FAST SINUSOID SYNTHESIS FOR MPEG-4 HILN PARAMETRIC AUDIO DECODING

Nikolaus Meine, Heiko Purnhagen

Laboratorium für Informationstechnologie
University of Hannover
Schneiderberg 32, 30167 Hannover, Germany
{meine,purnhage}@tnt.uni-hannover.de

ABSTRACT

Additive sinusoidal synthesis is a popular technique for applications like sound synthesis or very low bit rate parametric audio decoding. In this paper, different algorithms for the efficient synthesis of sinusoids on general purpose CPUs as found in today's PCs are investigated. Fast algorithms for time domain synthesis of constant and linearly changing frequencies are presented and compared to frequency domain synthesis approaches. Execution time and accuracy (SNR) of the algorithms are reported for different CPU types. Finally, the algorithms are implemented in a fast MPEG-4 HILN parametric audio decoder in order to evaluate their performance in a real world application.

1. INTRODUCTION

Sinusoidal modeling of audio signals is a popular technique because it nicely combines an efficient signal representation with the possibility of easy and intuitive signal modification. Furthermore, it is suitable for a broad range of real world audio signals, as these are mostly dominated by tonal signal components. Sinusoidal modeling has been utilized in musical applications for sound analysis/modification/synthesis, e.g. [1], [2], as well as in efficient (low bit rate) coding of speech and audio signals, e.g. [3], [4].

The generation of sinusoidal signals accounts for most of the computational complexity in additive sinusoidal synthesis. Both, time and frequency domain approaches to this problem have been presented, e.g. [5], [6], [7], [8]. The focus of this paper is on very efficient time domain synthesis of sinusoids on general purpose CPUs as found in today's PCs. This work was carried out in the course of the development of a fast decoder for HILN parametric audio coding as defined in the MPEG-4 standard [9], [10], [11].

In HILN coding, which stands for *Harmonic and Individual Lines plus Noise*, the input signal is separated into a series of overlapping frames which are decomposed into different signal components and then the model parameters for the components are estimated: *individual sinusoids* are described by their frequencies and amplitudes, a *harmonic tone* is described by its fundamental frequency, amplitude, and the spectral envelope of its partials, and a *noise* signal is described by its amplitude and spectral envelope. The modeling of transient components is improved by optional parameters describing their temporal amplitude envelope within a frame. Finally, the component parameters are quantized, coded, and multiplexed to form a bitstream. The target bitrate range of HILN is approximately 6 to 16 kbit/s, and typically an audio bandwidth of 8 kHz and a frame length (hop size) of 32 ms are used.

Because HILN supports sinusoidal trajectories with (slowly) varying amplitude and frequency that persist over several frames, the synthesis of sinusoids with constant frequency as well as synthesis of "sweeps" is addressed here. The proposed algorithms are implemented in floating point in order to utilize the FPU of modern CPUs and to achieve the required accuracy for 16 bit PCM waveforms.

This paper is structured as follows. Sections 2 and 3 present very efficient time domain synthesis algorithms for sinusoids with constant frequency and for sweeps, respectively. Section 4 discusses aspects of frequency domain sinusoidal synthesis. Experimental results for the execution time and the accuracy of the different algorithms are reported in Section 5. In Section 6, implementation in a fast MPEG-4 HILN parametric audio decoder is discussed. Finally, conclusions are drawn in Section 7.

2. TIME DOMAIN SYNTHESIS OF SINUSOIDS

2.1. Simple Oscillator

The signal $x[n]$ to be synthesized is a sampled sinusoid of amplitude A, frequency ω_0, and start phase φ_0. The parameters remain constant for one frame of length N.

$$x[n] = A\cos(\omega_0 n + \varphi_0) \quad \text{for } n = 0, 1, \ldots, N-1 \tag{1}$$

Exploiting the following property of the $\cos()$ function

$$2\cos\alpha\cos\beta = \cos(\alpha+\beta) + \cos(\alpha-\beta) \tag{2}$$

$$\cos(\alpha+\beta) = 2\cos\alpha\cos\beta - \cos(\alpha-\beta) \tag{3}$$

with $\alpha = \omega_0(n-1) + \varphi_0$ and $\beta = \omega_0$, an iterative calculation of $x[n]$ as described by Eq. 1 is possible. It consists of the initialization

$$x[0] = A\cos(\varphi_0) \tag{4}$$

$$x[1] = A\cos(\omega_0 + \varphi_0) \tag{5}$$

and the iteration step

$$x[n] = 2\cos(\omega_0)\,x[n-1] - x[n-2] \tag{6}$$

for $n = 2, 3 \ldots, N-1$.

The first two values $x[0]$ and $x[1]$ and the constant coefficient $2\cos(\omega_0)$ have to be calculated in a "setup" step before the actual iteration can start. Each $x[n]$ calculated by the iteration requires one multiplication and one addition. Even though only two floating point operations per sample are needed, the iteration gives very poor performance on modern general purpose CPUs. The reason is the dependency between the operations. Both operations need

the result of the preceding operation to proceed. Loop unrolling does not get around this problem, it is a property of the algorithm itself. So the algorithm must be modified in order to improve the performance.

2.2. Performance Optimization

To overcome the data dependencies, an additional hierarchy level is introduced. In the top level, only every l-th value of $x[n]$ is calculated using Eq. 7.

$$x[lm] = 2\cos(l\omega_0)x[l(m-1)] - x[l(m-2)] \tag{7}$$

The intermediate values can be obtained by interpolation of the two neighboring values. For $l = 2$ this requires only one additional constant which is revealed from Eq. 2 with $\beta = \omega_0$.

$$\cos\alpha = \frac{\cos(\alpha+\beta)+\cos(\alpha-\beta)}{2\cos\beta} \tag{8}$$

The highest speed has been achieved with $l = 4$. Using Eq. 7 to calculate every fourth value and interpolating these by applying Eq. 8 recursively two times results in an iterative algorithm with the initialization

$$x[n] = A\cos(\omega_0 n + \varphi_0) \tag{9}$$

for $n = 0, 1, \ldots, 4$, the iteration step

$$x[n] = 2\cos(4\omega_0)x[n-4] - x[n-8] \tag{10}$$

for $n = 8, 12, \ldots, N$, and the interpolation step

$$x[n-2] = \frac{1}{2\cos(2\omega_0)}(x[n-4] + x[n]) \tag{11}$$

$$x[n-3] = \frac{1}{2\cos(\omega_0)}(x[n-4] + x[n-2]) \tag{12}$$

$$x[n-1] = \frac{1}{2\cos(\omega_0)}(x[n-2] + x[n]) \tag{13}$$

for $n = 4, 8, \ldots, N-4$.

This iteration produces four new values by executing 8 floating point operations, which means two operations per value, as for the algorithm based on Eqs. 4 to 6. Nevertheless, an algorithm based on Eqs. 9 to 13 runs remarkably faster (more than twice as fast). Most of the operations are independent and can be executed in parallel. To achieve maximum speed, the iteration loop can be unrolled by a factor of three which makes it possible to eliminate any data move instructions.

2.3. Numerical Stability

Algorithms based on Eq. 6 represent an IIR filter with two poles $z_p = \exp(\pm j\omega_0)$ located on the unit circle. Rounding errors in the parameters do not change this property but may cause amplitude, phase and frequency to differ from the desired values. The optimized algorithm presented in Subsection 2.2 fails if $\cos(\omega_0) = 0$ or $\cos(2\omega_0) = 0$ because of the denominators in Eqs. 11 to 13. To guarantee numerical accuracy for all ω_0, a fallback algorithm is required if $|\cos(\omega_0)|$ or $|\cos(2\omega_0)|$ go below a certain limit. For further details, see Subsection 5.3.

3. TIME DOMAIN SYNTHESIS OF SWEEPS

3.1. Basic Algorithm

This Section presents an algorithm to synthesize a sweep $x[n]$, i.e., a cosine function with a start phase φ_0 and an instantaneous frequency that changes linearly from ω_0 at $n = 0$ to $\omega_0 + \Delta_\omega$ at $n = N$

$$x[n] = A\cos(\varphi[n]) \tag{14}$$

$$\varphi[n] = \int_0^n \left(\omega_0 + \frac{\Delta_\omega}{N}t\right)\, dt \; + \varphi_0 \tag{15}$$

$$= \frac{\Delta_\omega}{2N}n^2 + \omega_0 n + \varphi_0 \tag{16}$$

for $n = 0, 1, \ldots, N-1$. The instantaneous phase $\varphi[n]$ can be calculated incrementally by

$$\varphi[n] = \varphi[n-1] + \delta_1[n-1] \tag{17}$$

$$\delta_1[n] = \delta_1[n-1] + \delta_2 \tag{18}$$

resulting in

$$\varphi[n] = \delta_2\frac{n(n-1)}{2} + \delta_1[0]n + \varphi[0] \tag{19}$$

$$= \frac{\delta_2}{2}n^2 + \left(\delta_1[0] + \frac{\delta_2}{2}\right)n + \varphi[0] \tag{20}$$

where Eq. 20 must resemble Eq. 16. This gives the following initial values for the variables in Eqs. 17 and 18.

$$\varphi[0] = \varphi_0 \tag{21}$$

$$\delta_1[0] = \omega_0 + \frac{\Delta_\omega}{2N} \tag{22}$$

$$\delta_2 = \frac{\Delta_\omega}{N} \tag{23}$$

The complex function $X[n]$ is now defined as the analytic signal corresponding to $x[n]$.

$$X[n] = A\exp(j\varphi[n]) \tag{24}$$

$$x[n] = \mathrm{Re}(X[n]) \tag{25}$$

Applying the complex exponential function $\exp(j\alpha)$ to Eqs. 17 and 18 and to the start values $\varphi[0]$ and $\delta_1[0]$ and the constant δ_2 given in Eqs. 21 to 23 allows an iterative calculation of $X[n]$ with the initialization

$$X[0] = A\exp(j\varphi_0) \tag{26}$$

$$D_1[0] = \exp\left(j\left(\omega_0 + \frac{\Delta_\omega}{2N}\right)\right) \tag{27}$$

$$D_2 = \exp\left(j\frac{\Delta_\omega}{N}\right) \tag{28}$$

and the iteration step

$$X[n] = X[n-1]D_1[n-1] \tag{29}$$

$$D_1[n] = D_1[n-1]D_2 \tag{30}$$

for $n = 1, 2 \ldots, N-1$. Finally, $x[n]$ is obtained as the real part of $X[n]$ according to Eq. 25.

A complex multiplication requires six floating point operations, leading to a total computational complexity of 12 operations per sample.

3.2. Performance Optimization

Some optimization is possible by calculating only every second $X[n]$ and introducing a second phase difference to calculate $X[n+1]$ from $X[n]$. This leads to the initialization

$$X[0] = A\exp(j\varphi_0) \tag{31}$$

$$D_{1a}[0] = \exp\left(j\left(\omega_0 + \frac{\Delta_\omega}{2N}\right)\right) \tag{32}$$

$$D_{1b}[0] = \exp\left(j\left(\omega_0 + \frac{3\Delta_\omega}{2N}\right)\right) \tag{33}$$

$$D_{2a} = \exp\left(j\frac{\Delta_\omega}{N}\right) \tag{34}$$

$$D_{2b} = \exp\left(2j\frac{\Delta_\omega}{N}\right) \tag{35}$$

and the iteration step

$$X[n-1] = X[n-2]D_{1a}[n-2] \tag{36}$$

$$X[n] = X[n-2]D_{1b}[n-2] \tag{37}$$

$$D_{1a}[n] = D_{1a}[n-2]D_{2a} \tag{38}$$

$$D_{1b}[n] = D_{1b}[n-2]D_{2b} \tag{39}$$

for $n = 2, 4\ldots, N$.

In this iterative algorithm, $X[n-1]$ and $X[n]$ are calculated from $X[n-2]$, and $D_{1a}[n]$ and $D_{1b}[n]$ are calculated from $D_{1a}[n-2]$ and $D_{1b}[n-2]$. There are four complex multiplications for two samples, resulting in 12 floating point operations per sample, as above. But since the imaginary part of $X[n-1]$ is not needed, its calculation can be omitted. So three floating point operations are saved and the computational complexity becomes 10.5 floating point operations per sample.

4. FREQUENCY DOMAIN SYNTHESIS

4.1. Synthesis Algorithm

A sinusoid is highly localized in the frequency domain. This allows for an efficient approach to synthesize signals composed of many simultaneous sinusoids [7], [12, Section 2.5]. First, all sinusoids are accumulated in their frequency domain representation and then a single inverse fast Fourier transform (FFT) is applied to obtain the time domain representation of the composed signal.

For the frequency domain synthesis, an overlap-add approach with 50 % overlap has been chosen. For each frame of length N, $2N$ samples are synthesized (using an FFT with transform length $2N$) and then windowed by a windowing function $w(n)$. The first N samples of the current frame are added to the last N samples of the previous frame, and the last N samples of the current frame are saved for the next overlap-add step, resulting in a hop size of N samples. $w(n)$ must fulfill the condition of perfect reconstruction

$$w(n+N) + w(n) = 1 \quad \text{for } n = 0, 1, \ldots, N. \tag{40}$$

For the frequency domain synthesis, this windowing function is split into two factors $w_0(n)$ and $w_1(n)$. The first one is applied in the frequency domain in the form of a convolution while the second one is explicitly applied in the time domain after the transform has taken place. To maintain the perfect reconstruction property, $w_0(n)$ and $w_1(n)$ are chosen as follows, where c is an arbitrary constant.

$$w_0(n) = w(n)^c \tag{41}$$

$$w_1(n) = w(n)^{1-c} \tag{42}$$

For the synthesis, an oversampled frequency prototype $W_0(f)$ is generated by transforming the windowing function $w_0(n)$ into the frequency domain. For each sinusoid to be synthesized, this prototype is weighted by the desired complex amplitude A_0 and shifted to the position corresponding to the desired frequency f_0 (which generally does not fall exactly on the frequency grid of the $2N$ transform). The prototype $W_0(f)$ real-valued (due to symmetry of $w_0(n)$), but the amplitude and the Fourier spectrum are complex. The phase φ_0 of the sinusoid to be synthesized corresponds to the argument of the complex amplitude A_0. Since the bandwidth of the prototype is small (i.e., the frequency domain representation is highly localized), only a few values in the surrounding of this position have to be calculated by sampling the prototype at the integer positions of the transform bins. The prototype is sampled by linear interpolation between its pre-calculated, oversampled values. Figure 1 shows the synthesis process. A more detailed discussion of prototype oversampling and shifting can be found in [11].

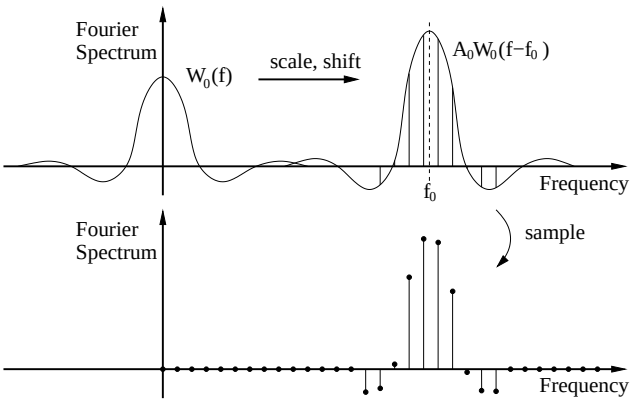

Figure 1: *Frequency domain synthesis.*

These spectral samples are added to a buffer which was initially set to zero. After all sinusoids have been added an inverse Fourier transform is applied to get a time domain signal. This signal is windowed by $w_1(n)$ and used in the overlap-add algorithm without further windowing to get the final output signal.

4.2. Windowing Function

A general approach for a windowing function that fulfills the conditions of perfect reconstruction is given by:

$$w_\infty(n) = \frac{1}{2} - \sum_{k=0}^{K-1} a_k \cos\left((2k+1)\frac{\pi}{N}n\right) \tag{43}$$

$$w(n) = \begin{cases} w_\infty(n) & \text{if } 0 < n < 2N \\ 0 & \text{else} \end{cases} \tag{44}$$

To achieve a high SNR (i.e., minimize the error caused by only using the central part of $W_0(f)$ as a prototype with limited width)

the bandwidth of $w_0(n) = w(n)^c$ must be minimized. Since the Fourier transform of $w_0(n)$ is mathematically hard to handle, the bandwidth of $w(n)$ is minimized instead.

The Fourier transform of $w(n)$ is the product of a sine function and a rational function. To achieve the lowest bandwidth, the coefficients a_k are calculated to minimize the order of the numerator. This condition is equivalent to one that demands as many derivatives of $w(n)$ as possible to be zero at the positions $n = 0$ and $n = N$. The solution for $K = 4$ is

$$a_k = \frac{1}{2048} \{1225, -245, 49, -5\} . \quad (45)$$

For all experiments, the order $K = 4$ and coefficients shown in Eq. 45 were used. A simple numerical optimization for best SNR gave $c = 0.82$. With these parameters and a prototype width of 20 transform bins, more than 100 dB SNR is reached.

4.3. Synthesis of Sweeps

For sweep synthesis, the frequency domain approach cannot be applied in the same way easily. Instead of only one prototype several would be needed, one for each sweep rate. Furthermore the bandwidth of the sweep prototypes is much higher so the main advantage of the frequency domain syntheses disappears.

A simple approach to synthesize sweeps is to use smaller hop sizes and treat the frequency as constant. The performance and accuracy of this approach are discussed in Subsection 5.2.

5. RESULTS

5.1. Test Setup and Execution Times

The presented algorithms were implemented in ANSI C, compiled by `gcc` 3.1 with optimization enabled, and tested on three different platforms running Linux:

- Intel Pentium III, 866 MHz (referred to as PIII)
- AMD Athlon XP1800+, 1533 MHz (referred to as K7)
- Compaq Alpha EV67, 667 MHz (referred to as Alpha)

The execution times are given in CPU clock cycles in Tables 1 and 2. The first part of the result pairs gives the clock cycles for the initial setup of the iteration, the second is the average number of clock cycles per synthesized sample in the iteration loop ($N = 1024$). Each test was run two times. In the first run, the calculated values were stored in memory ("store") while in the second run they were added to the values in the memory ("add"). The test routine runs the algorithm under test eleven times and takes the median execution time. The whole test program was invoked 10,000 times and the mean of the execution times is given here. Execution times for a "straight forward" implementation calling the $cos()$ function are given as reference.

5.2. Comparison with Frequency Domain Synthesis

Figure 2 shows the performance of the frequency domain algorithm compared to the time domain algorithm. The frequency domain algorithm always the overlap-add process as explained in Section 4. However, to synthesize sinusoidal trajectories that persist over several frames, no overlapping is needed for the time domain algorithm. In this case, time domain algorithm is the fastest for up to about 34 simultaneous sinusoids. For sinusoids existing in just a single frame, the HILN decoder uses a Hanning window of length $2N$ to achieve a smooth fade-in and fade-out. Hence, a 50 % overlap-add procedure is also required for time domain synthesis, so that the breakeven point lies at about 15 sinusoids.

Algorithm	PIII	K7	Alpha
$cos()$ function store	0 / 93	0 / 102	0 / 72
$cos()$ function add	0 / 95	0 / 105	0 / 73
simple iteration store	324 / 8.4	300 / 8.0	255 / 12.0
simple iteration add	324 / 9.2	300 / 8.0	264 / 13.0
optim. iteration store	397 / 3.8	369 / 2.9	394 / 2.0
optim. iteration add	397 / 5.1	338 / 4.2	382 / 2.5

Table 1: *CPU clock cycles for constant frequency sinusoid synthesis (setup / per sample).*

Algorithm	PIII	K7	Alpha
$cos()$ function store	0 / 98	0 / 105	0 / 74
$cos()$ function add	0 / 98	0 / 106	0 / 74
optim. iteration store	535 / 16.2	470 / 14.2	404 / 7.5
optim. iteration add	535 / 16.7	472 / 13.6	393 / 7.8

Table 2: *CPU clock cycles for sweep synthesis (setup / per sample).*

Using a shorter transform length reduces the time for the transform itself but increases the synthesis time because the width (in bins) of the prototype must remain the same to get the same SNR.

The FFT implementation used here is based on an optimized radix-4 algorithm. It has about the same performance as the FFTW algorithm by Matteo Frigo and Steven G. Johnson [13] for the transform lengths used here.

As noted in Subsection 4.3, the synthesis of sweeps in the frequency domain is difficult. Figure 3 shows the SNR that can be reached by approximating sweeps by short blocks of constant frequency sinusoids, i.e., a reduced hop size. Figure 4 shows the performance that can be achieved by this approach. HILN typically uses a frame length of 32 ms, corresponding to a default hop size of 512 samples for 16 kHz sampling rate. Hence, for practical hop sizes, the SNR is not sufficient to fulfill the MPEG-4 HILN conformance criteria [14]. The fundamental frequency of a speech signal, for example, can exhibit sweep rates of several 100 Hz/s.

5.3. Accuracy

Figures 5 and 6 show the SNR of the algorithms presented in Sections 2 and 3 respectively. The tests were carried out on a SUN UltraSPARC CPU which seems to have the most IEEE 64 bit floating point conform FPU. The x86-class CPUs generally achieve higher SNRs because they actually use 80 bit arithmetic. However, this advantage is only valid if the working set of variables can be hold in CPU registers, which is true for the sinusoid algorithm but not for the sweep algorithm. All tests use a frame length of $N = 1024$ an no overlap-add. In all diagrams, the y-axis displays the SNR in dB while the frequency ω_0 approaches one of the four critical points 0, $\frac{\pi}{4}$, $\frac{\pi}{2}$, and π. The x-axis shows the distance to the corresponding critical frequency (measured in terms of 2π). For each diagram, the lower bound of all curves represents the minimum SNR that is achieved by this algorithm.

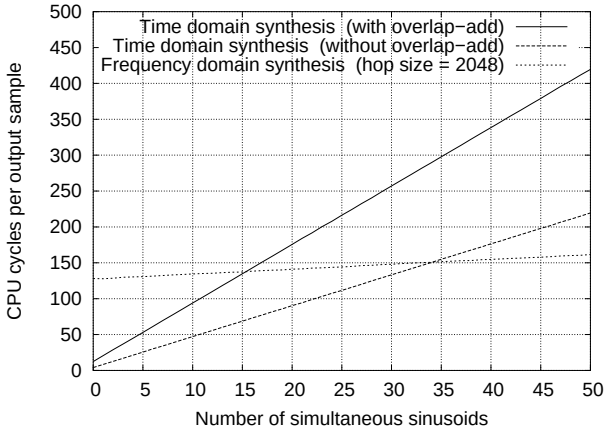

Figure 2: *Constant frequency sinusoid synthesis speed.*

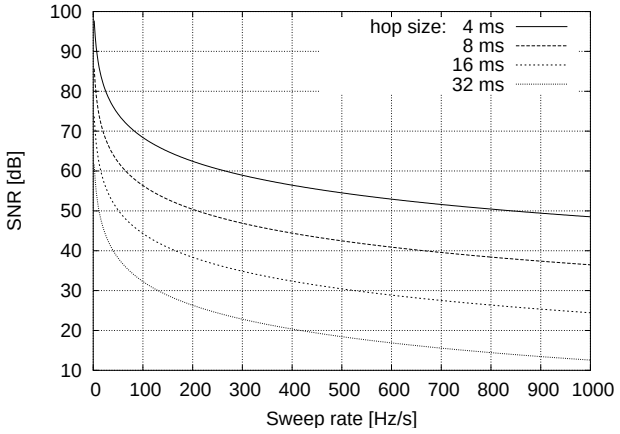

Figure 3: *Sweep approximation by overlap-add of constant frequency sinusoids.*

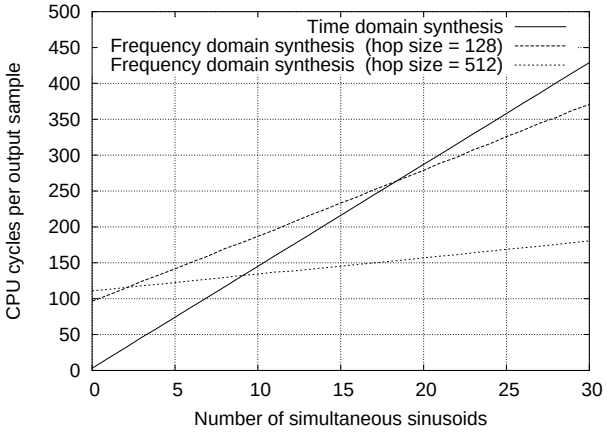

Figure 4: *Sweep synthesis speed.*

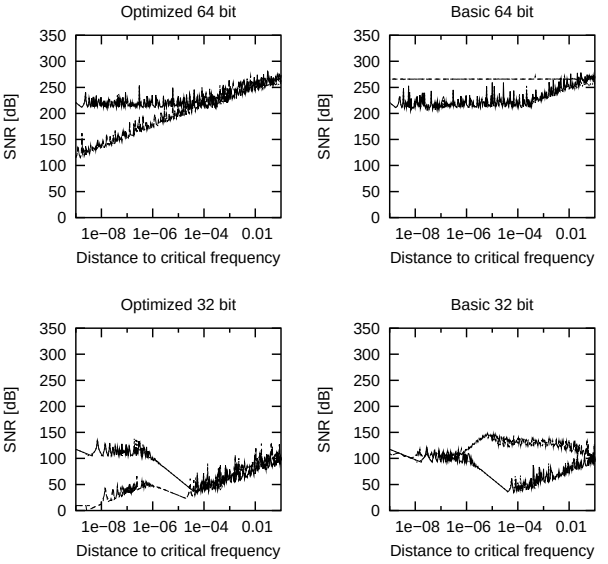

Figure 5: *Constant frequency sinusoid synthesis SNR.*

If using 64 bit arithmetic, the basic sinusoid algorithm (Subsection 2.1) gives an accuracy of more than 200 dB. The optimized algorithm (Subsection 2.2) fails to achieve 200 dB SNR when the frequency approaches the two critical points at $\omega_0 = \frac{\pi}{2}$ and $\omega_0 = \frac{\pi}{4}$ (lower curves). For $\omega_0 = 0$ and $\omega_0 = \pi$ the accuracy stays above 200 dB for both algorithms (upper curves). If falling back to the basic algorithm for frequencies whose distance to $\omega_0 = \frac{\pi}{2}$ and $\omega_0 = \frac{\pi}{4}$ is less than 10^{-4}, an accuracy of 200 dB is always achieved and the fallback algorithm is called in less than one out of 1000 cases. In case of 32 bit arithmetic both algorithms do not achieve a sufficiently high SNR to generate 16 bit PCM waveforms.

For the sweep synthesis algorithm (Subsection 3.1), there are no critical frequencies The optimized algorithm (Subsection 3.2) performs even better than the basic one because of less error propagation. The SNR stays well above 200 dB for 64 bit arithmetic and reaches about 100 dB for 32 bit arithmetic.

6. FAST HILN PARAMETRIC AUDIO DECODER

For the application in an MPEG-4 HILN Audio Decoder the time domain synthesis has several advantages. To be MPEG-4 conform [14], a decoder must exactly apply the prescribed windowing functions to the synthesized signal frames which may include an arbitrary triangular envelope. This is easily possible when using the time domain approach because all sinusoids are synthesized individually but it would require additional Fourier transform operations in the frequency domain approach. Even if strict MPEG-4 conformance is given up, the performance of the frequency domain synthesis does not reach the performance of the time domain synthesis in an MPEG-4 HILN decoder.

Based on the time domain synthesis algorithms presented here, a fast MPEG-4 HILN parametric audio decoder was implemented. For typical bit streams encoded at 6 to 16 kbit/s, a CPU load of approximately 10 to 20 MHz on a Pentium III was required for

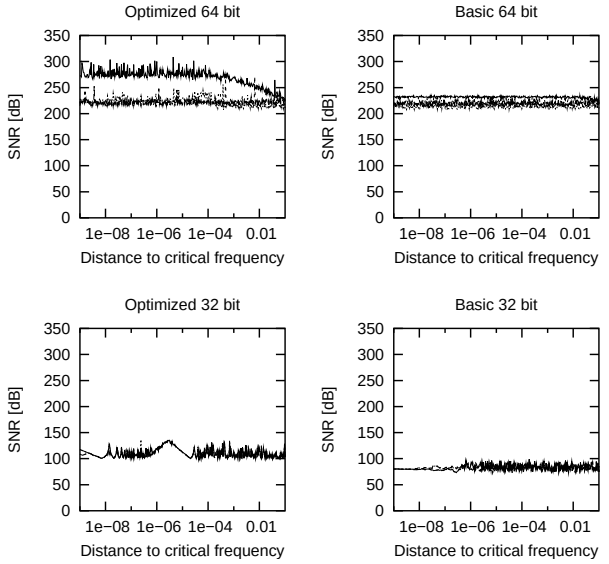

Figure 6: *Sweep synthesis SNR.*

decoding.

Table 3 shows the distribution of the CPU cycles while decoding a set of typical bitstreams. Even though the sinusoid and sweep synthesis take the largest part of CPU cycles, nearly half the cycles are consumed by other parts of the decoder. The mean synthesizer load per frame was 18.6 constant frequency sinusoids (i.e. 9.3 with overlap) and 8.1 sweeps.

Part	Execution Time
setup for sinusoids	4.94 %
iteration for sinusoids	18.34 %
setup for sweeps	3.94 %
iteration for sweeps	32.29 %
parameter decoding	5.74 %
audio output	10.80 %
everything else	23.95 %

Table 3: Execution times in an HILN decoder.

7. CONCLUSIONS

The presented algorithms are optimized for general purpose CPUs and give good performance here. For high precision, 64 bit floating point arithmetic is needed. This does not cause performance problems on most modern CPUs because their FPU hardware can handle at least 64 bit floating point values and the signal buffers fit into the CPU's primary cache which has enough bandwidth for this application.

Dedicated digital signal processors (DSP) typically provide less accuracy, e.g. 24 bit fixed point or 32 bit floating point. For these systems, various other sinusoid synthesis algorithms have been developed that can be more appropriate ([5], [6]).

The algorithms presented here are well suited for MPEG-4 HILN parametric audio decoding on general purpose CPUs as well as for software audio synthesizers utilizing sinusoidal representations. The flexibility, accuracy, and performance of the presented time domain synthesis algorithms is superior to frequency domain based algorithms in the given environment, i.e. for the synthesis of a few ten simultaneous sinusoids or sweeps. Both the sinusoid and sweep synthesis algorithm allow real time parameter modifications by re-calculating the state variables from updated parameters. For the sweep algorithm, the amplitude and sweep rate can be changed instantly without affecting the other state variables, which may be an interesting feature for sound synthesis.

8. REFERENCES

[1] J. L. Flanagan and R. M. Golden, "Phase vocoder," *Bell System Technical Journal*, pp. 1493–1509, Nov. 1966.

[2] J.-C. Risset and M. V. Matthews, "Analysis of musical instrument tones," *Physics Today*, vol. 22, pp. 22–30, Feb. 1969.

[3] R. McAulay and T. Quatieri, "Speech analysis/synthesis based on a sinusoidal representation," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 34, no. 4, pp. 744–754, Aug. 1986.

[4] H. Purnhagen, "Advances in parametric audio coding," in *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, Mohonk, New Paltz, Oct. 1999, pp. 31–34.

[5] S. A. Azizi, "Implementation of quadrature sinusoidal oscillator with reduced round off noise," in *AES 100th Convention*, Copenhagen, May 1996, Preprint 4237.

[6] J. Dattorro, "Effect design – part 3 oscillators: Sinusoidal and pseudonoise," *J. Audio Eng. Soc.*, vol. 50, no. 3, pp. 115–146, Mar. 2002.

[7] A. Freed, X. Rodet, and P. Depalle, "Synthesis and control of hundreds of sinusoidal partials on a desktop computer without custom hardware," in *Proc. Int. Conf. Signal Processing Applications & Technology*, Santa Clara, CA, US, 1993.

[8] T. Hodes and A. Freed, "Second-order recursive oscillators for musical additive synthesis applications on SIMD and VLIW processors," in *Proc. Int. Computer Music Conf. (ICMC)*, 1999.

[9] ISO/IEC, "Coding of audio-visual objects – Part 3: Audio (MPEG-4 Audio Edition 2001)," ISO/IEC Int. Std. 14496-3:2001, 2001.

[10] ISO/IEC JTC1/SC29/WG11, "MPEG-4 Audio Web Page," available: `http://www.tnt.uni-hannover.de/project/mpeg/audio/`.

[11] H. Purnhagen, N. Meine, and B. Edler, "Speeding up HILN – MPEG-4 parametric audio encoding with reduced complexity," in *AES 109th Convention*, Los Angeles, Sept. 2000, Preprint 5177.

[12] M. M. Goodwin, *Adaptive Signal Models: Theory, Algorithms, and Audio Applications*, Kluwer, Boston, MA, USA, 1998.

[13] M. Frigo and S. G. Johnson, "FFTW – Fastest Fourier Transform in the West," available: `http://www.fftw.org/`.

[14] ISO/IEC, "Coding of audio-visual objects – Part 4: Conformance testing AMENDMENT 1: Conformance testing extensions (MPEG-4 Conformance Version 2)," ISO/IEC Int. Std. 14496-4:2000/Amd.1:2001, 2001.