analog in order for it to not have spectral replicas of its own and thus be able to remove replicas centered at arbitrarily large multiples of f_s.

The role of the lowpass filter $H(f)$ is depicted in Figure B.11. Its specifications are

$$H(f) = \begin{cases} T & \text{if } |f| \leq f_s/2, \\ 0 & \text{otherwise.} \end{cases} \tag{B.5}$$

B.5 Practical sampling and analog reconstruction

The fact that we can reconstruct a band-limited analog signal from its samples provided we sample the signal frequently enough leads us to conclude that no information is lost when we sample the signal in such manner. However, a truly band-limited signal requires a filter with infinite attenuation in its stopband. Moreover, the signal must be of infinite length in order to have a finite frequency spectrum, so even if could construct a filter with infinite attenuation, it would not be enough to obtain a truly band-limited signal. In reality there is always some degree of aliasing introduced when sampling.

Nevertheless, let us assume that we had a perfectly band-limited analog signal that we have sampled in such a way that no aliasing occurs. In order to reconstruct the analog signal, we would require an ideal lowpass filter with infinite attenuation and a perfect brickwall response with no transition gap between passband and stopband.

In reality, both the anti-aliasing filter used to band-limit a signal prior to sampling and the lowpass filter used for analog reconstruction have both a finite amount of stopband attenuation and a non-zero transition gap between passband and stopband.

The finite attenuation can be dealt with by achieving enough attenuation so that the degrees of aliasing and distortion are tolerable. In many cases, oversampling techniques are used so digital filters provide some of the burden of attenuating the unwanted frequency components. Different applications may require different attenuation levels.

In order to deal with the non-zero transition of the filters, excess bandwidth is required. In other words, in practice, instead of sampling at twice the largest frequency of interest, we give ourselves some extra room by sampling at a slightly larger rate. The idea is illustrated in Figure B.12. The upper graph in the Figure shows the analog spectrum $|X_a(f)|$ after the analog signal has been filtered with an analog anti-aliasing prefilter. The anti-alias filter has a passband that coincides with the band of interest. Since the filter is non-ideal, it has a transition band that extends beyond the band of interest. The further we can push out the transition band, the easier it is to construct the analog prefilter (a lower order is required, which in turn means fewer analog components). Notice that the sampling frequency is *not* selected to be twice the stopband-edge of the anti-alias prefilter, $f_{\max}$. It is selected to be twice the frequency that lies in the mid-

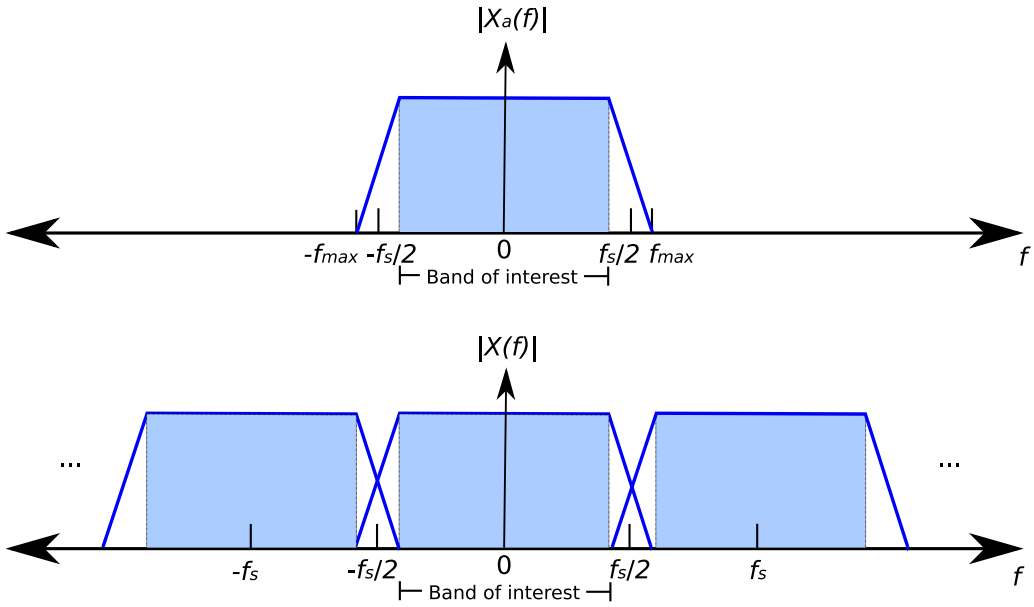


Figure B.12: *Practical sampling with excess bandwidth.*

point between the maximum frequency of interest and $f_{\max}$. As a result, sampling introduces aliasing in the excess frequency band. However, the aliasing does not corrupt any of the frequencies of interest. The magnitude spectrum of the signal sampled in the way just described, $|X(f)|$, is shown in the lower graph of Figure B.12.

The reason we select the sampling frequency in such a way that aliasing occurs in the excess bandwidth region is that we would like to keep the sampling frequency as low as possible. This is because we will have a smaller number of samples to deal with. Any signal processing algorithms performed on the sampled signal will be less computationally intensive as a result of having fewer samples.

Once the signal needs to be converted back to the analog domain, a lowpass analog reconstruction filter is needed. The design of this filter also takes advantage of the excess bandwidth available. Figure B.13 shows the requirements for a reconstruction filter. The filter takes the sampled signal with magnitude spectrum $|X(f)|$, and reconstructs the analog signal with magnitude spectrum $|X_a(f)|$ by removing all spectral replicas. As with the anti-alias prefilter, the passband of the reconstruction filter should extend up to the highest frequency of interest. The transition band require-

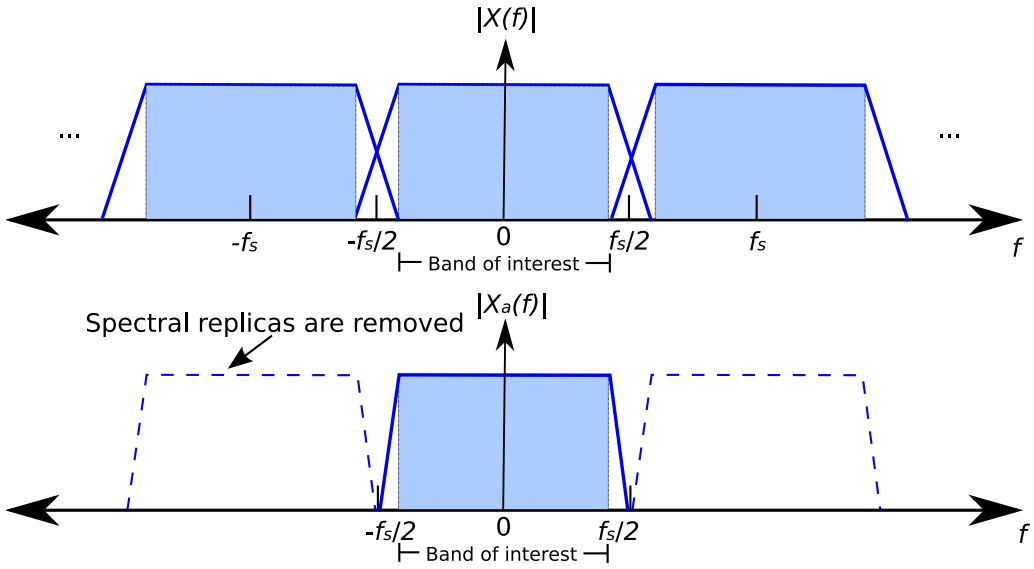


Figure B.13: *Practical reconstruction with excess bandwidth.*

ments will depend on how harmful the frequencies present in the excess bandwidth are for the application at hand. If necessary, the frequencies between the band of interest and $f_s/2$ can be attenuated digitally with a digital lowpass filter prior to analog reconstruction. This would allow a larger transition region for the reconstruction filter, making it simpler to build.

Although we won't get into details, it is worth noting that the reconstruction filter is usually built in two stages. The first stage is usually a simple staircase reconstruction that converts the signal to analog by holding each sample's value until the next sample. This simple filter has a poor lowpass response that allows for some high-frequency components to remain. These high frequency components are removed by the second stage filter. Increasing the sampling rate of the signal digitally prior to reconstruction further eases the job required by the reconstruction filter. We will discuss this next.

B.5.1 Oversampling

As we pointed out in the precious section, the further we can push out the stopband-edge frequency of the anti-alias analog filter, the easier it is for

such filter to be built. The idea can be extended to 2x or 4x oversampling (or more). In such cases, the stopband-edge can be pushed out all the way to two times or four times the maximum frequency of interest plus an extra 10 to 20 percent cushion.

There are two main consequences to doing this. First, since we oversample, we have more samples to deal with than we'd like. Usually we avoid having more samples than necessary, but in this case we purposely generate more samples in order to simplify the anti-aliasing filter. Second, since the anti-alias filter has a transition band that extends all the way to twice or more the band of interest, there is poor out of band attenuation, i.e. many undesired frequencies are present.

In order to deal with both of these issues, we use a combination of digital lowpass filtering plus downsampling. The filtering takes care of the poor out of band attenuation and allows for the downsampling to occur without aliasing corrupting the band of interest. Aliasing does occur when downsampling, but it is limited to the transition band of the digital filter, which overlaps with its spectral replicas. The whole idea is illustrated for the case 2x oversampling in Figure B.14. For this 2x case, the digital filtering can be performed with a very efficient halfband filter. The magnitude spectrum $|X(f)|$ shown in the Figure corresponds to the oversampled signal. The digital-lowpass-filtered signal's spectrum is labeled $|X'(f)|$, while the downsampled signal's spectrum is labeled $|X_d(f)|$. The implementation of the digital filtering and downsampling is always done in an efficient manner (polyphase or otherwise) as explained in Chapter 4. Note that the spectrum $|X'(f)|$ is conceptual since in practice we do not filter first and then downsample (that would be wasteful). It is shown for illustration purposes. In reality we go directly from $|X(f)|$ to $|X_d(f)|$ using an efficient decimation filter.

As we just saw, oversampling eases the job of the anti-aliasing analog prefilter by placing some of the burden on the filter used for decimation. Similarly, increasing the sampling rate can be used to ease the job of the analog reconstruction filter when we are performing digital to analog conversion. The idea is completely analogous to that of oversampling in analog to digital conversion. The burden of removing spectral replicas is shared between the digital filter used for increasing the sampling rate and the analog reconstruction filter.

The digital filter removes the adjacent replicas. If the sampling rate is increased by a factor of L, the filter removes $L-1$ adjacent replicas.

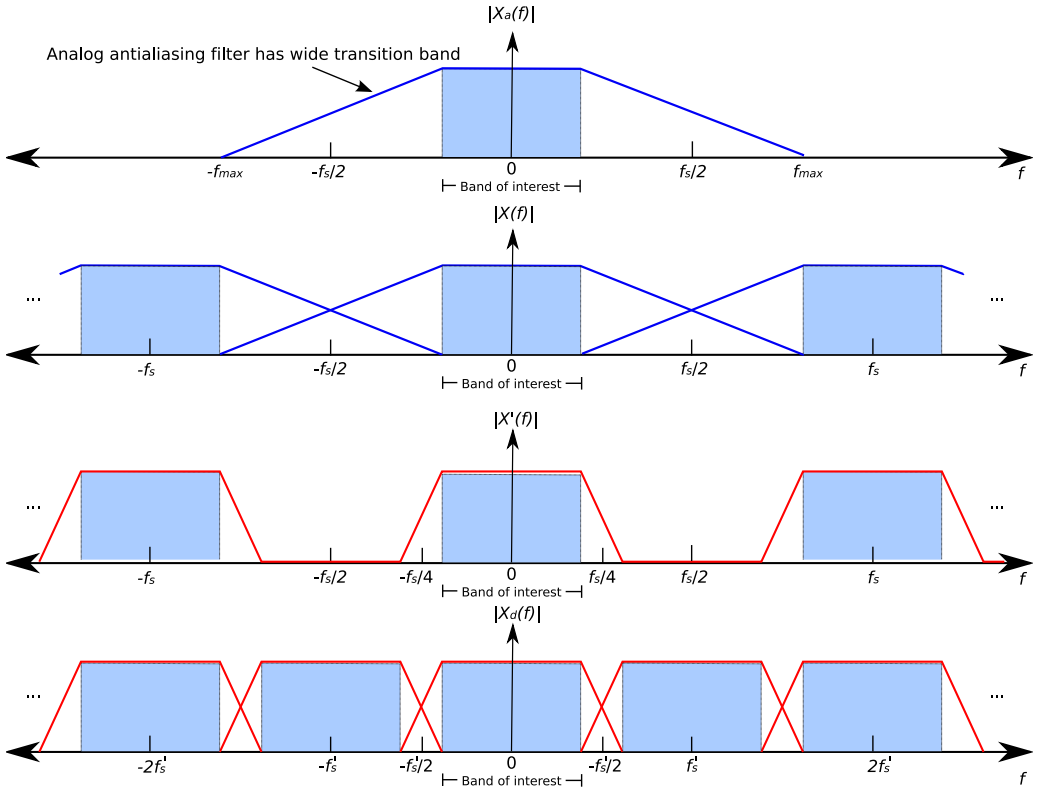


Figure B.14: *Illustration of 2x oversampling plus decimation.*

The analog reconstruction filter has to removed the remaining replicas that cannot be removed by the digital filter. Nevertheless, since the adjacent replicas have been removed, this allows for a wide transition width for the analog reconstruction filter which, once again, enables us to use a simple low-order filter that is easy (and cheap) to build.

The situation for the case $L = 2$ is illustrated in Figure B.15. $|X_d(f)|$ represents the magnitude spectrum of the digital signal we wish to convert to analog. This signal has been sampled (possibly downsampled as well) in such a way that the excess bandwidth overlaps with the replicas as previously discussed. $|Y(f)|$ represents an optional step. Depending on the application, we may need to filter some of the components in the band outside of the band of interest. This should be done at the lowest rate possible so that the transition width relative to the sampling rate is as large as possible (lowering the order required for the filter). Next, we

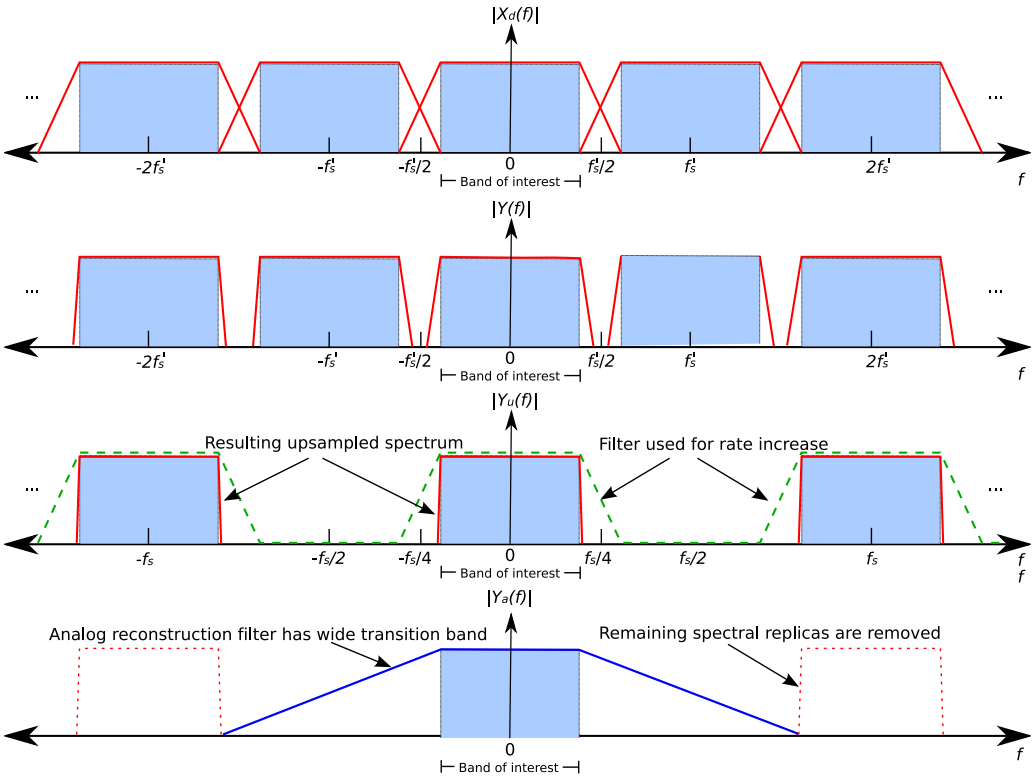


Figure B.15: *Analog reconstruction by first increasing the sampling-rate by a factor of* $L = 2$.

increase the sampling rate by two. The magnitude response of the filter used to remove the adjacent replicas is shown by the dashed line. Note that this filter also attenuates components out of the band of interest. It can be designed so that it does all the work done to get $|Y(f)|$ in addition to removing spectral replicas in one step. The resulting magnitude spectrum of the upsampled signal is represented by $|Y_u(f)|$. The remains spectral replicas are now widely separated so that an analog reconstruction filter with a wide transition band can be used for the final digital to analog conversion.

As mentioned before, the analog reconstruction filter is usually built in two stages. The staircase reconstruction filter causes some distortion in the band of interest. The more we increase the sampling rate prior to reconstruction, the less the distortion. If necessary, the distortion can be compensated for by pre-equalizing the signal digitally prior to sending it

through the staircase reconstructor. To do so, the filter used for increasing the rate is designed in such a way that it boosts the signal over a part of the band of interest that will be attenuated by the staircase reconstructor.

Finally, we mention that in practice noise-shaping sigma-delta quantizers are usually used in conjunction with the techniques outlined here in order to reduce the number of bits required to represent the signal at high sampling rates. The noise-shaping allows for the use of a smaller number of bits *without* decreasing the signal-to-noise ratio.

Appendix C

Case Study: Comparison of Various Design Approaches

Consider the following lowpass specs:

1. Cutoff frequency: 3.125 MHz
2. Transition width: 0.4 MHz
3. Maximum passband ripple: 1 dB
4. Minimum stopband attenuation: 80 dB
5. Input sampling frequency: 100 MHz

The table below compares the implementation cost of various single-rate designs. The following command is used for the specifications:

```
Fs  = 100e6;
TW  = 0.4e6;
Fp  = 3.125e6 - TW/2;
Fst = 3.125e6 + TW/2;
Ap  = 1;
Ast = 80;
Hf = fdesign.lowpass(Fp,Fst,Ap,Ast,Fs);
```

	NMult	NAdd	Nstates	MPIS	APIS	NStages	Decim. Fact
ellip	20	20	10	20	20	1	N/A
equiripple	642	641	641	642	641	1	N/A
IFIR	138	136	759	138	136	2	N/A
IFIR JO	115	113	680	115	113	2	N/A

IFIR JO denotes that the joint optimization option is used with the IFIR design. See Chapter 5 for more on this.

Next, we compare 3 designs that include decimation by a factor of 15. The following command sets the specs.:

```
Hf = fdesign.decimator(15,'lowpass',Fp,Fst,Ap,Ast,Fs);
```

	NMult	NAdd	Nstates	MPIS	APIS	NStages	Decim. Fact
equiripple	642	641	630	42.8	42.7333	1	15
IFIR JO	224	222	216	17.333	16.933	2	15
multistage	169	167	161	14.6	14.3333	2	15

Finally, we compare one CIC multistage design and 3 multistage Nyquist designs. This line is used to set the specs for the Nyquist designs:

```
Hf = fdesign.decimator(16,'nyquist',16,TW,Ast,Fs);
```

For the CIC design, we break it down in 3 stages. The first stage, which is the CIC decimator, will provide a decimation of 4. The next stage, the CIC compensator, will provide decimation of 2. The final stage will be a regular halfband, also providing decimation by 2.

The code is as follows:

```
% CIC decimator design
M1 = 4;
D = 1; % Differential delay
Hf1 = fdesign.decimator(M1,'cic',D,Fp,Ast,Fs);
Hcic = design(Hf1,'multisection');
% CIC compensator design
M2 = 2;
Nsecs = Hcic.NumberOfSections;
Hf2 = fdesign.decimator(M2,'ciccomp',D,...
      Nsecs,Fp,Fs/(M1*M2)-Fp-TW,Ap,Ast,Fs/M1);
Hcomp = design(Hf2,'equiripple');
```

```
% Halfband design
M3 = 2;
Hf3 = fdesign.decimator(M3,'halfband',...
      TW,Ast,Fs/(M1*M2));
Hhalf = design(Hf3,'equiripple');
Hcas = cascade(Hcic,Hcomp,Hhalf); % Overall design
```

Specifying the Nyquist band to be 16 implies that the cutoff frequency is $\frac{\pi}{16}$ which corresponds to the 0.0625π required. This value is almost always set to be the same as the decimation factor.

All Nyquist designs result in a 4-stage filter with halfband decimators in each stage. Note that the Nyquist designs far exceed the passband ripple specification (1 dB) even though it is not set explicitly. In the case of IIR halfbands, the passband ripple is of the order of 10^{-8} dB.

	NMult	NAdd	Nstates	MPIS	APIS	NStages	Decim. Fact
CIC multistage	86	94	166	6.0625	12.125	3	16
FIR equirip	98	94	180	10.325	9.375	4	16
IIR ellip	14	28	22	2.3125	4.625	4	16
IIR lin phase	37	74	80	3.8125	7.625	4	16

Appendix D

Overview of Fixed-Point Arithmetic

Overview

In this appendix we provide a brief overview* of fixed-point arithmetic and its implications on digital filters.

The basic trade-off that one should understand whenever dealing with fixed-point arithmetic is maximizing SNR vs. minimizing the possibility of overflows occurring. As is so often the case, these two requirements are in opposition to each other and a compromise needs to be reached.

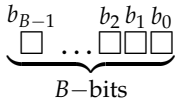
The SNR in this realm refers to signal to quantization noise ratio. The SNR can be increased in two ways: increasing the signal power by maximizing the use of the available dynamic range or decreasing the noise power by using more bits to represent data. Of course, using more bits is not always an option. Hardware constraints may pre-determine how many bits are available to represent a given quantity.

The issue with increasing the utilization of the available dynamic range is that a signal can easily overflow when some arithmetic is performed on it.

* For a more complete coverage, Chapter 2 of [3] and Chapter 8 of [36] are recommended.

D.1 Some fixed-point basics

Consider a register used to store a fixed-point number,



The register has B bits (it has a wordlength of B), the value of the kth bit (from right to left) is given by b_k which can obviously be only 0 or 1. The bits themselves are not enough to determine the value being held in the register. For this, we need to define the scaling of the bits. Specifically, a two's complement fixed-point number stored in such a register has a value given by

$$\text{value} = -b_{B-1}2^{B-1-F} + \sum_{k=0}^{B-2} b_k 2^{k-F} \tag{D.1}$$

where F is a (positive or negative) integer that defines the scaling of the bits. F is referred to as a fraction length (or `FracLength`) but this is somewhat of a misnomer. The reason is that F is not necessarily a length (for instance if F is negative or if F is greater than B). However, when F is positive and $F \leq B$, it will be equal to the number of bits to the right of the binary point hence the term fraction length.

From (D.1), we can see that the value of a fixed-point number is determined by assigning weights of the form 2^{-m} to each bit. The rightmost bit, b_0 has the smallest weight, 2^{-F}. This bit is called the least-significant bit (LSB). The leftmost bit, b_{B-1}, has the largest weight, 2^{B-1-F}, which is why it is called the most-significant bit (MSB).

Given the bit values, b_k, the pair $\{B, F\}$ completely characterizes a fixed-point number, i.e. suffices in determining the value that the bits represent.

D.2 Quantization of signals and SNR

Fixed-point representation can be used to represent data in any range. With two's complement representation, the most negative value that can be represented is given by the scaling of the MSB. The most positive value that can be represented is given by the sum of all the weights of all bits except for the MSB.

Figure D.1:

For example, with 4 bits available and a fraction length of 8 (so that the scaling of the LSB is 2^{-8} and the scaling of the MSB is 2^{-5}), the most negative value that can be represented is -2^{-5} and the most positive value that can be represented is $2^{-6} + 2^{-7} + 2^{-8}$ (which is equal to $2^{-5} - 2^{-8}$). In total, we can represent $2^4 = 16$ numbers between -2^{-5} and $2^{-5} - 2^{-8}$ with a resolution or increment of 2^{-8} between each number and the next consecutive one.

In general, the number of values that can be represented within those limits is given by 2^B. The step size or increment between the values represented is given by $\varepsilon = 2^{-F}$. So for a given dynamic range to be covered, the number of bits determines the granularity (precision) with which we can represent numbers within that range. The more bits we use, the smaller the quantization error when we round a value to the nearest representable value. As the number of bits increases, the SNR increases because the quantization noise decreases.

Specifically, consider Figure D.1. The range that is covered is denoted by R. The quantization step is denoted $\varepsilon = 2^{-F}$. The number of values that can be represented is given by 2^B. There are only two degrees of freedom in these quantities which are related by

$$\frac{R}{2^B} = \varepsilon$$

If we round all values to their nearest representable value, the maximum error introduced by such rounding is given by $\varepsilon/2$. If R is constant,

and B increases, then ε and consequently the largest quantization error decreases.

A simple calculation of SNR when quantizing a signal can be performed as follows [3]: If R is fully covered by the signal in question, the signal strength is R. The quantization error covers the interval $[-\varepsilon/2, \varepsilon/2]$, i.e. it has a range of ε, so the SNR is given by

$$\text{SNR} = \frac{R}{\varepsilon}.$$

In dB, we have

$$\text{SNR} = 20\log_{10}\left(\frac{R}{\varepsilon}\right) = 20\log_{10}(2^B) = 6.02B$$

This result is known as the *6 dB/bit rule* [3]. The result provides an upper bound on the achievable SNR when using B bits to cover a given range R. To achieve this upper bound, it is necessary that the signal being quantized can take any value within that range with equal probability. In other words, it is necessary that the signal behaves like uniformly distributed white noise over that range.

Example 73 *Consider the following quantization of a signal which covers the range* $[-1,1)$ *with equal probability,*

```
x   = 2*(rand(10000,1)-0.5);
xq  = fi(x,true,16,15);
SNR = 10*log10(var(x)/var(x-double(xq)))
SNR =
96.3942
```

We use 16 bits to represent the signal and the step-size is 2^{-15}*. The SNR we can achieve is in-line with the 6 dB-per-bit rule.*

Consider what would happen if instead of uniform white noise, we take samples from a Gaussian white noise signal and ensure they fall within the interval $[-1,1)$,

```
x   = randn(10000,1);
x   = x/norm(x,inf);
xq  = fi(x,true,16,15);
```

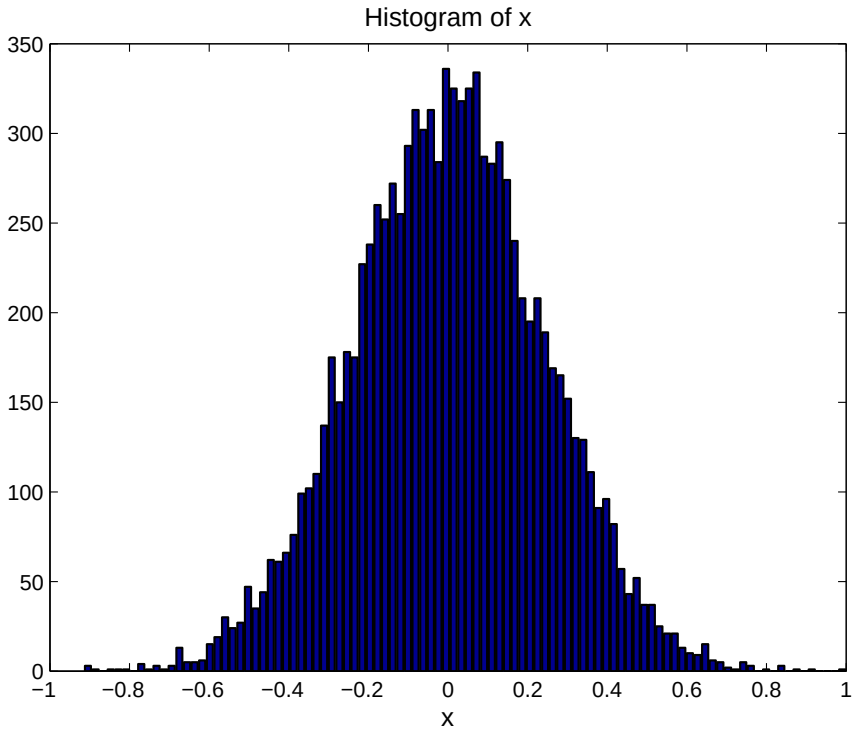


Figure D.2: *Histogram of a white-noise signal normalized so that the maximum values is* ± 1.

```
SNR = 10*log10(var(x)/var(x-double(xq)))
SNR =
   89.9794
```

The SNR is no longer satisfying the 6 dB-per-bit rule. The reason is that the signal very rarely takes values close to the largest available range (i.e. close to -1 or 1). This can easily be verified by looking at the histogram:

```
[n,c]=hist(x,100);bar(c,n)
```

The histogram for the white-noise case is shown in Figure D.2. On average the signal strength is not that large, while the quantization error strength remains the same. In other words, the quantization error can be equally large (on average) whether `x` takes a small or a large value. Since

`x` takes small values more often than it take large values, this reduces the SNR.

Indeed, if we look at the `var(x)` for the uniform case, it is much larger than `var(x)` in the Gaussian case, while `var(x-double(xq))` is the same on both cases.

While it may seem not so useful to consider white noise examples, it is worth keeping in mind that many practical signals that are quantized, such as speech and audio signals, can behave similar to white noise.

In summary, depending on the nature of the signal being quantized, the achievable SNR will vary. In some cases, depending on whether the application at hand permits it, it may be preferable to allow for the occasional overflow with the aim of increasing the SNR on average for a given signal.

D.2.1 Quantizing impulse responses for FIR filters

When implementing an FIR filter with fixed-point arithmetic, the first thing we need to do is quantize the filter's coefficients, i.e., its impulse response. Since the impulse response of most FIR filters has a few large samples* and many small samples, on average we can't expect to get the full 6 dB/bit. In practice we can expect somewhere between 4.5 and 5.5 dB/bit when quantizing FIR filter coefficients.

Example 74 *Consider the following highpass filter design:*

```
Hf = fdesign.highpass('Fst,Fp,Ast,Ap',0.5,0.6,80,.5);
Hd = design(Hf,'equiripple');
h  = Hd.Numerator; % Get the impulse response
norm(h,inf)
```

The impulse response is shown in Figure D.3. The largest sample has a magnitude of 0.434. Therefore, our fixed-point settings should be such to cover the range $[-0.5, 0.5)$ *since this is the smallest power-of-two interval that encompasses the values of the impulse response. This can be accomplished by setting F to be equal to B.*

Now let's compute the SNR for various values of B and see what SNRs we get. We will perform the following computation for different values of B:

* Typically, the middle sample dominates for linear-phase filters.

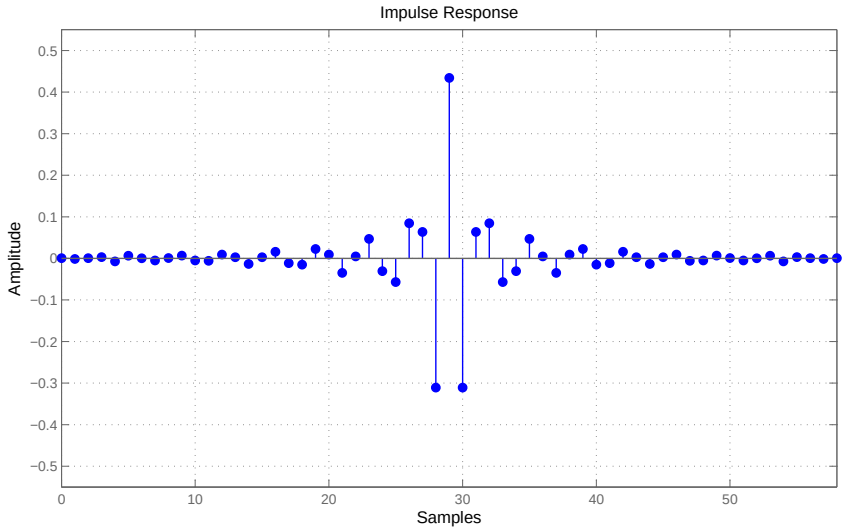


Figure D.3: *Impulse response of highpass filter. The largest coefficient has a magnitude of 0.434.*

```
B   = 8; % change to 10, 12, 16, 18, 24
F   = B;
hq  = fi(h,true,B,F);
SNR = 10*log10(var(h)/var(h-double(hq)))
```

If we divide the SNR by B, for the various values of B shown, we get the following values of dB/bit: 4.7690, 4.9704, 5.1000, 5.3796, 5.3760, 5.5890.

D.3 Fixed-point arithmetic

Let's look at a couple of things to keep in mind when performing additions/multiplications with fixed-point arithmetic.

D.3.1 Fixed-point addition

In general, when adding two fixed-point numbers, in order to avoid round-off error it is necessary to increase the wordlength of the result by one bit. This is to account for the possibility of a carry resulting from the addition.

For example suppose we want to add two 4-bit numbers that fall in the range $[-1,1)$, say $x_1 = 0.875$ and $x_2 = 0.75$. x_1 has the following binary representation (with $F = 3$): 0111. x_2 has 0110 as its binary representation.

The sum of the two is 1.6250. We need to represent it with 5 bits so that no roundoff error is introduced. The binary representation of the result is 01101. Note that in general because of the possibility that the LSB is equal to 1, it is not possible to discard the LSB without introducing roundoff error, hence the bit growth.

Many filter structures contain a series of additions, one after the other. However, it is not necessary to increase the wordlength by one bit for every addition in the filter in order to avoid roundoff (i.e., in order to have full precision). If N is the number of additions, it is only necessary to increase the wordlength by [*]

$$G = \lfloor \log_2(N) \rfloor + 1. \tag{D.2}$$

To see this, consider the addition of several numbers represented with 16-bit wordlength and a fraction length of 15. The worst case scenario is when the actual numbers are the largest possible representable number for a given format. Consider the largest positive value, 0.99[†]. If we add two of these numbers, the result will be 1.999. To avoid overflow, it is necessary to cover the range $[-2,2)$ in order to represent the result. In order to have no loss of precision, i.e. avoid roundoff, we need to increase the wordlength to 17 bits (and leave the fraction length fixed at 15). If we now add 0.99 to the value 1.999 the result is 2.999. Since this value falls outside the range $[-2,2)$, we need to further increase the wordlength by one bit in order to represent the result with no loss of precision. With an 18-bit wordlength and a fraction length of 15 we can represent numbers in the range $[-4,4)$. If we add 0.99 to the result we had so far, 2.999, we get 3.999 as an answer. Since this number falls within the representable range, it is not necessary to add an additional bit in order to represent this number. The next addition would require another bit to be aggregated, but the following three additions would not, and so forth.

[*] The operator $\lfloor \cdot \rfloor$ denotes the largest integer smaller or equal to the value operated on (i.e. the floor command in MATLAB). [†] A similar argument can be made by using the largest negative value.

Tree-adder view of multiple additions

An easy way of understanding this $\log_2$ bit growth when adding is to look at what would happen if we used a tree-like structure to implement the series of additions in a direct form FIR filter. Figure D.4 shows the case of adding 4 B-bit numbers together. If we added one bit for every addition, in order to obtain a full-precision result, we would need to add 3 bits in total.

However, as we can see in Figure D.4, we can get a full-precision result by adding only 2 rather than 3 bits overall. This takes into the account the worst possible case, meaning that we need to add 1 bit every time two numbers are added.

One could say the tree-adder view has the $\log_2$ bit growth built-in to it. The idea still applies when there is not a power-of-2 number of values to be added together. Some branches will be slightly incomplete, but the required bit growth should still be correctly determined.

Note that the (D.2) holds for the worst case scenario in which the numbers being added are the of largest possible magnitude. In Chapter 7, we will see that for FIR filters, we can reduce the bit growth required to avoid roundoff by looking at the actual coefficients of the filter.

D.3.2 Fixed-point multiplication

Since multiplication is nothing but repeated addition, it should be expected that bit-growth is required if we want to multiply two numbers without introducing loss of precision. Just as with addition, the bit growth is independent of the fraclengths involved.

Let's look at the worse case scenario if we multiply a number x_1, represented with B_1 bits, with a number x_2, represented with B_2 bits. Since the bit growth is independent of the fraclength, consider the binary digits involved. In the worst case, we are multiplying the bits in x_1 times -2^{B_2-1}. That means that we add (ignore the sign) x_1 to itself 2^{B_2-1} times. Using (D.2), the bit growth is B_2 bits.

In terms of the fraclength, if x_1 has a fraclength of F_1 and x_2 has a fraclengh of F_2, the LSBs have weights of 2^{-F_1} and 2^{-F_2} respectively. The product of the two will of course have a weight of $2^{-(F_1+F_2)}$.

In summary, given two fixed-point numbers, one with wordlength/fraclength given by $\{B_1, F_1\}$ and the other with wordlength/fraclength given by $\{B_2, F_2\}$,

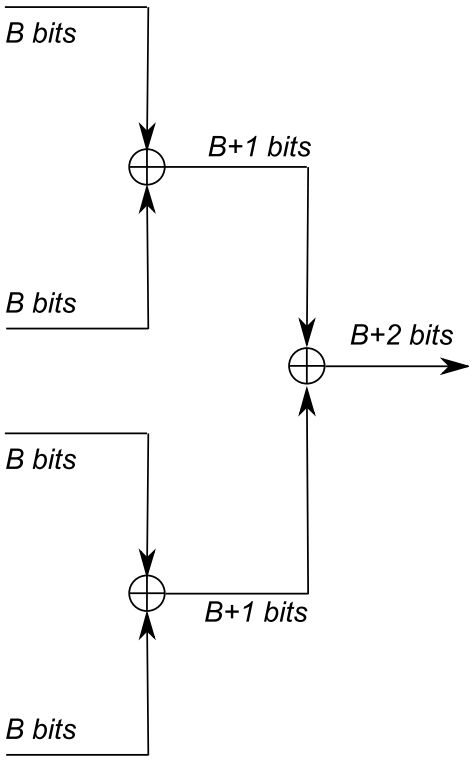


Figure D.4: *Tree-adder view of adding 4 B-bit numbers. Only $B + 2$ bits are needed for a full precision result.*

the product of the two numbers can be represented with no roundoff by using a wordlength/fraclength of $\{B_1 + B_2, F_1 + F_2\}$.

For example, given a number represented with 16 bits and a fraction length of 14 and another number represented with 18 bits and a fraction length of 10, the product of the two numbers can be represented without loss of precision by using a wordlength of 34 bits and a fraction length of 24.

As is the case with additions, in Chapter 7 we will see that for FIR filters, we can reduce the bit growth required to avoid roundoff when multiplying by looking at the actual coefficients of the filter.

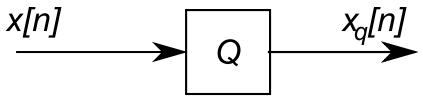


Figure D.5: *Quantization of a signal.*

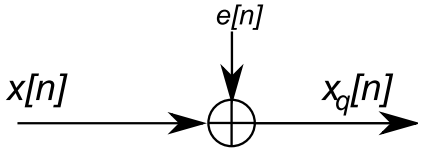


Figure D.6: *Statistical model for the quantization of a signal.*

D.4 Quantization noise variance

In many cases, a good model for the quantization error is as additive noise to the signal being quantized [3].

Specifically, consider the quantization of the signal $x[n]$ as shown in Figure D.5. The operator $Q\{\cdot\}$ takes the input signal and produces a quantized version of it, $Q\{x[n]\} = x_q[n]$.

Statistically, we can model the effect of $Q\{\cdot\}$ as additive noise signal as shown in Figure D.6.

Provided that the input signal changes fast enough, the noise signal itself, $e[n]$, can be modeled as uniformly distributed white-noise over the range $[-\varepsilon/2, \varepsilon/2]$.* The variance of the quantization noise signal can easily be found to be $\varepsilon^2/12$ [3].

Reducing the number of bits used to represent a signal

The results we have presented so far apply to quantizing an analog signal in order for it to be represented with fixed-point. This is of course important when one is considering analog-to-digital conversion.

In digital filters, we don't worry about the quantization of the input signal since this occurs elsewhere. However, it is often the case that we

* Again, we assume here we are rounding to the nearest value when we quantize. If instead we round towards $-\infty$, i.e. we truncate, things change slightly.

need to re-quantize a quantized signal, i.e. reduce the number of bits used to represent it.

Examples where this is common include throwing out bits when moving data from an accumulator (all the additions in a direct-form FIR filter) to the output of the filter because we need to preserve the input signal wordlength at the output. A typical example is keeping only 16 bits for the output from an accumulator that may have anywhere from 32 to 40 bits.

Another common case is in the feedback portion of an IIR filter since otherwise the repeated additions would result in unbounded bit growth.

Of course the bits that we throw out when we re-quantize a signal are the LSBs. This way the same range can be covered after re-quantization and overflow is avoided. The precision is clearly reduced as we throw out LSBs (the quantization step increases from 2^{-F} to $2^{-(F-Q)}$; where Q is the number of bits that are thrown away.

The variance of the quantization noise that we have just derived doesn't quite apply for the case of re-quantizing a quantized signal [36]. However, such variance becomes a very good approximation if at least 3 or 4 bits are thrown out. Since we are typically removing anywhere from 16 bits onward, the $\varepsilon^2/12$ value is usually used without giving it a second thought. The value of ε in this case corresponds of course to the quantization step after re-quantizing.

D.5 Quantization noise passed through a linear filter

When analyzing how quantization affects the output signal of a filter, the following result will come in handy: If a random signal, $x[n]$, with power spectral density (PSD) given by $\mathcal{P}_{xx}(e^{j\omega})$ is filtered through a filter with frequency response $H(e^{j\omega})$, the output of the filter, $y[n]$, will be a random signal with PSD given by,

$$\mathcal{P}_{yy}(e^{j\omega}) = \left|H(e^{j\omega})\right|^2 \mathcal{P}_{xx}(e^{j\omega})$$

In particular, if the input is a white-noise signal, such as the quantization noise, its PSD will be constant, with intensity given by the variance

scaled by 2π, i.e. $\sigma_x^2/2\pi$. In this case, the PSD of the output will be

$$\mathcal{P}_{yy}(e^{j\omega}) = \frac{\sigma_x^2}{2\pi}\left|H(e^{j\omega})\right|^2. \tag{D.3}$$

Therefore, in this case the PSD will take the spectral shape of the squared magnitude-response of the filter.

D.5.1 Computing the average power of the output noise

Once we have computed the PSD of the output due to the quantization noise, we can compute the average power at the output due to such noise by integrating the PSD,

$$P_{\text{AVG}} = \int_{-\pi}^{\pi} \mathcal{P}_{yy}(e^{j\omega})d\omega$$

In the case of FIR filters, when all arithmetic is performed with full precision and therefore the only source of quantization noise is due to the bits being discarded at the output, the transfer function between the noise source and the filter output is trivially $H(e^{j\omega}) = 1$. If the PSD intensity is $\sigma_x^2/2\pi = \varepsilon^2/(2\pi \cdot 12)$, the average power is simply

$$P_{\text{AVG}} = \int_{-\pi}^{\pi} \frac{\varepsilon^2}{2\pi \cdot 12}\left|H(e^{j\omega})\right|^2 d\omega = \frac{\varepsilon^2}{12} \tag{D.4}$$

D.6 Oversampling noise-shaping quantizers

A great deal of emphasis is placed throughout this document on multirate filters. One of the primary uses of decimation filters is in easing the implementation of analog to digital conversion. This process of course involves quantization of signals.

Similarly, one of the primary uses of interpolation filters is to help with digital to analog conversion (it is one of the few reasons to oversample).

In either the decimation or interpolation case, the use of noise shaping quantizers (or re-quantizers) in which multirate techniques are used in conjunction with shaping the quantization noise (via highpass filtering) can reduce the number of bits required to represent a signal *without* decreasing the SNR.

These schemes are widely used in practice. For more information see [3].

Bibliography

[1] R. A. Losada, "Design finite impulse response digital filters. Part I," *Microwaves & RF*, vol. 43, pp. 66–80, January 2004.

[2] R. A. Losada, "Design finite impulse response digital filters. Part II," *Microwaves & RF*, vol. 43, pp. 70–84, February 2004.

[3] S. J. Orfanidis, *Introduction to Signal Processing*. Upper Saddle River, New Jersey: Prentice Hall, 1996.

[4] T. Saramaki, *Handbook for Digital Signal Processing. S. K. Mitra and J. F. Kaiser Eds.*, ch. 4. Finite impulse response filter design, pp. 155–278. New York, New York: Wiley-Interscience, 1993.

[5] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall, 1975.

[6] E. W. Cheney, *Introduction to Approximation Theory*. Rhode Island: American Mathematical Society, 1998.

[7] J. H. McClellan, T. W. Parks, and L. R. Rabiner, "A computer program for designing optimum FIR linear phase digital filters," *IEEE Trans. Audio Electroacoust.*, vol. AU-21, pp. 506–526, 1973.

[8] O. Hermann and H. W. Schüßler, "Design of nonrecursive digital filters with minimum phase," *Electronic Lett.*, vol. 6, pp. 329–330, 1970.

[9] T. W. Parks and C. S. Burrus, *Digital Filter Design*. New York, New York: Wiley-Interscience, 1987.

[10] I. W. Selesnick and C. S. Burrus, "Exchange algorithms that complement the Parks-McClellan algorithm for linear-phase FIR filter design," *IEEE Trans. on Circuits and Systems II*, vol. CAS-44, pp. 137–142, February 1997.

[11] fred harris, *Multirate Signal Processing for Communication Systems*. Upper Saddle River, New Jersey: Prentice Hall, 2004.

[12] R. A. Losada and V. Pellissier, "Designing IIR filters with a given 3-dB point," *IEEE Signal Proc. Magazine; DSP Tips & Tricks column*, vol. 22, pp. 95–98, July 2005.

[13] R. A. Losada and V. Pellissier, *Streamlining Digital Signal Processing: A Tricks of the Trade Guidebook. Edited by R. Lyons*, ch. 4. Designing IIR Filters with a Given 3-dB Point, pp. 33–41. Hoboken, New Jersey: IEEE Press; Wiley-Interscience, 2007.

[14] A. Antoniou, *Digital Filters: Analysis, Design, and Applications*. New York, New York: McGraw-Hill, second ed., 1993.

[15] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*. New York, New York: McGraw-Hill, third ed., 2006.

[16] P. P. Vaidyanathan, *Handbook for Digital Signal Processing. S. K. Mitra and J. F. Kaiser Eds.*, ch. 7. Robust digital filter structures, pp. 419–490. New York, New York: Wiley-Interscience, 1993.

[17] P. A. Regalia, S. K. Mitra, and P. P. Vaidyanathan, "The digital all-pass filter: a versatile signal processing buildingblock," *Proceedings of the IEEE*, vol. 76, pp. 19–37, January 1988.

[18] L. Gazsi, "Explicit formulas for wave digital filters," *IEEE Trans. on Circuits and Systems*, vol. CAS-32, pp. 68–88, 1985.

[19] M. Lutovac, D. Tosic, and B. Evans, *Filter Design for Signal Processing Using MATLAB and Mathematica*. Upper Saddle River, New Jersey: Prentice Hall, 2001.

[20] H. W. Schüßler and P. Steffen, "Recursive halfband-filters," *AEÜ International Journal of Electronics and Communications*, vol. 55, pp. 377–388, December 2001.

[21] H. W. Schüßler and P. Steffen, "Halfband filters and Hilbert transformers," *Circuits, Systems, and Signal Processing*, vol. 17, pp. 137–164, March 1998.

[22] M. Lang, "Allpass filter design and applications," *IEEE Transactions on Signal Processing*, vol. 46, pp. 2505–2514, September 1998.

[23] N. J. Fliege, *Multirate Digital Signal Processing*. New York, New York: John Wiley & Sons, 1994.

[24] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1983.

[25] Y. Neuvo, C.-Y. Dong, and S. K. Mitra, "Interpolated finite impulse response filters," *IEEE Trans. on Acoust. Speech and Signal Proc.*, vol. ASSP-32, pp. 563–570, June 1984.

[26] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*. Englewood Cliffs, New Jersey: Prentice Hall, 1993.

[27] R. Lyons, "Interpolated narrowband lowpass FIR filters," *IEEE Signal Proc. Mag.*, pp. 50–57, January 2003.

[28] T. Saramaki, Y. Neuvo, and S. K. Mitra, "Design of computationally efficient interpolated FIR filters," *IEEE Trans. on Circuits and Systems*, vol. CAS-35, pp. 70–88, January 1988.

[29] R. Lyons, *Streamlining Digital Signal Processing: A Tricks of the Trade Guidebook. Edited by R. Lyons*, ch. 8. Turbocharging Interpolated FIR Filters, pp. 73–84. Hoboken, New Jersey: IEEE Press; Wiley-Interscience, 2007.

[30] T. Saramaki and Y. Neuvo, "A class of fir nyquist (*n*th-band) filters with zero intersymbol interference," *IEEE Trans. on Circuits and Systems*, vol. CAS-34, no. 10, pp. 1182–1190, 1987.

[31] E. B. Hogenauer, "An economical class of digital filters for decimation and interpolation," *IEEE Trans. on Acoust. Speech and Signal Proc.*, vol. ASSP-29, pp. 155–162, April 1981.

[32] R. A. Losada and R. Lyons, *Streamlining Digital Signal Processing: A Tricks of the Trade Guidebook. Edited by R. Lyons*, ch. 6. Reducing CIC Filter Complexity, pp. 51–58. Hoboken, New Jersey: IEEE Press; Wiley-Interscience, 2007.

[33] K. E. Atkinson, *An Introduction to Numerical Analysis*. New York: John Wiley and Sons, second ed., 1989.

[34] R. L. Burden and J. D. Faires, *Numerical Analysis*. Pacific Grove, CA: Brooks/Cole, seventh ed., 2001.

[35] M. Unser, "Sampling—50 years after shannon," *PROCEEDINGS OF THE IEEE*, vol. 88, pp. 569–587, April 2000.

[36] D. Schlichthärle, *Digital Filters: Basics and Design*. Berlin: Springer, 2000.