A Floating-Point to Fixed-Point Conversion Methodology for Audio Algorithms

Mihir Sarkar

Most Digital Signal Processors perform computations on integers, or fixed-point numbers, rather than floating-point numbers. In contrast, Digital Signal Processing algorithms are often designed with real numbers in mind and usually implemented in floating-point. Apart from finite word-length effects that may appear during signal acquisition and intermediate computations, limits on the signal precision and range often compromise the stability of the system.

Audio algorithms are particularly sensitive to fixed-point implementations due to the audible artifacts that the conversion process may introduce. Therefore, it is essential to validate the stability and static characteristics of the system after conversion. Then, the dynamic behavior of the system can be studied by applying suitable test signals.

Starting with a presentation of basic Digital Signal Processing concepts relevant to our discussion, this paper carries on with a floating-point to fixed-point conversion strategy for audio processing algorithms. Finally, a high-pass IIR filter implementation example is presented.

1. INTRODUCTION

An audio signal is an electrical (or optical) representation of the continuous variation in time of a sound pressure wave traveling through a medium such as air. In mathematical terms, we describe an audio signal as a real function of a single real variable, where the x-axis represents time, and the y-axis, the signal amplitude, which can take an arbitrary value at any point in time; in other words, an audio signal is analog.

Most modern audio products, however, process audio signals digitally, which is to say, they process numerical values *representing* an analog audio signal. Fuelled by increasingly complex audio algorithms, Digital Signal Processors (DSPs) have become a key component for audio systems. Moreover, fixed-point DSPs are preferred over their floating-point counterparts in an attempt to control costs and consume less power for portable applications. These fixed-point DSPs do not support floating-point data types and operations at the instruction level; furthermore, emulating a floating-point operation on a fixed-point DSP is out of the question as it results in an intolerable run-time penalty. Since fixed-point numbers require fewer bits than floating-point numbers to achieve the same accuracy in the mantissa [6], designers can reduce the chip size by not including a Floating-Point Unit (FPU), and still maintain a high throughput while running at a limited clock rate. On the flip side, fixed-point processors require a great number of macro manipulations on the data in order to complete the required math operations with minimal loss of precision. However, this additional effort is usually a one-time engineering cost that is quickly amortized on consumer products.

When analyzing a system, it is important to make a distinction between the input signals and the algorithms that are going to process those signals. In audio systems, input signals are typically digitized and coded as Pulse Code Modulation (PCM) data by an Analog-to-Digital Converter (ADC). A *CD-quality* signal, for instance, is sampled at 44.1kHz with 16-bit precision. Thus, audio signals are generally represented as fixed-point numbers in most systems; we shall not discuss here algorithms that handle real numbers (discrete-time signals). On the other hand, audio-processing algorithms can either be implemented on a floating-point or a fixed-point computing system.

The floating-point to fixed-point conversion process involves several steps. First, a static model of the fixed-point algorithm is designed. For example, we select the filter structure that maximizes stability, and quantize the filter coefficients. If the filter loses its properties (cut-off frequency, etc.) during this one-time conversion, alternate representations such as double precision or block floating-point can be used. Second, the dynamic behavior of the system is defined by appropriate scaling and quantization of the input signals and intermediate results. A careful selection of test signals is required to validate this step.

2. NUMBER REPRESENTATIONS

Floating-Point Representation

Representing infinitely many real numbers into a finite number of bits requires an approximate representation. Most calculations involving real numbers produce quantities that cannot be exactly represented using a finite number of bits. Therefore, not only the operands, but also the result of a floating-point calculation must often be rounded in order to fit into a finite representation.

There are two reasons why a real number is not exactly representable as a floating-point number. The decimal number 0.1 illustrates the most common situation. Although it has a finite decimal representation, it has an infinite repeating binary representation: the number 0.1 lies strictly between two floating-point numbers and is exactly representable by neither of them. Another, less common, situation is that the real number is out of range, i.e., its absolute value is larger than the maximum, or smaller than the minimum, floating-point value.

The IEEE floating-point standard, which is generally accepted by commercial hardware manufacturers, specifies the rounding method for basic operations (addition, subtraction, multiplication, division, and square root), and requires that implementations produce a *bit-accurate* result with that algorithm. By bit-accurate, we mean that the implementation produces a result that is identical bit-for-bit with the specification.

The IEEE floating-point representation is in the form $N = (-1)^S M 2^E$ where S is the sign bit, M, the (normalized) fractional mantissa, and E the (biased) exponent [2].

On most 32-bit systems (e.g. Intel Pentium-based personal computers), floating-point numbers are represented by the following data sizes:

- Single precision: $N_E = 8$, $N_M = 23$;
- Double precision: $N_E = 11$, $N_M = 52$;
- Accumulator: $N_E = 23$, $N_M = 64$;

where N_E is the number of bits in the exponent, and N_M , the number of bits in the mantissa.

The floating-point format allows a large dynamic range to be represented. The single precision format, for instance, can represent real values in the range $[1.17\ 10^{-38}; 3.40\ 10^{+38}[$. The main advantage of floating-point over fixed-point is its constant relative accuracy. However, it is more complex to use [8]. A floating-point multiplication requires the computation of the product of *M*s, the sum of *E*s, and the scaling of the result. A floating-point addition requires the scaling of *E*s, followed by the sum of *M*s, and a scaling of the result.

Fixed-Point Representation

In a fixed-point representation, the decimal point is actually an illusion–a formatting artifact that is always in the same position. We can actually imagine the fractional values to be scaled by a constant factor that would render it an integer. Programmers typically use the Q notation to indicate the virtual radix [6]. For example, *Q15* indicates a signed 16-bit fixed-point, of which 15 bits are fractional.

A fixed-point value is coded by 1 sign bit, N_1 bits before the point (integer word-length or *iwl*), and N_2 bits after the point (fractional word-length or *fwl*). In a 16-bit fixed-point DSP, numbers are typically represented in the following sizes:

- Memory and registers: $N_1 = 0$, $N_2 = 15$;
- Accumulator: $N_1 = 8$, $N_2 = 31$.

In the accumulator, N_1 holds 8 overflow bits, and N_2, a 15-bit MSB and 16-bit LSB. The fixed-point range is [-1; 1[in 16-bit memory locations and registers, and [-256; 256[in the 40-bit accumulator.

Further understanding of a particular Q format can be gained by calculating the *quantization step* q (smallest difference between two numbers that can be represented), and the $[-L; L[$ range of values that can be represented (signed and unsigned):

- $q = 2^{-\mathrm{fwl}}$;
- $[-L; L[_{signed} = [-2^{\mathrm{iwl}-1}; 2^{\mathrm{iwl}-1} - q]$;
- $[-L; L[_{unsigned} = [0; 2^{\mathrm{iwl}} - q]$.

For example, in *Q15*, the precision is 2^{-15}.

The principal advantage of fixed-point math is the inherent simplicity of its arithmetic operations. Addition and subtraction are the same operations as they are for ordinary integer values. Multiplication and division are expensive, divisions even more so. Therefore, we try to eliminate divisions by writing algorithms that, at most, contain divisions by a multiple of two, which, in effect, can be substituted by a right shift. In fact, when a number is divided by a multiple of eight, no actual shifting is necessary: the result is obtained by discarding the least significant byte(s). Multiplication involves intermediate values that are *2B* times larger than the operands (where B is the number of bits of the operands). This is realized in DSPs by having a 32-bit-wide product register to store the multiplication result of two 16-bit values. The addition of 2 B-bit signals gives $B + 1$ bits; which, when scaled back to B bits, may produce an overflow.

Signed Numbers

There is considerable variety in the way negative numbers are represented. By way of illustration, four common number representation for $B = 3$ are given in the following table:

Decimal value	Sign and magnitude	One's complement	Two's complement	Offset binary
+4	-	-	-	111
+3	011	011	011	110
+2	010	010	010	101
+1	001	001	001	100
+0	000	000	000	-
-0	100	111	-	011
-1	101	110	111	010
-2	110	101	110	001
-3	111	100	101	000
-4	-	-	100	-

Table 1: Signed Number Representations

Most DSPs use the two's complement representation. In the case of a signed fixed-point number with only fractional bits, the successive bits have the following value: $-(2^{-1})\ 2^{-2}\ 2^{-3}\ldots$

3. FIXED-POINT IMPLEMENTATIONS OF DIGITAL SIGNALS

When representing an analog signal in the digital domain, signals undergo non-reversible transformations to fit the finite representation format.

Quantization

Quantization is the process in which a quantity x with *infinite precision* is converted into a quantity x_Q that is approximately equal to x but can assume less different values than x. The relation between x and x_Q is called the *quantization characteristic*. The following conversion rules are applicable:

- Rounding-off (Fig. 1),
- Value truncation (Fig. 2), and
- Magnitude truncation (Fig. 3).

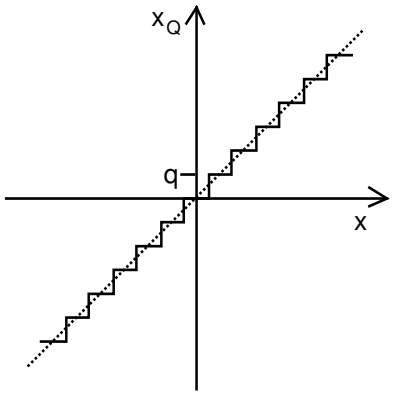

Fig. 1: Rounding Characteristic

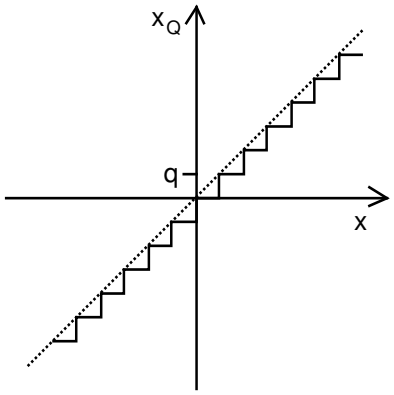

Fig. 2: Value Truncation Characteristic

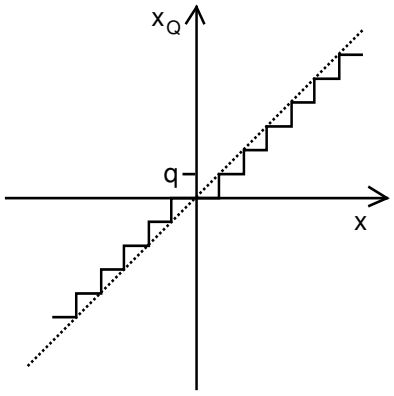

Fig. 3: Magnitude Truncation Characteristic

Overflow

Overflow takes care of signals that exceed the limits $[-L; L[$ enforced by the finite representation. In formal terms, we can describe this as a conversion of x into x_P, where the relation between of x and x_P is called the *overflow characteristic*. The following conversion rules can be applied:

- Saturation (Fig. 4),
- Zeroing (Fig. 5), and
- Sawtooth or wrap-around (Fig. 6).

Saturation is a specific DSP operation. The sawtooth characteristic comes for free when dropping the extra MSB in two's complement notation. In theory, any form of quantization can be combined with any form of overflow.

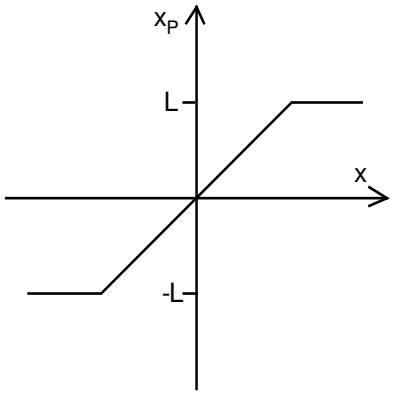

Fig. 4: Saturation Characteristic

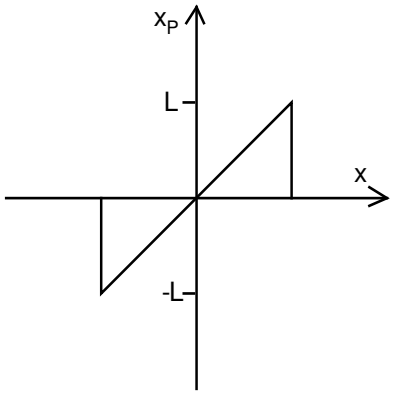

Fig. 5: Zeroing Characteristic

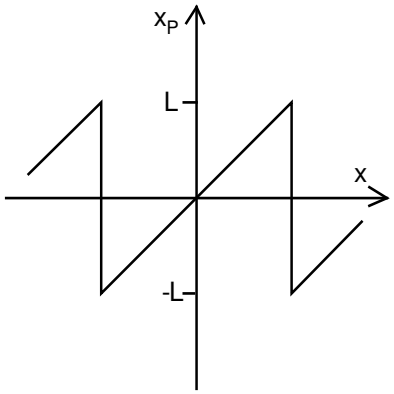

Fig. 6: Sawtooth Characteristic

Scaling

Scaling is a form of fixed-point operation with which we compress, or expand, the actual "physical" values of the original signal into a range suitable to the computer. By suitable, we mean that the computer will be able to store the input value into its registers, and use operations with sufficient range to hold intermediate results (in the accumulator or the product register for instance). We must ensure that during all of the calculation stages, we are using storage locations and operations suitable for the range of values required at that stage.

In practice, when scaling a signal, either at the input or at an intermediate stage, we generally have to check that the value we are using is still accurate enough (in terms of precision bits). The top and bottom of the range are usually a good choice for this check. If the values at both ends of the range scale correctly, it is reasonable to assume that all of the intermediate values will also scale correctly. We typically try to use power-of-two scaling so that this operation can be performed using shifts.

A comment of general validity is that an increase of the word length B by one bit improves the maximum achievable signal/quantization noise ratio by approximately 6dB [3].

In a multiplication, if both operands are scaled up by a factor N, the multiplication result is scaled up by N^2. To restore the original scaling, we need to divide the result by N. Before a division, we have to be careful not to scale down the denominator to zero. Usually, since division cancels the scaling, we keep the scaling at the numerator and ignore it at the denominator in order to obtain a scaled result without losing out on the resolution.

4. FINITE WORD-LENGTH EFFECTS

Finite word-length effects appear when converting continuous amplitude signals into discrete amplitude signals. Since the transition from continuous amplitude to discrete amplitude is in principle never completely reversible, there is always a loss of information, however slight. Finite word-length effects are very complicated to analyze because they are non-linear,
i.e., $x_Q + y_Q \neq (x + y)_Q$.

For example, by using the truncation characteristic on the fractional value, we can verify that $(3.7)_Q + (4.9)_Q \neq (3.7 + 4.9)_Q$.

A/D Conversion

In converting a signal of continuous amplitude x into a signal of discrete amplitude x_Q, we proceed as though x_Q was obtained by adding to x the noisy signal $e = x_Q - x$. In this way, we arrive at the useful concept of quantization noise. We generally assume that the error can be represented as a uniformly distributed white noise source, added to the ideal operation. This is not always true, but it gives an idea about the noise power in the filter. Other descriptions of the error include stochastic models [7].

Digital computations

The most complicated consequences of working with a finite word length are found when limiting the word length of intermediate results in digital systems. Just as in everyday calculations in the decimal system, addition and multiplication in the binary system often increase the word length. With recursive filters, this inevitably introduces problems because the result of one calculation is the starting point of the next. With non-recursive filters, the increase in the word length is always finite, but even then, some limitation may often be necessary.

In principle, we can choose the form of quantization and overflow to be used, and where they should be used in the filter. However, these choices can have completely different effects on the operation of the filter, and need to be studied thoroughly.

Limit Cycles

The great problem in analyzing the word-length limitation of intermediate results rests in the fact that, because of finite word-length effects, the filter is not completely linear. The most troublesome effects are usually due to overflow: signal distortion can then be very serious, and with recursive structures, there may even be *overflow oscillations* (also called *limit cycles*) of very large amplitude. Filters should therefore be designed so that this cannot happen.

It is found that quantization by means of magnitude truncation is not so liable to cause oscillation of this type as other forms of quantization [3]. It is also useful to know that limit cycles are always confined to a number of the least significant bits. If we reduce the quantization step q of all the quantizers in a filter where these cycles occur by increasing all the word lengths, then the amplitude of the limit cycle expressed in an absolute sense (e.g. the signal voltage) will also decrease. By quantizing the output signal, we can then make a filter where the limit cycles are negligible. In the case of audio signals, however, limit cycles can be very annoying, even at very low levels.

Several schemes can be used to prevent limit cycles [3]:

- Downscale the inputs, and the coefficients;
- Hide limit cycle (use more bits internally and quantize the output);
- Randomize the quantization.

A widely used aid in this context is scaling, particularly for filters consisting of cascaded sections. In the design, a multiplier by a constant factor smaller than one is inserted between the sections, to prevent the occurrence of overflow in the following section. If this factor is an integral power of two, the multiplication only means a shift of the sample value by one or more binary places. In some cases, scaling by a constant greater than one may be used if it appears during the design that the most significant bit would otherwise remain unused in the following section. After such measure, overflow does not have to be considered further. Then, an attempt is made to get some idea of the effects due to quantization of the intermediate results in the filter output signal. In practice there are distinct preferences for particular combinations of quantization and overflow.

5. FILTER DESIGN ASPECTS

In a practical digital filter, the number of bits used for representing the coefficients must be as small as possible, largely for reasons of cost. The values found must therefore be quantized. But this alters the characteristics of the filter, i.e., it changes the locations of the poles and zeros. These changes can be very substantial. It may happen that, after quantization, the filter no longer satisfies the design specifications that were used in calculating the real-valued coefficients. In extreme cases, a stable filter may even become unstable. However, the quantization of filter coefficients introduces no changes in the linear operation of a circuit, nor does it cause any effects that depend on the input signal or that vary with time. It introduces no more than a once-calculable change in the filter characteristics.

Some filter structures are much more sensitive to coefficients quantization than others. In general, a filter structure becomes less sensitive as the location of each pole and each zero becomes dependent on fewer coefficients.

In this part, we discuss various system descriptions that can be used to study the static behavior of filters. We shall confine ourselves to signals that can be abstracted to a function of a single independent variable representing *time*, while the value of the function itself is denoted as the *instantaneous amplitude*.

Difference Equation

Quite generally, a practical Linear Time-invariant Discrete (LTD) system can be described by:

$$y[n] = \sum_{i=0}^{N} b_i x[n-i] + \sum_{i=1}^{M} a_i y[n-i], \qquad (1)$$

where a_i and b_i are real constants. This equation is called a linear difference equation (of the *Mth* order) with constant coefficients.

Apart from being a good starting point for deriving other system descriptions, such as the system function, the difference equation gives a direct indication of a possible structure of a system. However, there can be many structures described by the same difference equation.

System function

The most abstract but at the same time the most versatile description of an LTD system is given by the system function *H(z)*. One, often very effective, manner of determining *H(z)* can be derived from the difference equation of the system:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{N} b_i z^{-i}}{1 - \sum_{i=1}^{M} a_i z^{-i}}. \qquad (2)$$

In practice, the analysis of an LTD system is often based on a block diagram. To determine the system function, a direct description is usually made by means of *z*-transforms. We can make considerable use of the property that a delay of one sampling interval in the time domain corresponds to multiplication by z^{-1} in the *z*-domain.

Poles and zeros

In eq. (2) we have found that the system function *H(z)* of practical LTD systems takes the form of a ratio of two polynomials in z^{-1}, i.e.:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + ... + b_n z^{-N}}{1 - a_1 z^{-1} - a_2 z^{-2} - ... - a_M z^{-M}}. \qquad (3)$$

We can always rewrite the numerator and the denominator as a product of factors:

$$H(z) = b_0 \frac{(z - z_1)(z - z_2)...(z - z_N)}{(z - p_1)(z - p_2)...(z - p_3)} z^{M-N}. \qquad (4)$$

The precise (complex) values of $z_1, z_2, \ldots, z_N$ and $p_1, p_2, \ldots, p_M$ depend on the (real) coefficients $b_0, b_1, b_2, \ldots, b_N, a_1, a_2, \ldots, a_M$ in eq. (3). The factor z^{M-N} can be neglected as it represents a simple shift of *h[n]* in time [3]. The poles and zeros fully determine the function *H(z)* and hence the corresponding LTD system, except for a constant b_0. The positions of poles and zeros are easily visualized in the complex z-plane. This gives the poles-and-zeros plot of *H(z)*, which is a very useful graphic aid.

We can derive some remarks of general validity for practically feasible systems that are relevant in our discussion in relation to quantization [3]:

(i) Since the frequency response corresponds to the system function on the unit circle in the z-plane, each pole and each zero has the most effect on the frequency range associated with the nearest part of the unit circle. Furthermore, the effect of a pole or zero on the frequency response increases the closer it is to the unit circle. In the extreme case of a zero actually on the unit circle, the amplitude of the frequency response is zero at the corresponding frequency, and there is a jump of π radians in phase at that point. On the other hand, a pole on the unit circle gives an infinite amplitude at the corresponding frequency, again with a phase jump of π radians.

(ii) In a stable system, all the poles are inside the unit circle; zeros may lie inside it, on it, or outside it.

These remarks show that a slight variation in the coefficient values may affect the locations of the poles and zeros and therefore completely alter the system behavior.

We shall not discuss further about other system descriptions, such as the impulse response, and the frequency response, although they might be useful in studying system characteristics.

Filter Coefficients

The effects of quantization can be verified by calculating and plotting the new frequency response and/or the pole-zero diagram. These effects can be reduced by using different filter structures (cascade connection instead of direct-form structures for higher order filters), or using other number representations (e.g. $0.9995 = 1.0 - 0.5\ 10^{-3}$).

Filter Structure

We can classify discrete filters by structure, as indicated by their block diagram. We have to remember that a particular structure is rarely unique: a specified filter characteristic can usually be achieved with different structures. Significant differences in characteristics between structures may also be found when the finite word-length effects of digital filters are taken into account.

We note that a Finite-Impulse Response (FIR) filter can only possess zeros (apart from the origin), while an Infinite Impulse Response (IIR) filter can have both zeros and poles [2]. In terms of stability, an FIR filter is always stable; an IIR filter, as we

have seen, is unstable if one or more poles lie on the unit circle or outside it.

Let us take a look at the Recursive Discrete Filters (RDFs). One possible structure follows directly from the general description of discrete filters in eq. (1): the *direct-form-I* structure (Fig. 7).

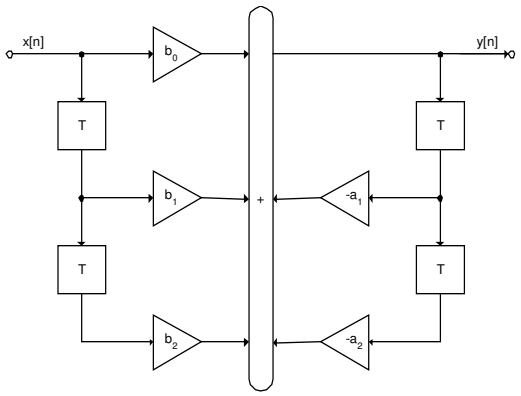

Fig. 7: Direct-Form-I Structure

We can think of it as split up into a transversal part with filter coefficients b_0, b_1, …, b_N followed by a purely recursive part with filter coefficients a_1, a_2, …, a_M. If the recursive part and the transversal part are made to change places–this is permissible because each part is an LTD system in itself–the result is the *direct-form-II* structure (Fig. 8).

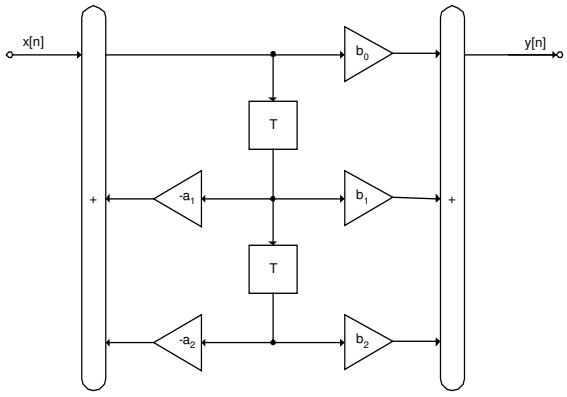

Fig. 8: Direct-Form-II Structure

The direct-form-I structure comprises $M + N$ unit-delay elements, whereas the direct-form-II structure only contains M unit-delay elements. The system function is the same in both cases; the zeros are determined by the coefficients b_i and the poles by the coefficients a_i. A disadvantage of the direct-form-II structure is that each coefficient b_i affects all of the zeros. The same thing applies for each a_i and all of the poles. This means that even small changes in the coefficients can have a considerable effect on the frequency response of the filter. This is definitely a serious drawback if we have to use quantized coefficient values in a digital filter. A solution to this problem can be found by building up the total filter from smaller units in which each pole and each zero is determined by a smaller number of coefficients. Filters of arbitrarily high-order can be produced by cascading 1st-order and 2nd-order sections, without the high-sensitivity to coefficient variations that characterizes the direct form structures. In some cases, double precision for the coefficients, the input data and/or the intermediate calculations might be required to prevent the filter from diverging. Additionally, we have to take particular care about a possible overflow in the 1st order section that might carry over to the 2nd order section, or go into the feedback loop, which can generate limit-cycles.

Alternative computation techniques can be used to prevent overflow in the intermediate calculations, as illustrated by the following examples:

$$y[n] = (\frac{b_o}{2}x[n] + \frac{b_1}{2}x[n-1] + \frac{b_2}{2}x[n-2] - \frac{a_1}{2}y[n-1] - \frac{a_2}{2}y[n-2]) \times 2 \quad (5)$$

$$y[n] = b_0 x[n] + \frac{b_1}{2}x[n-1] + \frac{b_1}{2}x[n-1] + b_2 x[n-2] - \frac{a_1}{2}y[n-1] - \frac{a_1}{2}y[n-1] - a_2 y[n-2] \quad (6)$$

Fig. 9 is an example of a recursive filter where scaling and saturation are used to prevent limit cycles:

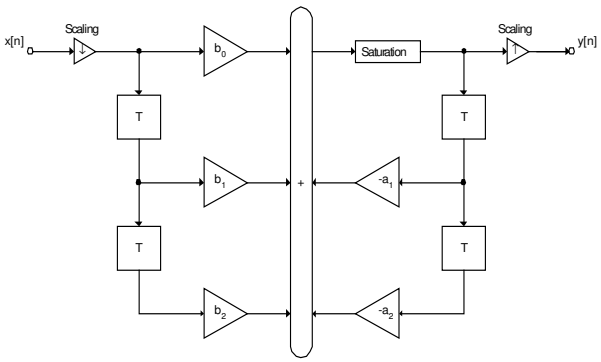

Fig. 9: Direct-Form-I Filter with Scaling and Saturation

6. A FLOATING-POINT TO FIXED-POINT CONVERSION METHODOLOGY

Let us discuss a widely used DSP algorithm implementation strategy. Generally, the algorithm parameters (e.g. the filter coefficients, etc.) are computed as real, or sometimes complex, numbers. A floating-point reference code is then generated from the algorithm blueprint, and tested with reference input signals. If, for cost reasons, a fixed-

point target processor is selected, the code needs to be ported from the floating-point reference code to a fixed-point implementation. Sometimes, an intermediary step includes a target-independent fixed-point model (usually in a high-level language like C), which is then followed by the target implementation (often in assembly). This intermediate step provides a way to reuse the fixed-point conversion effort for various target implementations that may contain hardware-specific optimizations. The final implementation is tested for bit-accuracy with the fixed-point reference code. This section focuses on the conversion from a floating-point reference code to a target-agnostic fixed-point model.

(i) The starting point of the conversion process is the floating-point reference code provided by the algorithm designer. This code may require large dynamic ranges as well as high precision; it may also include non-linear functions (e.g., logarithm, square root, cosine, etc.). Typically, some functions will require a block floating-point representation to maintain their accuracy.

(ii) The first pass of the conversion process consists in managing the signals dynamically by scaling the data to be in the [–1; 1[range. If the input value is smaller than the quantization step (i.e., $x < 2^{B-1}$), then the quantized value is set to zero: $x_Q = 0$. We should be careful about denominators here. Moreover, when applying scalars, we should keep in mind that one right shift means losing one bit of precision.

In terms of scaling, we determine that data can either be scaled statically or dynamically. In the former case, a fixed scalar, usually a power of two, is pre-defined, and all incoming data is divided (or shifted) by this factor. In the latter, the dynamic range of the data is evaluated at run-time, and a scalar is applied to the current block. The fixed scalar method is driven by the worst-case input and does not take into account characteristics of the current input buffer; hence, it can result in a loss of precision for low-level signals. On the other hand, dynamic scaling, sometimes also called block floating-point, maintains the largest Signal-to-Noise Ratio (SNR) possible. Dynamic scaling is a trade-off between double precision and single precision. By storing a normalized mantissa for each sample, and one common shift (scalar) value per block, dynamic scaling requires less storage space than double precision data, but needs more CPU time to compute the scale factor, normalize the buffer, and perform operations on the data. Formally, a block floating-point value can be described as: $A[i] = A_{Norm}[i]\ 2^{Scalar}$.

(iii) Then, we consider non-linear functions. We analyze the input data and make implementation choices, such as look-up tables versus polynomial approximations. The choice is mainly driven by the trade-off between precision, computational requirements and memory. Up to this point, we do not consider intermediate results.

(iv) The final step in this methodology deals with intermediate results and quantization effects. We apply the required arithmetic operators, study the effects of quantization on intermediate results, and use the accumulator (which provides the maximum precision) as much as possible.

7. AUDIO ALGORITHMS TESTING STRATEGY

In most audio systems, signals are represented in 16-bits, providing 96dB of dynamic range, which is deemed to be sufficient for human hearing characteristics. However, consecutive operations on the audio signal, especially scaling, may reduce the dynamic range (1 bit less equals 6dB decrease in the SNR). Quantization noise and saturation are to be taken into consideration since the human ear is susceptible to them. On the other hand, the human ear is largely insensitive to phase distortion.

Several test signals can be applied to assess the dynamic behavior of the system.

(i) Worst-case criterion

In the case of a Non-Recursive Discrete Filter (NRDF), the output signal can be defined as:

$$y[n] = \sum_{i=0}^{M} x[i]h[n-i]. \tag{7}$$

Assuming that |x[i]| ≤ 1.0, we have:

$$|\ y[n]\ | = \sum_{i=0}^{M} |\ h[i]\ |. \tag{8}$$

To get $|y[n]| \leq 1.0$, we can scale down the input signal by $\sum_{i=0}^{M} |\ h[i]\ |$.

However, this is a very pessimistic scaling; it should be used sparingly as downscaling reduces the number of bits, and thus, the SNR.

(ii) Second, we assume that the input signal is a sine wave. Then, the maximum gain for the signal is determined by the peak value of $|H(z)|$ for $z = e^{j\omega T}$ where $(-\pi \leq \omega T \leq \pi)$. To get $|y[n]| \leq 1.0$, we scale down the signal by $max(|H(e^{j\omega T})|)$. Other scaling methods exist but don't guarantee that there will be no overflow because they all make some assumptions on the input signal. The best bet is to find a *reasonable* compromise, and when overflow occurs, use saturation.

(iii) Lastly, we test the system by using white noise signals at various amplitudes. This lets us visualize the frequency response of the system. However, it is also recommended to check with other signals; in some instances, consecutive medium values could accumulate to generate a single large value that could overflow [5].

8. AN EXAMPLE: FLOATING-POINT TO FIXED-POINT CONVERSION OF AN IIR FILTER

We shall discuss a high-pass IIR filter that processes a speech signal input. The floating-point code is based on a direct-form-II filter structure (Fig. 8) whereas the fixed-point code has been implemented as a direct-form-I structure (Fig. 7).

The output of the filter is given by the equation:

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2]. \quad (9)$$

Filtering can be done with the following options:

- Single-precision or double-precision coefficients;
- Single-precision, double-precision or block scaling feedback loop;
- Static, dynamic or no scaling before and after the filter output.

Block scaling is used in order to optimize the code in terms of throughput and memory while maintaining a double-precision-like output.

The floating-point coefficients for a cut-off frequency at 50Hz have the following value:

$a_1 = -1.94447765776709$;
$a_2 = 0.945977936232282$;
$b_0 = 0.972613898499844$;
$b_1 = -1.94522779699969$;
$b_2 = 0.972613898499844$.

These coefficients have been converted into 16-bit fixed-points with the following value:

$a_1 = -1.944458$;
$a_2 = 0.9459839$;
$b_0 = 0.9725952$;
$b_1 = -1.945251$;
$b_2 = 0.9725952$.

A comparison of the floating-point frequency response with the fixed-point (single-precision coefficients) frequency response gives the theoretical expected behavior (Fig. 10).

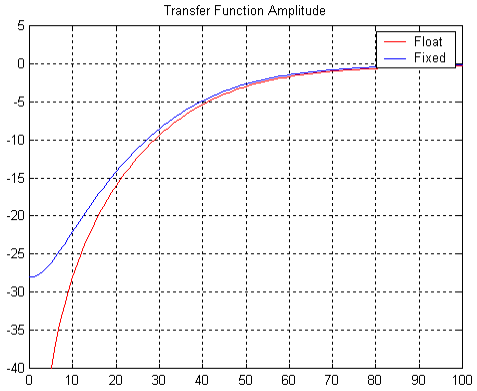

Fig. 10: Theoretical Floating-Point and Fixed-Point Frequency Response

For the following computations, we use single-precision fixed-point coefficients and low amplitude white noise as input. The resulting frequency responses are depicted in Fig. 11, 12 and 13, respectively for a single-precision, double-precision and block floating-point feedback loop.

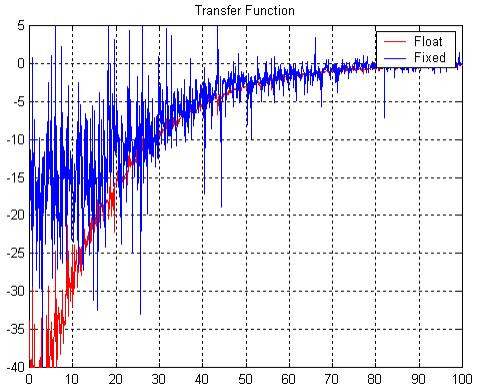

Fig. 11: Single-Precision Feedback Loop

In Fig. 11 and 13, the fixed-point filters diverge considerably from the expected response.

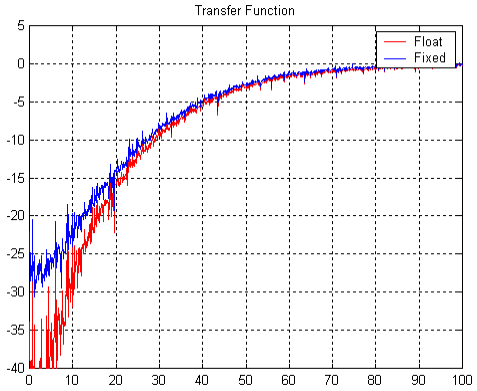

Fig. 12: Double-Precision Feedback Loop

The results in Fig. 12 show that the filter output follows its theoretical behavior when using double precision data.

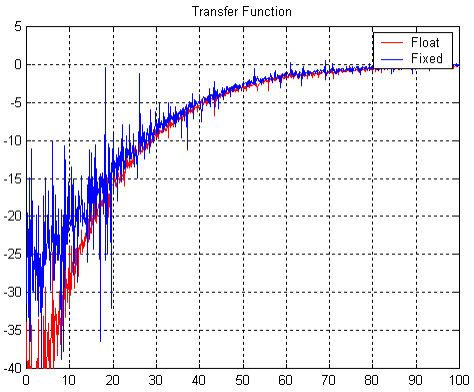

Fig. 13: Block-Scaling Feedback Loop

With maximum amplitude white noise and different scaling modes, we get the frequency responses depicted in Fig. 14 and 15.

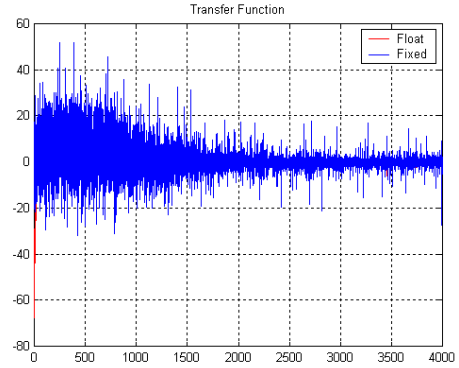

Fig. 14: Single-Precision Feedback Loop, No Scaling

In Fig. 14, with no scaling around the filter, we enter into limit cycles with high-amplitude signals.

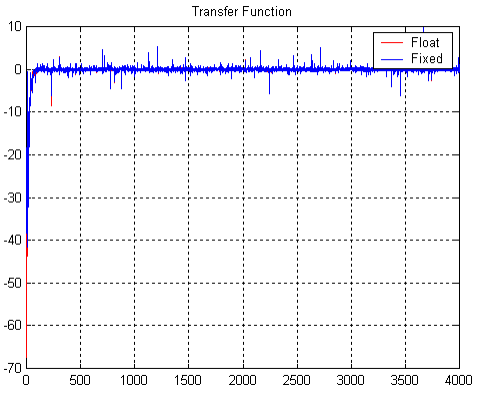

Fig. 15: Single-Precision Feedback Loop, Static Scaling

Fig. 15 shows a reasonable frequency response for static scaling. So we do not need to evaluate the system with dynamic scaling (which gives an identical frequency response).

Thus, we see that the optimum fixed-point solution for this filter is reached by using single-precision coefficients and a double-precision feedback loop with static scaling before and after the filter, with a structure similar to that shown in Fig. 9.

9. CONCLUSION

We have discussed how fixed-point arithmetic refers to the handling of numbers that are scaled up by a certain factor to allow space for fractional parts. Moreover, we have studied fixed-point DSP characteristics in order to understand the techniques for implementing a floating-point algorithm on a fixed-point processor.

Techniques described here reflect current industry practices to port algorithm designs to audio and multimedia consumer products by keeping costs down, while providing high-quality solutions. It is important for programmers to understand trade-off parameters in order to take informed design decisions.

Additional investigations in the area of floating-point to fixed-point conversion can be found in automatic conversion mechanisms, and fixed-point simulation tools.

REFERENCES

[1] A.V. Oppenheim, R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975.

[2] E.C. Ifeachor, B.W. Jervis, *Digital Signal Processing, A Practical Approach*, Addison-Wesley, 1993.

[3] A. W. M. van den Enden and N. A. M. Verhoeckx, "Digital Signal Processing: Theoretical Background," *Philips Tech. Rev.*, vol. 42, no. 4, pp. 110-144, Dec. 1985.

[4] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, Mar. 1991.

[5] R. Gordon, "A Calculated Look at Fixed-Point Arithmetic," *Embedded Systems Programming*, Apr. 1998, pp. 72-78.

[6] M. Jersák and M. Willems, "Fixed-Point Extended C Compiler Allows More Efficient High-Level Programming of Fixed-Point DSPs," *Proc. Int'l Conf. Sig. Proc. App. and Tech. (ICSPAT'98)*, Toronto, Canada, Sept. 1998.

[7] J. Tielen, "Restrictions when Implementing Filters on a Fixed-Point DSP," presentation, Philips, Leuven, Belgium, Mar. 2001.

[8] T. Gouraud, "Floating-Point to Fixed-Point Conversion: Some Ideas for Complex Algorithms," presentation, Philips, Leuven, Belgium, June 2000.

[9] M. Sarkar, *Fixed-Point Implementation of a High-Pass IIR Filter*, tech. report, Philips, Leuven, Belgium, July 2001.

[10] T. Gouraud, *Fixed Point Arithmetic*, tech. report, Philips, Leuven, Belgium, Apr. 2000.