Near Field filters for Higher Order Ambisonics

Fons ADRIAENSEN
fons.adriaensen@skynet.be

Abstract

A digital implementation of the near-field filters used in Higher Order Ambisonics may introduce some problems with numerical precision and stability. This is due to the fact that these filters mainly operate at the very low end of the audio range. This technical note documents a simple practical solution to this problem, and introduces a set of C++ classes implementing the filters up to fourth order.

1 Introduction

The importance of compensating for the near-field (NF) effect of the loudspeakers used in an Ambisonics reproduction system was already pointed out in the early days of this technology by Michael Gerzon.

While in a first order system and at practical source distances only the very low end of the audio range is affected, this is no longer the case for HOA. At higher orders the effect starts at higher frequencies, and for order m it is inversely proportional to the m-th power of both frequency and source distance.

Filters (and their inverses) emulating the higher order near-field effects are required in three situations:

- Compensation of NF effects in a HOA reproduction system. At a practical speaker distance of a e.g. two meters, the NF effect in the second and higher order components can not be ignored.
- To create virtual sound sources close to the listener. This is mainly important in the context of electro-acoustical music encoded into HOA, and for virtual reality and telepresence systems.
- In the synthesis of HOA from signals captured by a cluster of omnidirectional or first-order microphones. In this case, encoding for a finite (and relatively small) reproduction rig radius limits the excessive gains required at low frequencies to manageable proportions.

Near field effects in HOA were analysed in detail by Jérôme Daniel in his paper (Daniel, 2003) presented at the 23rd AES Conference in Copenhagen. This paper also discusses the derivation of the digital filters corresponding to these NF effects. However, a 'textbook' form of these filters will not work correctly unless it uses very high precision arithmetic.

2 Digital realisation of NF filters

The NF effect for a signal of order m corresponds exactly to a simple analog filter which is the inverse of an m-th order highpass. This means that the forward filters amount to m-fold integration at low frequencies, and have infinite gain at DC. Any offset voltage (in an analog realisation) or biased round-off error (for a digital one), no matter how small, will result in an uncontrolled DC component. So in practice the forward filter must always be combined with some feedback mechanism that turns it into a finite-gain shelf filter, or with an inverse NF filter corresponding to a larger source distance. The latter approach is taken in the C++ classes documented in the final section. As we will see this can be done with essentially zero overhead.

We start with equation (5) of Jérôme Daniel's paper[1], which can be rewritten as

$$F_m(s) = \sum_{i=0}^{m} a_{m,i} X^i \tag{1}$$

with

$$a_{m,i} = \frac{(m+i)!}{(m-i)!\ i!\ 2^i} \tag{2}$$

$$X = \frac{c}{sr} \tag{3}$$

[1] There is a small typographical error in the equation as printed in that paper: the factor 2^i is missing (confirmed by the author).

$$s = j\omega = j2\pi f \quad (4)$$

with c the speed of sound, and r the distance to the source. The $a_{m,i}$ for orders 1 to 4 are:

m	$a_{m,i}$
1	1, 1
2	1, 3, 3
3	1, 6, 15, 15
4	1, 10, 45, 105, 105

For a practical realisation we need to factor the polynomials in s into first and second order sections. By considering them to be polynomials in X we need to do this only once:

m	1	X	X^2
1	1	1	
2	1	3	3
3	1	3.6778	6.4595
	1	2.3222	
4	1	4.2076	11.4877
	1	5.7924	9.1401

Given these factorisations, we can now for each order m and for a given c and r express (1) as a product of sections of the form

$$H_1(s) = 1 + b_{1,1}s^{-1} \quad (5)$$
$$H_2(s) = 1 + b_{2,1}s^{-1} + b_{2,2}s^{-2} \quad (6)$$

2.1 Using the bilinear transform

The standard way to convert an analog filter into the digital z domain is to use the *bilinear transform* which consists of the substitution

$$s = 2F_s\frac{1 - z^{-1}}{1 + z^{-1}} \quad (7)$$

with F_s being the sample frequency. This transform causes a warping of the frequency scale, but this can be safely ignored in this case. The first and second order sections (5), (6) are transformed into respectively

$$H_1(z^{-1}) = g_1\frac{1 + c_{1,1}z^{-1}}{1 - z^{-1}} \quad (8)$$
$$H_2(z^{-1}) = g_2\frac{1 + c_{2,1}z^{-1} + c_{2,2}z^{-2}}{1 - 2z^{-1} + z^{-2}} \quad (9)$$

with

$$b'_{k,i} = b_{k,i}/(2F_s)^i \quad (10)$$
$$g_1 = 1 + b'_{1,1} \quad (11)$$
$$c_{1,1} = -(1 - b'_{1,1})/g_1 \quad (12)$$
$$g_2 = 1 + b'_{2,1} + b'_{2,2} \quad (13)$$
$$c_{2,1} = -2(1 - b'_{2,2})/g_2 \quad (14)$$
$$c_{2,2} = (1 - b'_{2,1} + b'_{2,2})/g_2 \quad (15)$$

Two things should be noted about these. First, in both H_1 and H_2 the denominator does not depend on $b_{k,i}$. So it will cancel out if we combine forward and inverse operations into one section. Second, the coefficients in the numerator and denominator are almost equal for realistic values of c/r and F_s, and the entire action of these filters is determined by minute differences between them.

For example, in the second order section for $m = 2$, $r = 10$m, $F_s = 48$kHz we have $b'_{2,2} \simeq 3.8e-7$. Given that we have about seven decimal digits of precision in an IEEE floating point number or a 24-bit fixed point value, it is clear that the resulting value of $c_{2,1}$ will not accurately represent the small difference with the corresponding coefficient (-2) in the denominator.

2.2 An alternative form

A first solution that comes to mind is to replace the z^{-1} delay element by

$$\zeta^{-1} = \frac{1 + z^{-1}}{1 - z^{-1}} \quad (16)$$
$$s = \frac{2F_s}{\zeta^{-1}} \quad (17)$$

This would result in an $H_1(\zeta^{-1})$ and $H_2(\zeta^{-1})$ similar to $H_1(s)$ and $H_2(s)$. However, this is not a practical solution. The problem is that (16) contains a zero delay path, and this forces the denominator to be empty. So we can not add a feedback term to limit the DC gain nor combine with an inverse NF filter.

A second approach is to use a delay element of the form

$$\zeta^{-1} = \frac{z^{-1}}{1 - z^{-1}} \quad (18)$$
$$s = \frac{2F_s}{1 + 2\zeta^{-1}} \quad (19)$$

In practical terms this means that instead of shifting a sample into a delay element, it is added to the current value. The first and second order sections (5), (6) are now transformed into respectively

$$H_1(\zeta^{-1}) = g_1(1 + d_{1,1}\zeta^{-1}) \quad (20)$$

$$H_2(\zeta^{-1}) = g_2(1 + d_{2,1}\zeta^{-1} + d_{2,2}\zeta^{-2}) \quad (21)$$

with

$$b'_{k,i} = b_{k,i}/(2F_s)^i \quad (22)$$
$$g_1 = 1 + b'_{1,1} \quad (23)$$
$$d_{1,1} = 2b'_{1,1}/g_1 \quad (24)$$
$$g_2 = 1 + b'_{2,1} + b'_{2,2} \quad (25)$$
$$d_{2,1} = (2b'_{2,1} + 4b'_{2,2})/g_2 \quad (26)$$
$$d_{2,2} = 4b'_{2,2}/g_2 \quad (27)$$

The $d_{k,i}$ are now free of added constants, and can be represented accurately in single precision floating point format. The two sections still have infinite gain at DC, but since (18) includes a delay, we can use feedback terms to add an inverse NF filter.

3 The C++ filter classes

The code documented below is made available under the terms of the GPL license. The complete text of this license can be found in the file named COPYING that comes with the source files.

The filter classes are designed to be used in Linux applications where the standard sample format is single precision IEEE float, and processing is typically done in blocks of N samples. There is however nothing that makes them Linux specific. They only use the very basic C++ feature of a class and nothing else.

There are four classes named NF_filt$k$, where k is 1...4, and represents the order. They combine the forward and inverse filters in one operation and all have the same interface. The first order one is used as an example below.

```
NF_filt1 (void) {}
~NF_filt1 (void) {}
```

Constructor and destructor. Each object contains precomputed coefficients for the forward and inverse filters, and the filter state preserved in between *process()* calls (see below). So you should have one object per channel to be processed.

```
void init (float w1,
           float w2,
           float g = 1.0f);
void init (NF_filt1& F);
void reset (void) { _z1 = 0; }
```

The *init()* method initialises the filter characteristics. The parameters w_1 and w_2 are for the forward and inverse filters respectively. Both are interpreted as

$$w_i = \frac{c}{r_i F_s} \quad (28)$$

with c = speed of sound, r_i = source or speaker distance, and F_s = sample frequency.

If no forward filter is required, w_1 can be set to zero. This should *not* be done for w_2, as the result is a filter with infinite DC gain. In practice the maximum distance for the inverse filter should be something like 10m. Note that a high distance ratio will lead to extreme LF gains (e.g. 80 dB for the 4th order filter and a ratio of 1:10), so some care should be taken with this. If the forward filter is used in 'panner-with-distance' module for example, it may be wise to encode for a reasonable reproduction rig radius, limit the minimum virtual source distance, and high-pass the panner input signal.

The optional parameter g specifies the HF gain of the filter. This can be useful sometimes and it comes for free since it just modifies a gain factor that has to be there anyway.

The second *init()* call is just for convenience and copies the filter coefficients from an other filter of the same order.

The *reset()* call resets the internal state to all zeros. Note that *init()* does not imply *reset()*.

```
void process (int n,
              float *ip,
              float *op,
              int d = 1);
void process1 (int n,
               float *ip,
               float *op,
               int d = 1);
```

Process n samples from input array *ip* and place the result in *op*. The two pointers can be identical for in-place processing. The second form saves some CPU cycles if only the inverse filter is required (i.e. if w_1 was zero). The optional parameter d specifies the distance between two samples of the same channel in case an interleaved representation is used.

4 Some examples

Figures 1 and 2 show some plots obtained by filtering a Dirac pulse and using a 64K FFT on the result.

References

Jérôme Daniel. 2003. Spatial sound encoding including near field effect: Introducing distance coding filters and a viable, new Ambisonic format. 23rd AES conference, Copenhagen, Denmark.

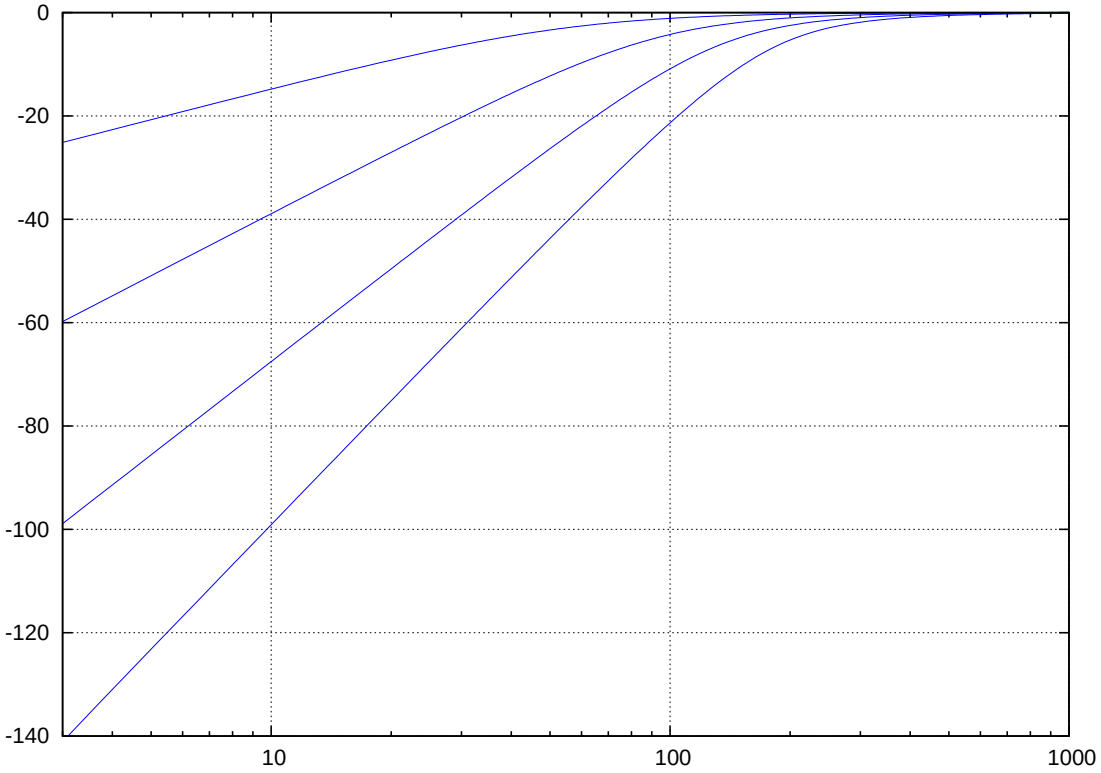

Figure 1: Inverse filters for a source distance of 1m

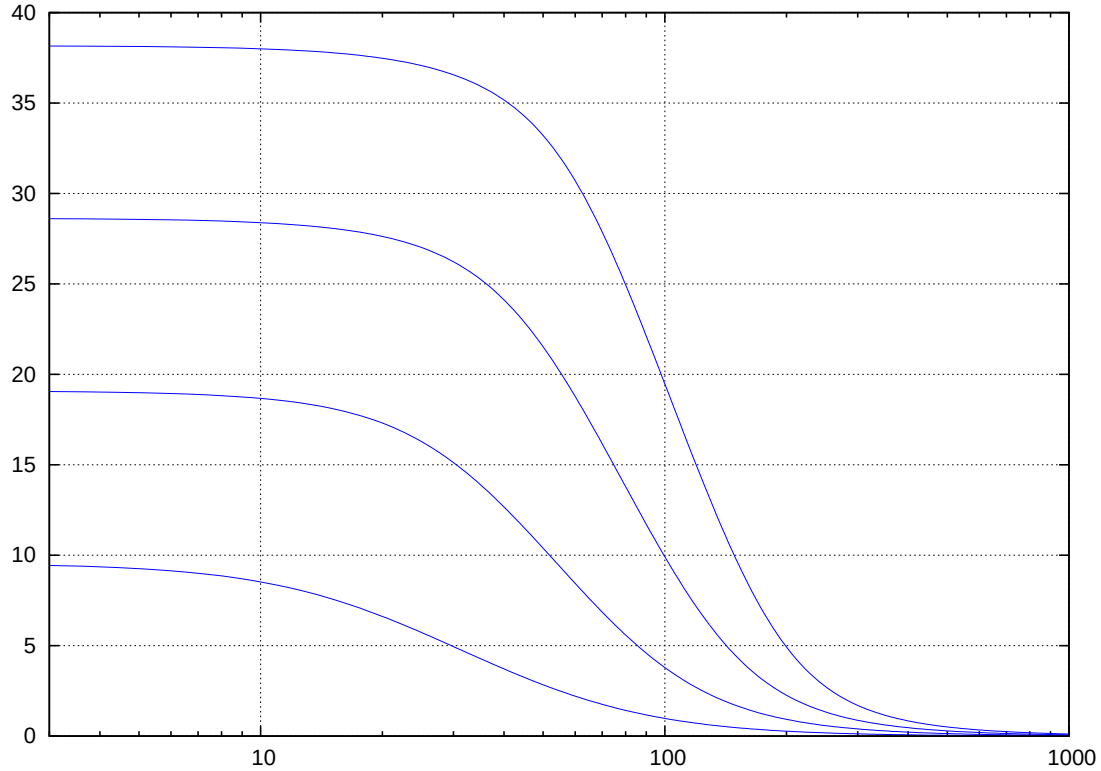

Figure 2: Filters for virtual source at 1m and 3m speaker distance