Software implementation of digital filters

Peter Moylan

Glendale, NSW, Australia

peter@pmoylan.org http://www.pmoylan.org

February 2017

ABSTRACT: This report deals with the problem of designing and implementing digital filters in software. The report is, in effect, documentation for the accompanying software.

1. Introduction

A very common requirement in embedded real-time applications is to implement a filter in software. The point of this document is to show how to do it. We are naturally concerned with real-time applications, where there is often a focus on making the sampling rate as high as possible; but the details described here apply to any sampling rate.

In terms of theory, there is nothing original in this report. All of the theory is well-known and well-understood, to the point where we don't even need to mention primary references. (Fortunately, because I do not have the library access that would allow consulting the primary sources.) Unfortunately, the available sources rarely mention how to design the software to do the job. In descriptions of frequency pre-warping, for example – something that is necessary when converting from a Laplace transform to a discrete-time description – it is not entirely clear whether one should use the warping mapping or its inverse. The easiest way to clarify this is to point to the source code that does the job.

All of the steps described in this report are easy in principle, but tedious and error-prone when one tries to do the algebra. This is the sort of thing that needs to be automated. The novel part of this report is that it can be automated, with no need to worry about transferring design parameters from off-line calculations to the real-time software.

It is natural to distinguish between the *implementation* of a digital filter and the *design* of that filter, because the implementation needs to be made as efficient as possible – especially in applications with high sampling rates – while the design involves more work but has a less pressing deadline. Typically, the design is done off-line and then the parameters are copied manually into the software. I would argue here that this is a bad approach; there is too much scope for error in the copying, and the whole process has to be re-done if the specifications change. My preference is to put the design calculations into the initialisation code of the real-time software. This is an unconventional approach, but in practice it works well. The overhead is small, and the calculations happen only when the software is going through its start-up operations. (Which can be lengthy, because of things like integrity checking and checking that the communications channels are working.) The designer need only specify the basic things like filter bandwidth. The software can deal with the tedious details of turning

this into a filter implementation. The only difference between this and the conventional approach is that we are not copying coefficients from one program to another.

This does not remove the requirement of simulating the filter response before committing the specifications. As we shall see, digital filters become unstable, because of rounding error, if we set the filter order too high, so we need to work out what the order should be. (The order also affects the computation time, which is usually a limited resource in real-time applications.) In addition the anti-aliasing filter, which is typically a combination of analogue and digital filters, needs to be checked to see that it is really rejected the unwanted frequencies.

1.1 The code

The code that accompanies this report can be fetched from [Moy17].

There are two versions, in C and in Modula-2. The tests described in later sections were done using XDS Modula-2 and Open Watcom C. Historically, the C version was developed first – using a compiler that is no longer available to the author – because the target processors had only C compilers. After translation to Modula-2 some errors were found (running off the beginning of an array) that had remained undetected for a few years because C does not have true arrays. Tests on that code led to the belief that the implementation could not run reliably for filter order greater than 8. With the Modula-2 version this stability problem was much reduced, and some improvements were discovered. Finally, the Modula-2 code was translated back into C, giving a much superior version. The two implementations are now almost identical. The Modula-2 version runs faster and can handle higher-order filters, but that is only because the rules of low-level languages like C and C++, and possibly Java, make it harder to do code optimisation. A better C compiler might well get around that problem.

The Modula-2 versions have been used for most of the results in this report, basically because we do not fully trust the C versions.

One difference that can be observed between the C and Modula-2 implementations is the way the memory is allocated for the filter data structure. (This is the data structure that tracks the filter state.) The C code requires the caller to reserve a suitable record. The Modula-2 code dynamically allocates the record. This difference can be explained by a difference in conventions in the two languages – the Modula-2 tradition places a higher priority on information hiding, while the C and C++ traditions prefer revealing the details of structures to the client software – but there is also a difference in that real-time programmers are reluctant to use dynamically allocated memory. We would argue, in this case, that the dynamic allocation occurs *only* during the initialisation code, so that the memory allocation happens only in the initialisation phase. In any case, the languages being used do not support garbage collection, a feature that should definitely be disabled for real-time applications.

2. Overview

In the remainder of this report we take a bottom-up approach. First, we look at the anti-aliasing issue. Then we consider how a z-transform description can efficiently be turned into a fast implementation. Next, we look at how an s-domain transfer function can be turned into a discrete-time z-domain transfer function. Finally, we consider how a filter description in terms of bandwidth can be turned into a transfer function.

The emphasis throughout is on the question "How can we automate the calculations, rather than requiring the implementer to rely on either hand calculations or off-line calculations"? The goal is to ensure that the crucial design calculations are done during the initialisation phase of the software, rather than being off-line calculations with what that implies in terms of manual copying errors.

3. The Nyquist sampling theorem

A filter implemented in software necessarily operates in discrete time. A hardware filter usually operates in continuous time. In a typical real-time application, the sampling rate is so high that there is no practical difference between the two. That is true in every situation but one: when we need an anti-aliasing filter.

The Nyquist sampling theorem [Nyq] says that sampled data is a good representation of the incoming continuous signal only if the incoming signal has no frequency components above a certain critical frequency. That critical frequency is where the sampling rate is such that we are getting two samples per cycle. That is, if the sampling interval is T, then the critical frequency is

$$f_{Nyquist} = \frac{1}{2T}$$

Equivalently, if the sample rate is f_s samples/second, then

$$f_{Nyquist} = \frac{1}{2} f_s$$

That means that we have to pass the incoming signal through an anti-aliasing filter that rejects all signals of frequency $f_{Nyquist}$ and above. Because no filter has a perfectly sharp cutoff, the bandwidth of the anti-aliasing filter must be below $f_{Nyquist}$. How far below depends on the filter order and on how we define "negligible" for the output of the filter. The calculation of the required bandwidth is a matter of simple arithmetic. This calculation must, however, be done. It is tempting to make a guess and say that it should be good enough to set the bandwidth at, say $0.8 f_{Nyquist}$. When you check the arithmetic, you will find that the difference between 0.8 and 1.0 is very small on a logarithmic scale; and it is important to use the logarithmic scale because of the way filter gain varies with frequency.

4. Analogue vs digital filters

It is *fundamentally impossible* to implement an anti-aliasing filter in software, except in the unrealistic case where we can guarantee that there is no high-frequency noise in the data. The reason for this is that the aliasing that is the subject of the Nyquist criterion happens at the A/D converter. By the time the software sees the samples, the data stream is already corrupt. The only solution to this is to put an analogue anti-aliasing filter ahead of the A/D converter.

What *is* possible is to partition the job between the hardware filter and a software filter, especially if oversampling is used. The order of a hardware filter is constrained by component tolerances, to the point where it is expensive to go beyond fourth order. A software filter, being limited only by processing time and the effect of round-off errors, can achieve a sharper cutoff.

Consider the case where the application requires a sampling rate of 10 kHz, but where the A/D converter is capable of handling 50,000 samples/second. At the 50 kHz rate the Nyquist frequency is 25 kHz, so we can design a hardware filter with a cutoff somewhere between 10 kHz and 25 kHz. The software then can assume that the input has no components above 25 kHz, or a little less. We can now design a software low-pass filter, working at 50 kHz, to remove frequencies between 10 kHz and 25 kHz. The output of that filter can be downsampled [Dec], by taking only every fifth sample, to produce a 10 kHz data stream that has been through an anti-aliasing procedure. At this point, if necessary, we can use the filter that the application really wants, with confidence that the aliasing problem has been solved.

Downsampling has another benefit. By, in effect, averaging every block of 5 samples, we get a better precision than the A/D converter can provide. This becomes really significant when we can use much larger downsampling factors.

5. Practical implementation of a z-domain transfer function

Consider the z-domain transfer function, of degree N,

$$G(z) = \frac{n_0 z^N + n_1 z^{N-1} + n_2 z^{N-2} + \cdots + n_N}{d_0 z^N + d_1 z^{N-1} + d_2 z^{N-2} + \cdots + d_N}$$

or equivalently

$$G(z) = \frac{n_0 + n_1 z^{-1} + n_2 z^{-2} + \cdots + n_N z^{-N}}{d_0 + d_1 z^{-1} + d_2 z^{-2} + \cdots + d_N z^{-N}}$$

It is essential that $d_0 \neq 0$, because otherwise we would have a non-causal transfer function, where the output depended on future inputs. This lets us scale the coefficients by dividing the numerator and denominator by d_0. From now on, then, let us assume that $d_0 = 1$.

If this transfer function represents a filter with input u and output y, and if $u(k)$ and $y(k)$ are the input and output at time k, then

$$\begin{aligned} d_0 y(k) + d_1 y(k-1) + d_2 y(k-2) + \cdots + d_N y(k-N) \\ = n_0 u(k) + n_1 u(k-1) + n_2 z^{-2} u(k-2) + \cdots + n_N u(k-N) \end{aligned}$$

or more compactly

$$y(k) = n_0 u(k) + \sum_{j=1}^{N} n_j u(k-j) - \sum_{j=1}^{N} d_j y(k-j)$$

This can be implemented with the aid of two arrays holding the last N inputs and outputs. We can, however, get a simpler implementation by introducing the concept of *state*.

Consider the state equations

$$x(k+1) = Ax(k) + Bu(k)$$

$$y(k) = Cx(k) + Du(k)$$

where A is an $N \times N$ matrix, B is an $N \times 1$ matrix, C is a $1 \times N$ matrix, D is a 1×1 matrix, and x is an N-vector. (This is the appropriate formulation for a system with a single input u and a single output y). It is a well-known result of system theory that the transfer function from u to y is

$$G(z) = C(zI - A)^{-1}B + D$$

Given the transfer function, the choice of state-space representation is not unique. For our purposes, a convenient choice turns out to be

$$A = \begin{bmatrix} -d_1 & -d_2 & \dots & -d_{N-1} & -d_N \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

$$C = [n_1 - n_0 d_1 \quad n_2 - n_0 d_2 \quad \dots \quad n_N - n_0 d_N] \qquad D = [n_0]$$

The simple structure of A makes it not too difficult to confirm this claim. For those who want to check the state-space literature, the particular structure we have chosen is similar to, but not identical to, a form known as controllable canonical form. That, too, has as a major motivation the simple form of the A matrix.

Naturally, we would not want the overhead of doing full matrix calculations at the (often high) sampling rate. Luckily, these calculations simplify down to

$$x_1(k+1) = \sum_{j=1}^{N} a_j x_j(k) + u(k)$$

$$y(k) = \sum_{j=1}^{N} c_j x_j(k) + n_0 u(k)$$

where the coefficients $a_j = -d_j$ and $c_j = n_j - n_0 d_j$ can be precomputed. To this we must add

$$x_j(k+1) = x_j(k) \text{ for } 2 < j \leq N$$

The key to making this efficient is to store the state $x_1 .. x_N$ in a circular buffer; that is, in an array whose subscript wraps around. We need to keep track of the location of x_1 in the array, because it changes at every time step. At each time, we step through the circular buffer performing the above calculations of $x_1(k+1)$ and $y(k)$. This leaves us at the place in the circular buffer where x_N currently is. We store the new value of x_1 at that position, effectively moving all states by one position. (But without doing any physical movement.) That automatically gives $x_2 .. x_N$ their new values, without any need to touch them.

This approach requires the same number of multiplications and additions as the obvious method of using two arrays. That was inevitable, because there are at least $2N$ coefficients regardless of how we formulate the problem. Using a circular buffer to hold the state does, however, speed the calculation by simplifying the bookkeeping work. It also lets a sophisticated compiler arrange the code to keep as many intermediate results as possible inside the hardware floating point registers, which improves both accuracy and speed.

Ideally, we should keep the entire state in a form that retains the full hardware precision. (80 bits, for the processor used for these tests.) We are not aware of a compiler that allows that option. (Non-optimising compilers store intermediate results in 64-bit variables.) We can reasonably assume, however, that optimising compilers keep intermediate results in internal registers for as long as possible.

6. Conversion from continuous time to discrete time

Any software implementation of a filter necessarily operates in discrete time. In most cases, though, the specification of the required filter will be in continuous-time terms, probably in the form of a Laplace transform transfer function. The process of converting a continuous-time transfer function $G(s)$ to a discrete-time $H(z)$ is tedious if done by hand, so we would prefer to automate the process.

An accurate transformation would require us to use the mapping

$$z = e^{sT}$$

where T is the sampling interval. Unfortunately this would give a discrete-time transfer function that is not a ratio of polynomials, which would be unworkable. In practice, we have to use the bilinear approximation

$$z = \frac{1 + sT/2}{1 - sT/2}$$

To justify this, consider the Taylor series expansions

$$e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \cdots$$

$$\frac{1 + x/2}{1 - x/2} = 1 + x + \frac{1}{2}x^2 + \frac{1}{4}x^3 + \cdots$$

so the approximation is good if x is not too large. The Nyquist sampling theorem says that we must be working with frequencies $< {}^{1}/_{2T}$, so our filter only has to be accurate in the complex frequency range $|sT| < \frac{1}{2}$. At the limit of that range the cubic term in the Taylor series is not totally negligible, but it is reasonably small.

The inverse of the bilinear mapping is

$$s = \frac{2}{T}\frac{z - 1}{z + 1}$$

Given an s-domain transfer function

$$G(s) = \frac{\sum_{j=0}^{N} p_j s^j}{\sum_{j=0}^{N} q_j s^j}$$

the bilinear transform maps this to

$$G_d(z) = \frac{\sum_{j=0}^{N} p_j \left(c\frac{z - 1}{z + 1}\right)^j}{\sum_{j=0}^{N} q_j \left(c\frac{z - 1}{z + 1}\right)^j}$$

where for convenience we have set $c = \frac{2}{T}$. This can be rewritten as

$$G_d(z) = \frac{\sum_{j=0}^{N} c^j p_j (z - 1)^j (z + 1)^{N-j}}{\sum_{j=0}^{N} c^j q_j (z - 1)^j (z + 1)^{N-j}}$$

We can expand some of the terms as

$$(z-1)^j = \sum_{m=0}^{j} (-1)^m B_{jm} z^{j-m}$$

$$(z+1)^{N-j} = \sum_{n=0}^{N-j} B_{N-j,n} z^{N-j-n}$$

where B_{ij} is the binomial coefficient

$$B_{ij} = \prod_{k=1}^{j} \frac{i+1-k}{k} = \frac{i!}{j!\,(i-j)!}$$

It is not hard to show that the above product is

$$(z-1)^j (z+1)^{N-j} = \sum_{n=0}^{N} K_{jn} z^n$$

where

$$K_{jn} = \sum_{k=0}^{\min(n,j)} (-1)^m B_{jk} B_{N-j,n-k}$$

From there it is obvious how to calculate the numerator and denominator coefficients for $G_d(z)$. This is an excellent example of a calculation that is tedious and error-prone if done by hand, but easy to do in software.

The formulae given for B_{ij} are not the most efficient for practical use. It is better to store a Pascal's triangle in the array *B* before starting the calculation.

It is worth noting the following point. For a stable minimum-phase transfer function in the s domain, the corresponding transfer function in the z domain is going to have a denominator with alternating sign, which implies complications in the final results (we are subtracting terms which are almost equal). We shall deal with this problem in a later section.

7. Frequency warping

As shown in the last section, the bilinear transformation used to map from continuous time to discrete time is only an approximation to an ideal transformation. It turns out [Bilin] that there is a slight nonlinear distortion in the frequency scale, given by

$$\omega_a = \frac{2}{T} \tan\left(\omega \frac{T}{2}\right)$$

or, in terms of frequency in Hz,

$$f_a = \frac{1}{\pi T} \tan(\pi f T)$$

such that the discrete-time filter behaves at frequency ω the same way that the original s-domain transfer function behaves at frequency ω_a. This means that if, for example, you want the filter to have a pole at ω, you have to put the pole at ω_a in your continuous-time

frequency specification. This is called pre-warping: we warp the original frequency specification so that, after the bilinear transform, the pole is warped back to where you originally wanted it.

It is not practical to pre-warp the entire frequency range of interest, because that would lead to a nonlinear filter even if the calculations were feasible. In practice, we pre-warp the corner points: the poles and zeros of the original transformation.

Consider an example where the sampling frequency is 20 kHz (so that the Nyquist frequency is 10 kHz), and we want a low-pass filter with a bandwidth of 7 kHz. We would not want to go much closer to the Nyquist frequency than that. We have $T = 5 \times 10^{-5}$ seconds, so

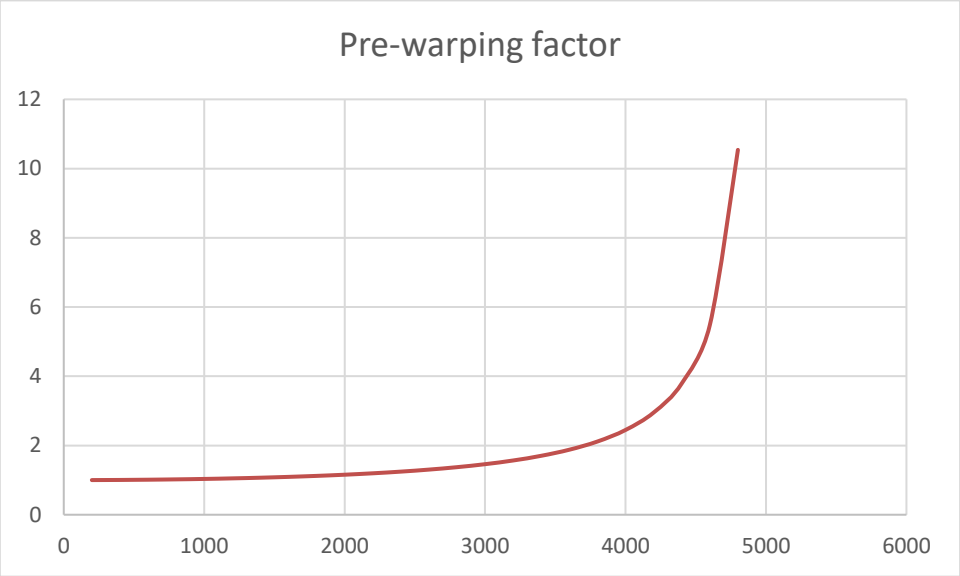
$$f_a = \frac{2}{5 \times 10^{-5}} \tan(\pi \times 7 \times 10^3 \times 5 \times 10^{-5})$$

$$f_a = 7.85 \text{ kHz}$$

This is more than a 10% change, so should not be ignored. If we neglect the pre-warping, the final bandwidth turns out to be only 5.3 kHz, a long way below the specification. For frequencies that are a long way below the Nyquist frequency, pre-warping is less critical.

Note that the warping must be done while we are still at the high-level description using poles and zeroes. Once the transfer function is expressed as the ratio of two polynomials, prewarping is no longer feasible, unless of course we are willing to factor the polynomials.

The following graph shows f_a/f, for the case of a 10 kHz sampling rate. (So that the Nyquist frequency is 5 kHz.) It can be seen that the ratio remains close to 1 for small frequencies, but rises rapidly as one approaches the Nyquist frequency.



8. Butterworth filters

Often enough, a filter specification is not in the form of a transfer function, but is instead expressed in terms of passbands and/or stopbands. In this section we look at how to implement low-pass, high-pass, bandpass, and bandstop Butterworth filters. Doing the same job for other filter types is left as an exercise for the reader.

The advantage of a Butterworth filter is that it is maximally flat in the passband. Other filter types can give a sharper cutoff at the edge of the passband, but at the cost of some ripple near the edge of the passband.

8.1 Normalised Butterworth filter

A normalised Butterworth filter is a low-pass filter with a bandwidth of 1 rad/s. This is the starting point of any Butterworth filter design. Once we have the normalised filter, we can apply scaling to produce the desired bandwidth.

The normalised Butterworth filter of order N has frequency response

$$|G(j\omega|^2 = \frac{1}{1+\omega^{2N}}$$

To achieve this, we place poles in an equally-spaced pattern on the unit circle. (But, of course, we retain only the ones in the left half-plane.) The result is a transfer function

$$G(s) = \frac{1}{p(s)}$$

where, for a filter of order N, $p(s)$ turns out to be [But]

$$p(s) = \prod_{k=1}^{N/2}\left[s^2 - 2s\cos\left(\frac{2k+N-1}{2N}\pi\right) + 1\right] \qquad \text{for } N \text{ even}$$

$$p(s) = (s+1)\prod_{k=1}^{(N-1)/2}\left[s^2 - 2s\cos\left(\frac{2k+N-1}{2N}\pi\right) + 1\right] \qquad \text{for } N \text{ odd}$$

This can be confirmed by simple trigonometry, given the requirement for equally spaced poles. Apart from the extra factor for odd N, this is just a product of quadratics, so what we need in software is the elementary job of multiplying polynomials.

8.1 Low-pass filter

Given a normalised filter

$$G_{norm}(s) = \frac{1}{s^N + a_{N-1}s^{N-1} + a_{N-2}s^{N-2} + \cdots}$$

with bandwidth 1 rad/s, we can turn it into a low-pass filter with bandwidth B rad/s by changing s to s/B. This gives

$$G(s) = \frac{1}{s^N/B^N + a_{N-1}s^{N-1}/B^{N-1} + a_{N-2}s^{N-2}/B^{N-2} + \cdots}$$

whose implementation in software is obvious.

8.2 High-pass filter

For a high-pass filter with cutoff B rad/s, we instead change s to B/s. This gives

$$G(s) = \frac{s^N}{B^N + a_{N-1}B^{N-1}s + a_{N-2}B^{N-2}s^2 + \cdots}$$

Note that the powers of s in the denominator are in the opposite order from the expression for the low-pass case.

8.3 Bandpass filter

A bandpass filter with two frequency limits (after pre-warping, of course) ω_1 and ω_2, can be thought of as a combination of low-pass and high-pass filters. That suggests a mapping

$$s \to Cs + \frac{D}{s}$$

where C and D are constants to be determined. In terms of real frequency, this is

$$j\omega \to j\omega C + \frac{D}{j\omega}$$

or

$$\omega \to \omega C - \frac{D}{\omega}$$

Since the normalised filter has a corner frequency of 1 rad/s, it is tempting to say that the right side must evaluate to 1 for both $\omega = \omega_1$ and $\omega = \omega_2$. That turns out to be a mistake. It does map those two frequencies as desired, but it does the wrong thing in the rest of the complex plane, giving an unstable filter.

Instead, we need to remember that the normalised Butterworth filter also has a response for negative frequencies. We can therefore map ω_1 to -1 and ω_2 to +1. Now we have

$$-1 = \omega_1 C - \frac{D}{\omega_1}$$

$$+1 = \omega_2 C - \frac{D}{\omega_2}$$

This is easily solved to give

$$C = \frac{1}{\omega_2 - \omega_1} \qquad\qquad D = \frac{\omega_1 \omega_2}{\omega_2 - \omega_1}$$

so the map is now

$$s \to \frac{1}{W}\left(s + \frac{A}{s}\right)$$

where $A = \omega_1 \omega_2$ and $W = \omega_2 - \omega_1$.

Let N be the order of the desired bandpass filter. Because a bandpass filter has twice as many poles as a similar low-pass filter, it is reasonable to insist that N be even, and our starting point should be a normalised Butterworth filter of order $M = N/2$.

$$G_{norm}(s) = \frac{1}{\sum_{j=0}^{M} a_j s^j}$$

So the final transfer function is

$$G(s) = \frac{1}{\sum_{j=0}^{M} a_j \frac{1}{W^j}\left(s + \frac{A}{s}\right)^j}$$

which simplifies down to

$$G(s) = \frac{W^M s^M}{\sum_{j=0}^{M} a_j W^{M-j} \sum_{k=0}^{j} A^{j-k} B_{jk} s^{2k+M-j}}$$

where B_{jk} is the binomial coefficient as in earlier sections. The denominator is not quite in a neat polynomial form, but that is not a problem. Each time through the inner loop, the $(2k + M - j)$ tells us which denominator coefficient to increment.

Another way to design a bandpass filter is as a cascade of a low-pass and a high-pass filter, but experimentation shows that this is not an attractive option. When the passband is wide, the performance of a filter designed this way is almost identical to the performance with the approach just discussed. When the passband is narrow, the performance is noticeably worse. The reason for this is the same as for the case of a bandstop filter designed that way, as shown in the next subsection.

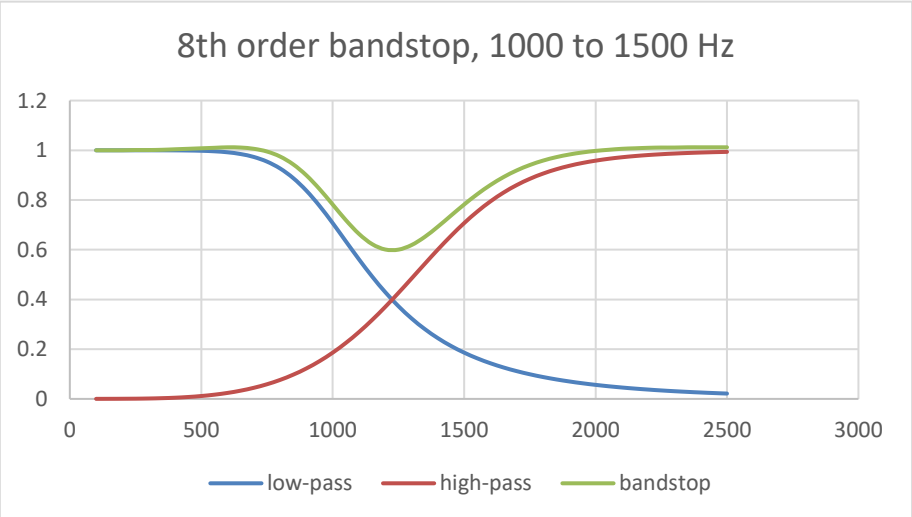
8.4 Bandstop filter

Designing a bandstop filter turns out to be a little more difficult than the cases already considered. In this section we look at three approaches.

One obvious way to design a bandstop filter is to calculate the transfer function $G_{bp}(s)$ for the corresponding bandpass filter, and then use $G_{bs}(s) = 1 - G_{bp}(s)$. This turns out to give a good result for a second-order filter. For any higher order, unfortunately, the results are disappointing: an implementation gives a transfer function that is rather different from the ideal.

The reason is not hard to trace. It turns out that, for any order greater than 2, the numerator of $G_{bs}(s)$ is almost (but not quite) identical with the denominator. That means that all of the subsequent calculations rely critically on small differences between large numbers, so that rounding error ruins the results.

A more traditional approach is to view the bandstop frequency response as the sum of a low-pass response and a high-pass response. Although this sounds like a sensible approach, the attenuation in the stop band is often disappointing. The following graph shows the frequency response for an 8th order filter, made of the sum of a 4th order low-pass filter and a 4th order high-pass filter. For our present purpose it is convenient to display the graph on linear rather than logarithmic scales. The bandstop result is not precisely equal to the sum of the two components, again because of rounding error, but the general trend is clear.



It can be seen that the minimum gain in the stopband is about 0.6. (The graphs seem to indicate that the minimum gain of the sum should really be 0.8, but one also has to take phase difference into account.) The reason is obvious. The two critical frequencies are close enough together that the two component filters both have non-negligible gain in the stopband This can be remedied by moving the two critical frequencies further apart, or by increasing the filter order. Still, it is a little disappointing that even 8th order is not good enough for this application.

There is one further possible approach, one that does give much reduced gain in the stopband. At the beginning of this subsection it was mentioned that a 2nd order bandstop filter derived from a bandpass filter works well. By looking at the bandpass calculations for 2nd order, it is easily found that the transfer function for this filter is

$$G_2(s) = \frac{s^2 + A}{s^2 + Ws + A}$$

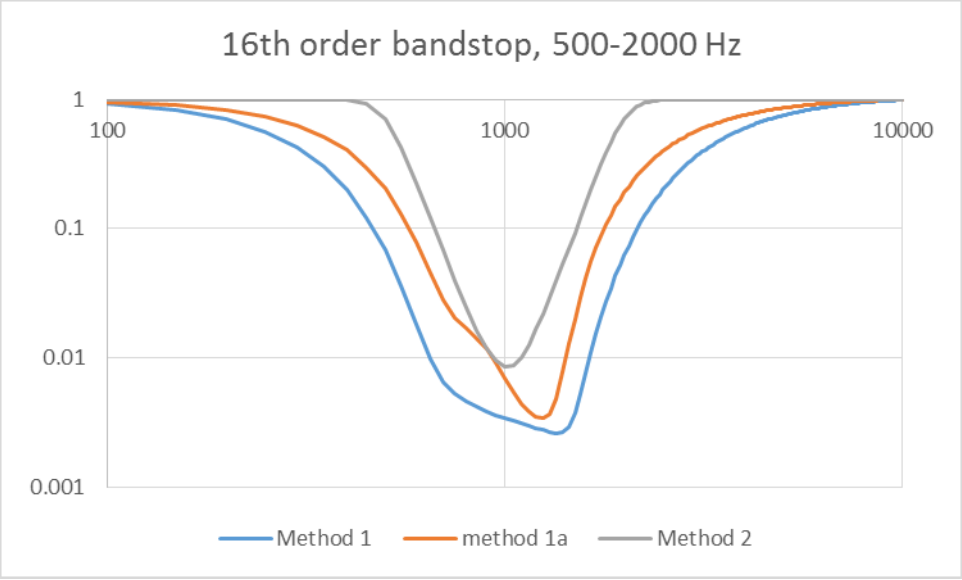
where $A = \omega_1\omega_2$ and $W = \omega_2 - \omega_1$. We can cascade several of these filters to get a higher-order filter

$$G_N(s) = \left(\frac{s^2 + A}{s^2 + Ws + A}\right)^{N/2}$$

where the filter order N is required to be even.

One disadvantage of cascading is that it moves some attenuation into the passband. Normally the gain at the two corner frequencies is approximately 0.71, but when we cascade the sections we get a corner gain of $0.71^{N/2}$. That means that the corners are more rounded than intended. We can reduce this effect a little by reducing the W factor a little at each iteration, but that has to be done cautiously because cumulative errors can destabilise the filter.

The following graph shows the results for three design methods. Method 1 is the method where we cascade the second order filters. Method 1a is a slight modification where we multiply W by 0.9 at each stage (thus giving those stages a smaller stopband but with the same centre frequency). Method 2 uses a low-pass and a high-pass filter in parallel.



It can be seen that method 2 gives the best "corner" behaviour, but less stopband attenuation. Methods 1 and 1a show some departure from symmetry inside the stopband. That is because

imperfections due to rounding errors – see next section – show up under conditions where the filter output is small. Because all three methods have their advantages, our software implements all three.

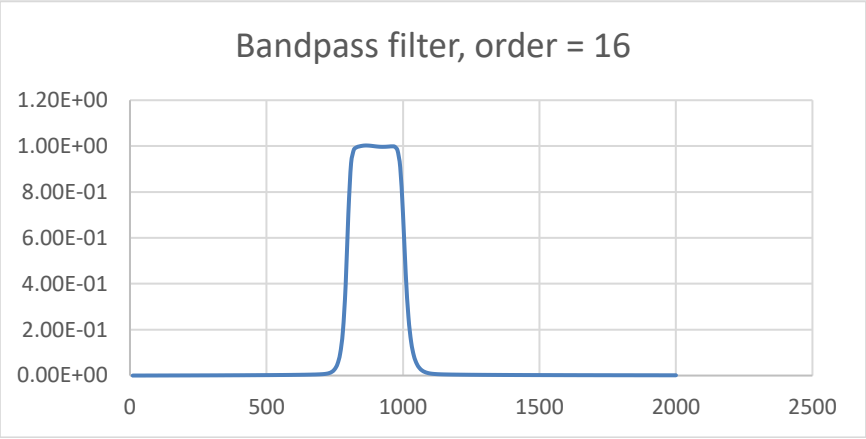
9. Accuracy considerations

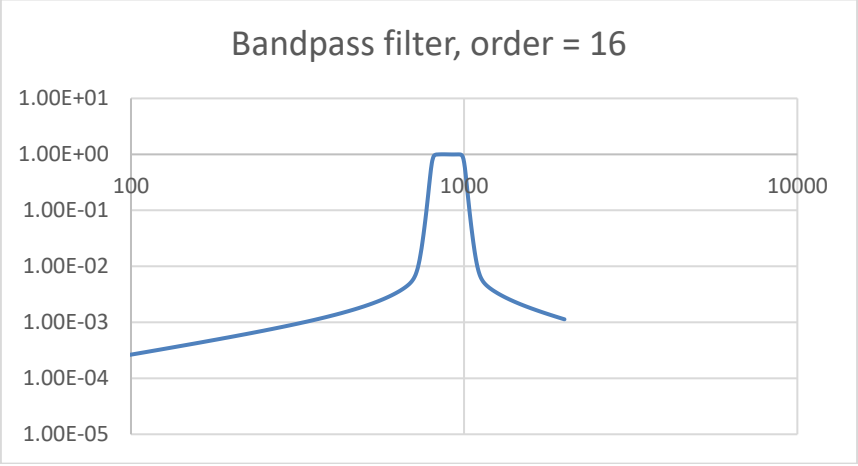
It is generally understood that high-order filters are sensitive to the filter coefficients, to the point where a practical implementation can have a frequency response very different from the intended result. Because of this, it is a common practice to implement a high-order analogue filter as a cascade of second-order filters.

This "cascade" approach is less attractive for digital filters, because of the extra computational load, and propagation of rounding error. Furthermore, it appears to be unnecessary; in the digital case, the only "component tolerance" problem is the rounding of finite-precision numbers. To reduce the number of calculations, and therefore the effect of rounding error, it is better to work directly with the discrete-time transfer function.

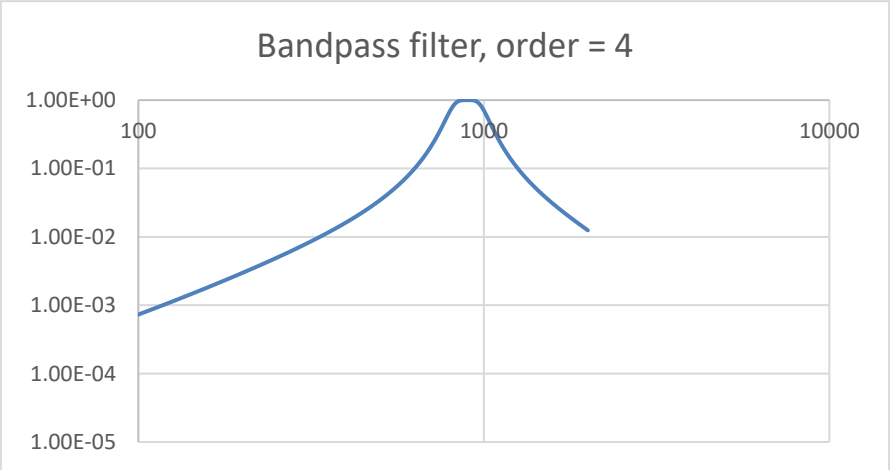
The following two graphs show the frequency response of a 16th order bandpass filter, with passband 80 to 100 Hz. On a linear scale, the result looks excellent. On a log-log scale, we see a departure from ideal behaviour when the gain drops lower than about 0.006. In that low-gain region, the filter looks like a lower-order filter.

This is not just a feature of this example. For all examples tested, the frequency response is close to the theoretical ideal in and near the passband, but becomes non-ideal when the gain is low.





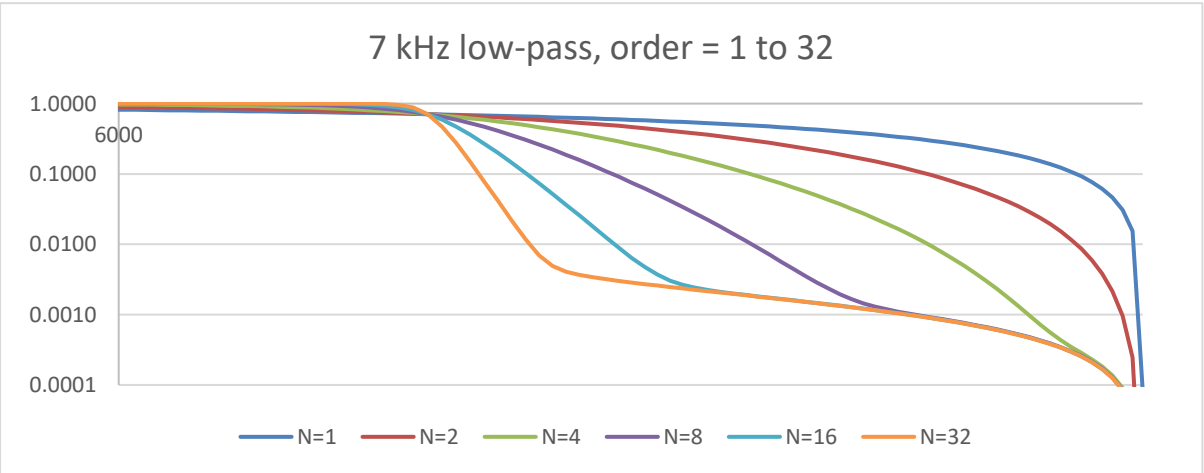
For comparison, here is the result for a lower-order version of this filter. The departure from ideal in the stopband is still there, but it is less obvious because a lower-order filter has less of a sharp edge.



The reason for this behaviour is obvious enough in hindsight. At any frequency where the gain is supposed to be low, we have an input of reasonable amplitude but an output that is small. Looking at the filter state equations, this small output is typically caused by the subtraction of large numbers that are nearly equal. That means that rounding error, caused by finite arithmetic precision, can be comparable with the small numbers we are trying to compute. The result is an output that is small, but not quite as small as would be expected from the theory. What we are seeing in the stop band is the cumulative effect of rounding error.

It is instructive to look at the details of how we convert from an *s*-domain transfer function to a *z*-domain one. A stable minimum-phase transfer function – that is, one that has all its poles and zeroes in the left half-plane – has positive (or, at worst, zero) coefficients in both the numerator and denominator polynomials. When we convert to discrete time, the resulting *z* transform usually has coefficients of alternating sign. This is the main reason we get large numbers that almost, but not quite, cancel each other.

Let us consider the case of a low-pass filter with a bandwidth of 7 kHz and a sampling frequency of 20kHz. (Which places the bandwidth close to the Nyquist frequency of 10 kHz.) We get the following frequency response.



Here we see two interesting phenomena:

(a) When the frequency approaches the Nyquist frequency, the gain drops rapidly towards zero. This is hardly surprising, given that we do not expect our filter to work above the Nyquist frequency because of aliasing problems. We do not expect "ideal" responses near the Nyquist frequency.
(b) The higher-order filters, of order $N \gg 1$, behave as the theory predicts until the gain drops below about 0.005, after which they appear to behave as lower-order filters.

In practice, this rarely matters. The departure from ideal behaviour occurs when the outputs are very small. In most applications, those small numbers are not going to have any adverse effect. Nevertheless, it is helpful to understand why it is happening.

To remove the distracting effect of what happens near the Nyquist frequency, let us reduce the bandwidth a little while keeping the same sampling rate. The following graph adds a couple of straight lines, to allow us to estimate the slope. It appears that the slope, in the region where we depart from ideal behaviour, is about 26 dB/decade. That is what we would get from a filter of order 1.3, if such a thing could exist. Since the "order" is not an integer, it seems that we cannot explain the phenomenon in terms of order reduction or spurious zeroes.

The reason appears to be a consequence of finite-precision arithmetic. In the s domain, the denominator of the transfer function of this filter has coefficients with the property that the ratio between the biggest and the smallest is more than 10^{60}. In the z domain the range is more reasonable – it is affected not so much by the absolute frequency as by the ratio of the bandwidth to the Nyquist frequency – but something more subtle is happening.

A stable and minimum phase transfer function in the s domain has nonnegative coefficients. When this is transformed into the z domain, the denominator has coefficients with alternating sign. This should not matter until we consider the values. It is inconvenient to illustrate this for a high-order filter, so let us instead look at the 6th order case. The transfer function is

$$G(z) = 10^{-5}\,\frac{0.86 + 5.15z^{-1} + 12.9z^{-2} + 17.2z^{-3} + 12.9z^{-4} + 5.15z^{-5} + 0.86z^{-6}}{1.00 - 4.79z^{-1} + 9.65z^{-2} - 10.47z^{-3} + 6.44z^{-4} - 2.13z^{-5} + 0.30z^{-6}}$$

To this precision, the sum of the denominator coefficients is zero. A more precise calculation shows that the sum is 5.49×10^{-4}. If it really were zero, the DC gain would be infinite, and the filter would be unstable. When we turn this transfer function into a filter implementation,

the filter is adding and subtracting numbers that almost, but not quite, cancel out. The accuracy of the filter depends heavily on that "not quite". It should not be surprising, then, that when we get up to frequencies where the filter state is numerically small, rounding error might well make a significant contribution to the result.

We do not know whether this is a full explanation of the behaviour in the stop band. In particular, it is not clear from this why the filter should act almost like a first order filter once it gets well into the stop band. It is clear, though, that rounding error is an important effect.

Let us now move on to the topic of stability. Not surprisingly, this also depends on arithmetic accuracy. For a given filter specification (passband limit(s), sampling rate), we can try the effect of various filter orders. As the order goes beyond the usable range, a ripple develops in the passband, and soon becomes unacceptable. Shortly after that it goes unstable. The 1 kHz filter just discussed works well up to order N=17 using the software implementation that accompanies this report. It goes unstable at N=21 for the C code version, or N=22 for the Modula-2 code version. The difference is probably due to code optimisation, because more efficient code often means a smaller accumulation of rounding error. It is possible that the quality of the library code accompanying the compilers plays a part.

In practice, of course, one would do tests to establish the point at which ripple appears in the frequency response, and back off a little from that point as a safety margin.

The point of instability seems to depend on how close to the Nyquist limit the critical frequencies are. The 1 kHz filter just discussed goes unstable at N=22, but the 7 kHz filter has good performance at N=32. (They have the same sampling rate, 20,000 samples/s.) If we keep the same sampling rate but drop the bandwidth to 50 Hz, then we already get unacceptable ripple at N=8. (Although N=7 works.) The reason is easy to see: with N=8, the sum of denominator coefficients is only 1.9×10^{-15}. That sum varies a lot with N.

For the same 50 Hz filter, but with a sampling rate of only 800 samples/s, we can go all the way up to N=20 with satisfactory results. In this case the sum of denominator coefficients is 7.9×10^{-10}.

It appears, then, that non-ideal behaviour and instability both have the same cause: denominator coefficients whose sum is too close to zero. This, in turn, is affected by the choice of sampling rate. If the sampling rate is too high, the accuracy suffers, which limits how high a filter order we can choose. If it is too low, our operating frequencies can go too close to the Nyquist frequency. The system designer must steer a path between these extremes.

The filter order and the sampling rate are, of course, also constrained by the available processor time. This depends not only on which processor is chosen, but also on what other software is running. (Typically the filtering is only a small part of the processing.) Good analysis tools can work out how to allocate the time/sample, but ultimately we have to rely on trial and error to work out how much time is available.

10. Conclusions

The only really novel idea in this report is the suggestion that a large part of the filter design can be done during the initialisation phase of a real-time system. Traditionally all of the filter design work is done off-line, with the parameters then copied into the real-time code. The

copying is error-prone, and, we assert, not necessary. The processor time needed to do the calculations is small enough to fit easily into a few milliseconds of the initialisation code.

All of the algorithms here can be derived from information freely available on the web. Unfortunately, most web sources give a general overview without the details, leaving the reader with a great deal of algebra to do. It is hoped that this report will fill those gaps.

An unresolved problem, where further research is desirable, is why the gain of higher-order filters start looking like the response of first-order filters once we are well into the stopband. An answer to this question would require a good model of how rounding errors affect a calculation.

The software for the algorithms described here are available for download [Moy17].

.

References

[Bilin] Bilinear transform, https://en.wikipedia.org/wiki/Bilinear_transform

[But] Butterworth filter, https://en.wikipedia.org/wiki/Butterworth_filter

[Dec] Decimation (signal processing), https://en.wikipedia.org/wiki/Decimation_(signal_processing).

[Moy17] Filter software package, ftp://ftp.pmoylan.org/software/filters.zip

[Nyq] Nyquist/Shannon Sampling Theorem, https://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem.