FAST ADDITIVE SOUND SYNTHESIS USING POLYNOMIALS

Matthias Robine, Robert Strandh, Sylvain Marchand

SCRIME – LaBRI, University of Bordeaux 1
351 cours de la Libération, F-33405 Talence cedex, France
firstname.name@labri.fr

ABSTRACT

This paper presents a new fast sound synthesis method using polynomials. This is an additive method, where polynomials are used to approximate sine functions. Traditional additive synthesis requires each sample to be generated for each partial oscillator. Then all these partial samples are summed up to obtain the resulting sound sample, thus making the synthesis time proportional to the product of the number of oscillators and the sampling rate. By using polynomial approximations, we instead sum up only the oscillator coefficients and then generate directly the sound sample from these new coefficients. Most of computation time is consumed by a data structure that manages the update of the generator coefficients as a priority queue. Practical implementations show that Polynomial Additive Sound Synthesis (PASS) is particularly efficient for low-frequency signals.

1. INTRODUCTION

We present a method for additive sound synthesis with a complexity that is proportional to the sum of the frequencies of the oscillators, as opposed to the traditional method whose complexity is proportional to the number of the oscillators, regardless to their frequencies. Thus, the new method is especially efficient for low frequencies. Perhaps more surprisingly, this method is not so dependent on the sampling frequency. This makes our new method especially well-suited for high sampling frequencies.

First, we review in Section 2 the principles of classic additive synthesis, as well as the methods proposed for real-time implementations. Then we present in Section 3 our new method using polynomials, with a lower complexity. We explain how to approximate with polynomials the sine functions of the sinusoidal oscillators, and how to generalize this approximation for all the partials. All the polynomial coefficients from these approximations are then summed, becoming coefficients of a global polynomial generator. Thus, sound samples are computed from a single polynomial.

As the approximation of each partial is limited in time (on a part of a period of the sine function), its coefficients must be updated. We propose in Section 4 to manage all these update events with a priority queue, efficiently implemented as a binary heap. The element with the highest priority is always the next update event to be processed. We explain how to use this data structure for our method. We show then in Section 5 how this priority queue can again be useful to manage the change of the amplitudes and frequencies of the partials at the right moment.

Finally we present some performance results in Section 6 and we compare our method with other techniques proposed in the literature.

2. ADDITIVE SOUND SYNTHESIS

Additive synthesis (see for example [1]) is the original spectrum modeling technique. It is rooted in Fourier's theorem, which states that any periodic function can be modeled as a sum of sinusoids at various amplitudes and harmonic frequencies. For stationary pseudo-periodic sounds, these amplitudes and frequencies evolve slowly with time, controlling a set of pseudo-sinusoidal oscillators commonly called *partials*. This is the well-known McAulay-Quatieri representation [2] for speech signals, also used by Serra [3] in the context of musical signals. As they evolve slowly in time, we consider the frequencies and amplitudes as constant for a short length. The audio signal s can be calculated from the sum of the partials using:

$$s(t) = \sum_{i=1}^{N} a_i \sin\left(2\pi f_i t + \phi_i\right) \tag{1}$$

where N is the number of partials in the sound and the parameters of the model are f_i, a_i, and ϕ_i, which are respectively the frequency, amplitude, and initial phase of the partial number i. This equation is valid if the frequency is constant. However, for practical sound examples, both the frequency and the amplitude must be updated regularly. Equation 1 then holds for each sound segment between two update times.

In the general approach derived from Equation 1, for each sample the partials are processed separately, and then summed. Thus the complexity of the method is proportional to the product of the number of partials and the sample rate. Computing the sine function for every partial and every sound sample can be very time-consuming. Using additive synthesis to synthesize a whole orchestra is a big challenge, and even more so if we want to do it in real time. This is why we need to reduce the computation time of the additive synthesis, while keeping the control of all the parameters of the sound partials in time.

The most straightforward – rather naive – way to calculate a partial contribution is to use the sine function. But it consumes a lot of computation time. Other techniques are possible, such as the use of the digital resonator method (see for example [4, 5]), which computes the samples of each separate partial with an optimal number of operations. In this method, the sine is calculated with an incremental algorithm that avoids computing the sine function for every sample. We proposed the use for fast additive synthesis of the digital resonator with floating point arithmetic in [6, 7]. For each partial the resonator is initialized as Equation 2 shows, with F_s the sampling rate of the synthesis, a, f, and ϕ respectively the amplitude, frequency, and initial phase of the partial, and Δ_ϕ the phase increment. The incremental computation of each oscillator

sample requires only 1 multiplication and 1 addition.

$$\begin{cases} \Delta_\phi & = & \frac{2\pi f}{F_s} \\ s[0] & = & a\sin(\phi_0) \\ s[1] & = & a\sin(\phi_0 + \Delta_\phi) \\ C & = & 2\cos(\Delta_\phi) \\ \\ s[n+1] & = & C \cdot s[n] - s[n-1] \end{cases} \quad (2)$$

This algorithm is optimal in a sense that 1 multiplication with no addition will lead to a geometric progression, whereas no multiplication with 1 addition will lead to an arithmetic one; none of these progressions being a sine function. Again for the purpose of real-time additive synthesis, we then proposed to limit the number of partials to be synthesized by removing inaudible ones, using a psychoacoustic model together with an efficient data structure [8].

In order to efficiently synthesize many sinusoids simultaneously, Freed, Rodet, and Depalle propose in [9] to use the inverse Fourier transform, provided that the oscillator parameters vary extremely slowly. The idea is to reconstruct the short-term spectrum of the sound at time t, by adding the band-limited contribution of each partial, then to apply the Inverse Fast Fourier Transform (IFFT or FFT^{-1}) in order to obtain the temporal representation of the sound, and finally to repeat the same computation further in time, thus performing a kind of "inverse phase vocoder". The gain in complexity is when the number of oscillators is large in comparison to the number of samples to compute at each frame. This approach is very interesting, because its complexity is no more the product of the number of partials and the sampling rate. However, the control of the additive parameters is more complex.

3. USING POLYNOMIALS

Polynomials have been traditionally used in order to model the parameters of the sinusoidal model [10, 11, 12]. Here, we propose to use polynomials to replace the sine function. Our method consists of first calculating a set of polynomial coefficients for each partial. Polynomial values from polynomials computed with these coefficients approximate the signal of the partial on a part of its period. The classic approach would evaluate the polynomial associated to each oscillator, and then sum up the results, which is quite inefficient. The idea is yet to sum the coefficients in a polynomial generator, then to evaluate the resulting polynomial only once. Indeed, summing polynomials leads to another polynomial of the same degree. The sound samples can be computed from this single resulting polynomial, with a fairly low degree – independent of the number of partials to synthesize. The general process is illustrated by Figure 1.

3.1. Partial Approximation

The time-domain signal generated by each partial is defined by a sine function. We propose to approximate this function by a polynomial. To get the polynomial coefficients that can approximate any partial of a sound, we decide to first approximate a unit signal u with amplitude $a = 1$, frequency $f = 1$, and phase $\phi = 0$, *i.e.*:

$$u(t) = \sin(2\pi t)$$

We have to choose a part of the period where we will do the approximation. We call this part the validity period p of the polynomial coefficients. Thus, if we approximate a half period of u, then $p = 1/2$.

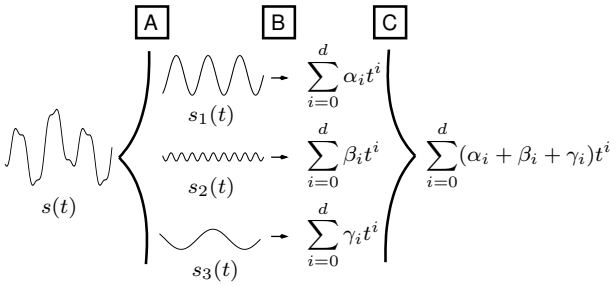


Figure 1: *PASS.* Step A: *A periodic signal can be divided into sinusoidal components.* Step B: *Computing polynomial coefficients to approximate the signal for each partial.* Step C: *A polynomial generator is obtained by summing the coefficients from the polynomials of the partials. The values computed by the generator will be the samples of the sound signal.*

Measuring the performance of the approach of a signal $u(t)$ by a polynomial $U(t)$ on a validity period p can be done using the SNR ratio given by:

$$\text{SNR} = 10\log_{10} \frac{\int_0^p u^2(t)dt}{\int_0^p (u(t) - U(t))^2 dt}$$

For a given polynomial degree, we propose to find the polynomial coefficients that minimize the value of the denominator of the SNR:

$$\int_0^p (u(t) - U(t))^2 dt$$

These coefficients have to respect other constraints to maintain a piecewise continuity. For example, with a 2-degree polynomial U and $p = 1/2$, to impose a C^1 continuity, it is sufficient that $U(0) = U(1/2) = 0$, and the coefficients a_i that minimize:

$$\int_0^{\frac{1}{2}} (\sin(2\pi t) - (a_0 + a_1 t + a_2 t^2))^2 dt$$

are:

$$\begin{cases} a_0 = 0 \\ a_1 = 240/\pi^3 \\ a_2 = -480/\pi^3 \end{cases}$$

To approach the sine function, we can use in this case alternately U_a for a first half period and U_b for the second:

$$\begin{cases} U_a(t) = a_1 t + a_2 t^2 \\ U_b(t) = -a_1 t - a_2 t^2 \end{cases}$$

The choice of the validity period, and the highest degree of the polynomials to use, have a big influence on the performance of the approximation, as shown in Table 1. We explain in 3.3 how a high polynomial degree can lead to numerical instability of the method, or in Section 6 why the choice of a short validity period increases the computation time. We can note that using a validity period $p = 1/2$ with a polynomial degree $d = 2$ is particularly suited for a very fast synthesis, and that using a validity period $p = 1/2$ with a polynomial degree $d = 4$ is suited for a fast synthesis with good quality.

The coefficients we compute define a unit polynomial $U(t)$ by validity period. When the unit polynomial is found, every partial can be approximated from it. In the general case of a partial i with

amplitude a_i, frequency f_i, and initial phase ϕ_i, the approximating polynomial P_i is then given by:

$$P_i(t) = a_i U\left(f_i t + \frac{\phi_i}{2\pi}\right)$$

Notice that the amplitude, frequency, or phase parameters do not modify the approximation error given in Table 1. In addition to the sinusoid, the polynomial approximation generates a noise consisting of harmonics of this sinusoid. The magnitudes of these harmonics are small, and depend on the validity period p and the polynomial degree d.

p	d	C^0 SNR (dB)	C^1 SNR (dB)
1/4	2	36	28
1/4	3	57	28
1/4	4	79	59
1/4	5	102	59
1/2	2	28	28
1/2	3	28	28
1/2	4	59	59
1/2	5	59	59
1	4	17	17
1	5	42	42

Table 1: *Error of polynomial approximation of a partial. For two different continuity requirements (C^0 and C^1), the Signal-to-Noise Ratio (SNR) obtained with the approximation error $u - U$ compared to the target signal u are shown as functions of the validity period p and the polynomial degree d.*

Since for now we consider only constant parameters for the partials, the generated functions are periodic. It is thus possible to compute the polynomial coefficients for only one period of any partial.

Each set of polynomial coefficients is valid for a part of the period of the sine function. For example, if we choose to approximate sine functions using a fourth of their period, we need to compute four sets of coefficients by partial. As long as the amplitude and the frequency of a partial are constant, we can continue with the same pre-calculated sets. During the sound synthesis, the coefficients that approximate the partials must be updated regularly (the rate depending on the frequency of each partial), and must also be changed if the parameters have changed.

3.2. Incremental Calculation of Polynomials

To avoid the problem of computing a polynomial with large time values, leading to numerical imprecision, we propose to use the Taylor's theorem to compute it. The polynomial can be evaluated at every instant $t_0 + \Delta_t$ by using the value and the values of its derivatives at a preceding instant t_0:

$$P^k(t_0 + \Delta_t) = P^k(t_0) + \sum_{i=k+1}^{d} \frac{\Delta_t^{i-k}}{(i-k)!} P^i \qquad (3)$$

where P^k is the k-th derivative of the polynomial function (P^0 being the polynomial itself). The number of necessary values depends on the degree of the polynomial (*e.g.*, three values with a 2-degree polynomial).

With the polynomial coefficients of the partials obtained according to the method presented in Section 3.1, we compute the first value of the polynomial and of its derivatives. To compute each of the following values, we use Equation 3 with a step Δ_t corresponding to the time between two time events. A time event is either the time of a sound sample or of a scheduled update of the coefficients. When time reaches or exceeds the validity period we have chosen, the coefficients are updated, and the incremental algorithm goes on with the new coefficients.

3.3. Polynomial Generator

Using this incremental method for each individual partial would be very expensive in terms of computation time. For that reason, we propose a technique in which we sum the coefficients to compute only a global polynomial, the generator. During the synthesis, the generator is computed incrementally. When a partial reaches the end of its validity period, the different values of the generator (value and values of the derivatives) are summed with the new values from the partial. When a sound sample must be produced, the generator is computed to get the sound sample value.

As the generator is computed incrementally, we have to care about numerical precision: using the preceding values to compute new ones accumulates floating-point precision errors in the result. Thus, there is a validity limit for updating the generator. According to the polynomial degree used, to the number of partials in the sound, and the floating-point precision we can use, we need to re-initialize the generator coefficients regularly with the authentic sine function.

The complexity of our method is dominated by the management of the update events from individual partials. To optimize this process, we propose the use of an efficient data structure, in a way similar to that of [8] which uses a skip-list to increase the performance of additive synthesis. In the PASS method, we use a priority queue implemented as a binary heap to manage the update events from the partials.

4. DATA STRUCTURE

4.1. Using a Heap as a Priority Queue

A priority queue is an abstract data type supporting two operations: *insert* adds an element to the queue with an associated priority; *delete-max* removes the element from the queue that has the highest priority, and returns it. We use a priority queue to manage update events from partials of the sound. During synthesis, update events are regularly inserted in, or removed and processed.

The standard implementation of priority queues is based on binary heaps (see for example [13]). With this implementation, queue operations have $O(\log(N))$ complexity, with N the number of elements in the queue. A binary heap is a binary tree satisfying two constraints:

1. the tree is either a perfect binary tree, or, if the last level of the tree is not complete, the nodes are filled from left to right;
2. each node is greater (in priority) than or equal to each of its children. The top of the binary heap is always the next event to process.

4.2. Heap Optimization

Most of the computation time of the PASS method is due to the management of the heap. Consequently, heap primitives need to be highly optimized. Our first approach was to *delete-max* the priority queue, to process the update event and to *insert* a new one in the queue. With this approach, the heap is substantially reorganized twice for each insert / delete pair of operations.

To improve the performance, we replaced the insert / delete primitives with *top* and *replace*. *top* returns the element of the queue that has the highest priority, without removing it. It is possible to process an update event without removing it from the queue. The *replace* method replaces the element from the queue that has the highest priority with an other element by initially putting it on top of the heap, and then letting it trickle down according to normal heap-reorganization primitives. In the worst case, the *replace* method needs $O(\log(N))$ operations, N being the number of elements in the heap. The heap is reorganized only once per update.

Partials with high frequencies must be updated more often than the others, because their validity period is smaller. With the *replace* method, the update events concerning high frequencies stay near to the top of the heap. Using fewer operations for the most frequently updates improves the complexity of our method. Figures 2 and 3 give an example of heap management, where the *delete-max* then *insert* methods use 4 elements swaps, whereas the *replace* method takes only 1 swap.

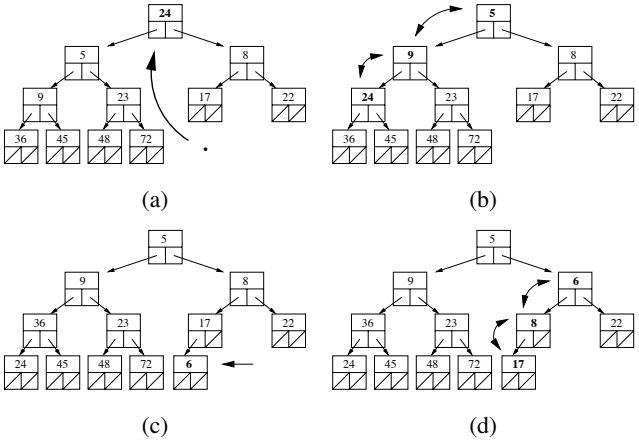


Figure 2: delete-max *then* insert *priority queue methods within a heap. (a) Delete-max of the element with highest priority. (b) Heap reorganization. (c) Insert of the new element. (d) Heap reorganization.*

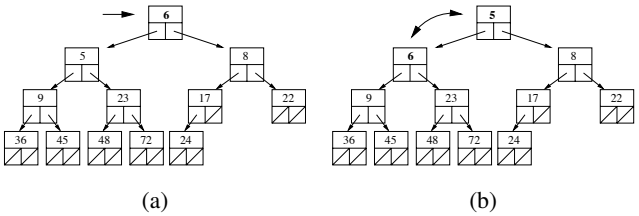


Figure 3: replace *priority queue method with a heap. (a) Replace of the element with highest priority by the new one. (b) Heap reorganization.*

5. CHANGE OF SOUND PARAMETERS

We consider that the parameters of the partials are constant on a short length. But since they change, they have to be updated regularly. In [14] we indicate the best time in a period to change parameters of a partial, as illustrated by Figure 4 and 5. The best moment to change the amplitude is when the signal is minimal, to preserve the continuity of the signal. And the best moment to change the frequency is when the signal is maximal, to preserve the continuity of the signal derivative.

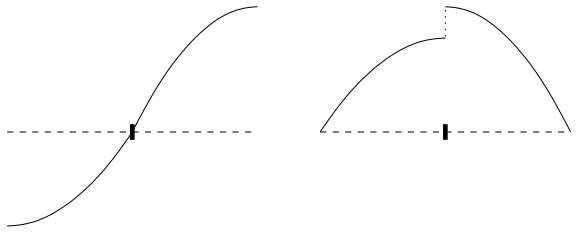


Figure 4: *Changing the amplitude either when the signal is minimal (left) or maximal (right). It appears that the left case is much better, since it avoids amplitude discontinuities (clicks).*

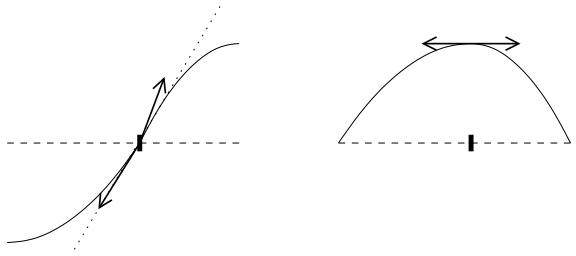


Figure 5: *Changing the frequency either when the signal is minimal (left) or maximal (right). It appears that the right case is better, since it avoids derivative discontinuities (clicks).*

Changing parameters of partials with the PASS method consists in changing the polynomial coefficients of an oscillator when this oscillator must be normally updated, because of the end of its validity period. Thus, this change does not need more computation time than without changing the parameters. The best case is with the validity period $p = 1/4$. In this case we can update the parameters at the best moment we described before for the frequency or the amplitude. Otherwise an update event will be added in the priority queue, to indicate the time to change the parameters of the partials. But few events will be added regarding to the normal updates of the coefficients, and it will not really affect the general computation time. The parameters are updated in the right moment to avoid clicks in the sound. This moment is different for every partials, regarding to their frequencies.

6. COMPLEXITY AND RESULTS

The complexity of the PASS method depends on the use of the priority queue. For one second of synthesis, the priority queue will be used f/p times per partial, where p is the validity period chosen ($1/2$ for a half of a sine period), and f the frequency of the

partial. For every partial, the queue operations are called X times per second, where:

$$X = \frac{1}{p}\sum_{i=1}^{N} f_i = \frac{N\bar{f}}{p} \tag{4}$$

with N being the number of partials in the sound, p the validity period, f_i the frequency of the partial i, and we denote by $\bar{f}$ the mean frequency of the partials.

If we consider that each priority queue operation is done with $O(\log(N))$ complexity for an update event, with N the number of elements simultaneously in the queue (*i.e.* the number of partials in the sound), the complexity C_1 due to the priority heap managing is given by:

$$C_1 = O\left(\frac{N\bar{f}}{p}\log N\right) \tag{5}$$

We can note that this complexity is very dependent from the mean frequency of the partials. It explains why the method is particularly efficient for low-frequency signals. If the validity period p is doubled (from a fourth to a half of period for example), the performance of the PASS method is doubled too. The higher is p and the better is the complexity of the method. But if we want to increase p, we need to use higher polynomial degree to approximate the sine function. And it leads to numerical instability. We have to find a trade-off between complexity and stability.

In addition to C_1 is the C_2 complexity, to produce the samples of the sound. If we use F_s as the sampling rate and d the polynomial degree of the generator, it is given by:

$$C_2 = O(dF_s) \tag{6}$$

Thus, the general complexity of PASS is:

$$C_{\text{PASS}} = O\left(\alpha\frac{N\bar{f}}{p}\log N + dF_s\right) \tag{7}$$

where α is some constant which is architecture-dependent (in practice, the synthesis methods were implemented in C language, compiled using the GNU C compiler (`gcc`) versions 4.0 and 4.1, and executed on PowerPC G4 1.25-GHz and Intel Pentium 4 1.8-GHz processors). This global complexity is a function of the sum of the frequencies of the partials, and we notice that it does not strongly depend on the sample rate anymore. Increasing the sample rate from 44.1 kHz to 96 kHz does not really affect the computation time, as illustrated by Table 2.

One might reasonably ask how our method compares to other methods for additive synthesis. Recently, we showed in [7] that the digital resonator method was a little better than the synthesis using the FFT^{-1} method. But later, Meine an Purnhagen [15] compared different methods and concluded that the fastest additive synthesis is now the FFT^{-1} method. In fact, this highly depends on the implementation details and computer used, as well as the number of partials and sampling rate. However, the digital resonator easily allows the fine control of each partial of the sound, which is not really the case the FFT^{-1} method. The PASS method allows the same fine control, and thus we have compared the performance of our method with that of the digital resonator method.

And if we have just noted that for PASS method the sample rate does not really affect the computation time, it is not the same with the digital resonator. The complexity C_{DR} of the digital resonator method is given by:

$$C_{\text{DR}} = O(NF_s) \tag{8}$$

with F_s the sampling rate, and N the number of partials in the sound. Here N and F_s are multiplied. This difference of complexity between digital resonator and PASS methods is illustrated by Table 2.

As shown in Table 3 or in Figure 6, the method we present is clearly better than the digital resonator for low frequencies. Using a 2-degree polynomial to approximate a half of a period of each partial, PASS is better for 2500 partials when the mean frequency of the partials is under 300 Hz, and even 500 Hz for a 96-kHz sampling rate. Real-time synthesis can be achieved with 5000 partials with a frequency of 150 Hz for example.

N	$\bar{f}$	F_s (Hz)	DR	PASS
4000	300	22050	**3.2** s	6.6 s
4000	300	44100	**6.3** s	6.6 s
4000	300	96000	13.7 s	**6.6** s

Table 2: *Comparison of the computation time of the Digital Resonator (DR) and PASS methods using different sampling rates, for 5 seconds of sound synthesis, implemented in C language, compiled using the GNU C compiler (`gcc`) version 4.1, and executed on an Intel Pentium 4 1.8-GHz processor. The PASS method is used with 2-degree polynomials and a validity period $p = 1/2$. N is the number of partials, $\bar{f}$ is the mean frequency of the partials, and F_s is the sampling rate.*

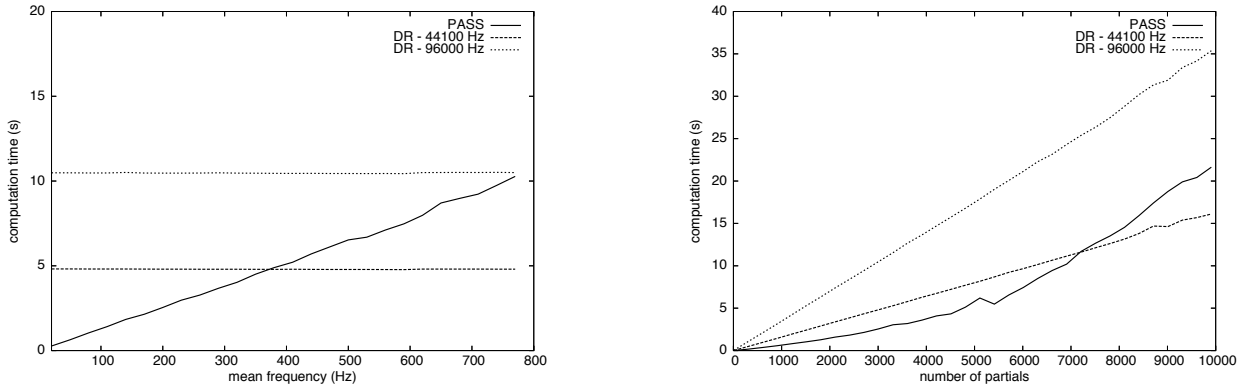
N	$\bar{f}$	DR	PASS
2500	200	3.9 s	**2.0** s
2500	300	3.9 s	**3.0** s
2500	400	**3.9** s	4.0 s
2500	500	**3.9** s	5.0 s
5000	200	7.9 s	**7.3** s
5000	300	**7.9** s	10.6 s
5000	400	**7.9** s	14.4 s

Table 3: *Comparison of the computation time of the Digital Resonator (DR) and PASS methods, for 5 seconds of sound synthesis with a sampling rate of 44100 Hz, implemented in C language, compiled using the GNU C compiler (`gcc`) version 4.1, and executed on an Intel Pentium 4 1.8-GHz processor. The PASS method is used with 2-degree polynomials and a validity period $p = 1/2$. N is the number of partials, $\bar{f}$ is the mean frequency of the partials.*

7. CONCLUSION AND FUTURE WORK

We have presented PASS, a new additive synthesis method using polynomials. The computation time of this method depends mainly on the sum of the frequencies of the partials. We have shown that the method is fast, and particularly efficient for signals with low frequency partials, as well as for high sampling rates.

In the near future, we plan to tune the trade-off between complexity and stability for our method on the fly, by combining the advantages of the PASS and Digital Resonator (DR) methods, since these methods both manipulate oscillators. For this hybrid method, the idea is, for a given partial, to use either PASS or DR depending



(a) (b)

Figure 6: *Comparison of the computation times of the Digital Resonator (DR) and PASS method, for 5 seconds of sound synthesis. (a) Computation times are functions of the mean frequency $\bar{f}$, with a fixed number of partials $N = 3000$. (b) Computation times are functions of the number of partials N, with a fixed mean frequency $\bar{f} = 200$ Hz. The PASS method is used with 2-degree polynomials and a validity period $p = 1/2$. For this comparison, both methods were implemented in C language, compiled using the GNU C compiler (*`gcc`*) version 4.0, and executed on a PowerPC G4 1.25-GHz processor.*

on the frequency of the partial. For low frequencies, PASS will be preferred. Also, the DR method will take advantage of the priority queue to schedule its optimal update times (see Section 5). At each update time, for the concerned oscillator the decision of switching from PASS to DR or from DR to PASS could be decided. And since at this update time, the amplitude, frequency, and also phase of the oscillator is known, the switch of method is really straightforward.

8. REFERENCES

[1] J. A. Moorer, "Signal processing aspects of computer music – a survey," *Computer Music J.*, vol. 1, no. 1, pp. 4–37, 1977.

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

[3] X. Serra and J. O. Smith, "Spectral Modeling Synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition," *Computer Music J.*, vol. 14, no. 4, pp. 12–24, 1990.

[4] J. W. Gordon and J. O. Smith, "A sine generation algorithm for VLSI applications," in *Proc. Int. Comp. Music Conf. (ICMC'85),* Burnaby, Canada, 1985, pp. 165–168.

[5] J. O. Smith and P. R. Cook, "The second-order digital waveguide oscillator," in *Proc. Int. Comp. Music Conf. (ICMC'92),* San Francisco, USA, 1992, pp. 150–153.

[6] S. Marchand and R. Strandh, "InSpect and ReSpect: Spectral modeling, analysis and real-time synthesis software tools for researchers and composers," in *Proc. Int. Comp. Music Conf. (ICMC'99),* Beijing, China, 1999, pp. 341–344.

[7] S. Marchand, "Sound models for computer music (analysis, transformation, and synthesis of musical sound)," Ph.D. dissertation, University of Bordeaux 1, France, 2000.

[8] M. Lagrange and S. Marchand, "Real-time additive synthesis of sound by taking advantage of psychoacoustics," in *Proc. COST-G6 Conf. on Digital Audio Effects (DAFx-01),* Limerick, Ireland, 2001, pp. 5–9.

[9] A. Freed, X. Rodet, and P. Depalle, "Synthesis and control of hundreds of sinusoidal partials on a desktop computer without custom hardware," in *Proc. ICSPAT*, 1992, pp. 98–101.

[10] Y. Ding and X. Qian, "Processing of musical tones using a combined quadratic polynomial-phase sinusoid and residual (QUASAR) signal model," *J. Audio Eng. Soc.*, vol. 45, no. 7/8, pp. 571–584, 1997.

[11] L. Girin, S. Marchand, J. di Martino, A. Röbel, and G. Peeters, "Comparing the order of a polynomial phase model for the synthesis of quasi-harmonic audio signals," in *Proc. IEEE Workshop Appl. of Dig. Sig. Proc. to Audio and Acoust.,* New Palz, NY, 2003, pp. 193–196.

[12] M. Raspaud, S. Marchand, and L. Girin, "A generalized polynomial and sinusoidal model for partial tracking and time stretching," in *Proc. Int. Conf. on Digital Audio Effects (DAFx-05),* Madrid, Spain, 2005, pp. 24–29.

[13] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, ser. Series in Computer Science and Information Processing. Addison-Wesley, 1983, pp. 392–407.

[14] R. Strandh and S. Marchand, "Real-time generation of sound from parameters of additive synthesis," in *Proc. Journées d'Informatique Musicale*, 1999, pp. 83–88.

[15] N. Meine and H. Purnhagen, "Fast sinusoid synthesis for MPEG-4 HILN parametric audio decoding," in *Proc. Int. Conf. on Digital Audio Effects (DAFx-02),* Hamburg, Germany, 2002, pp. 239–244.