Fast Time-Domain Volterra Filtering

Harald Enzinger*, Karl Freiberger*, Gernot Kubin* and Christian Vogel*†
*Signal Processing and Speech Communication Laboratory, Graz University of Technology, Austria
†FH Joanneum - University of Applied Sciences, Austria
Email: enzinger@tugraz.at, freiberger@tugraz.at, g.kubin@ieee.org, c.vogel@ieee.org

***Abstract*—We present two algorithms for fast time-domain Volterra filtering. The first algorithm computes the Volterra series input products using only one multiplication per input product. Since the input products are explicitly computed, this algorithm can be used for adaptation as well as for filtering. The second algorithm generalizes Horner's method for polynomial evaluation and directly computes output samples without computing input products. This way, time-domain Volterra filtering can be implemented with only one multiplication per parameter. We implemented several variants of the proposed algorithms in C and compared their runtime, demonstrating a speedup up to 5.**

I. Introduction

The Volterra filter is a widely used structure for nonlinear filtering problems with applications like digital enhancement of analog circuits [1], nonlinear acoustic echo cancellation [2] or distortion mitigation of digital pulse width modulation [3].

A discrete-time Volterra filter is given by

$$y[n] = \sum_{p=1}^{P} \sum_{m_1=0}^{M} \cdots \sum_{m_p=0}^{M} h_p[m_1, \ldots, m_p] \prod_{l=1}^{p} x[n - m_l] \quad (1)$$

where $x[n]$ is the input signal, $y[n]$ is the output signal, $h_p[m_1, \ldots, m_p]$ is the p-th order kernel, P is the maximum order of nonlinearity and M is the memory depth.

For $P = 1$ and $M > 0$ the Volterra filter reduces to a linear finite impulse response filter with $M + 1$ filter taps and an impulse response given by $h_1[m_1]$. For $P > 1$ and $M = 0$ it reduces to a polynomial of order P with coefficients given by $c_p = h_p[0, \ldots, 0]$ with $p = 1, \ldots, P$. For $P > 1$ and $M > 0$ it represents a combination of a linear filter and a polynomial.

A major obstacle for the practical application of Volterra filters is their high computational complexity. The complexity of a Volterra filter can be quantified by the number of parameters in the kernel and the number of operations required for each parameter. In (1), each p-th order kernel $h_p[m_1, \ldots, m_p]$ represents a p-dimensional hypercube with $(M + 1)^p$ parameters and direct evaluation of (1) requires p multiplications and one accumulation for each parameter.

To reduce the computational complexity of Volterra filtering, one can either reduce the number of parameters, or the number of operations per parameter. In the remaining part of this section, we give an overview of techniques for complexity reduction, followed by an outline of our contributions.

The research leading to these results has received funding from the FFG Competence Headquarter program under the project number 4718971.

A well known method [4] to reduce the number of parameters follows from the commutativity of multiplication. Since all permutations of the time-lags m_1 to m_p correspond to the same product of input samples, we can restrict the sum in (1) to a specific order of the index variables like $m_1 \leq m_2 \leq \ldots \leq m_p$, without loss of generality. This reduces the number of parameters in h_p from $(M + 1)^p$ to

$$\left(\!\binom{M+1}{p}\!\right) = \binom{M+p}{p} = \frac{(M+p)!}{M!\,p!} \quad (2)$$

where $\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k}$ is the number of k-multicombinations out of n objects and $\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$ is the binomial coefficient. A further reduction of parameters can be achieved by sparse kernels [5], banded kernels [6], approximation based on matrix factorization [7] or the use of multirate techniques [8]. In the present work, however, we focus on the full Volterra model and use all non-redundant parameters in $h_p[m_1, \ldots, m_p]$.

A reduction of the number of operations can be achieved either by time-domain or by frequency-domain techniques. In the time-domain, the number of operations can be reduced by re-use of previously computed input products like computing $x^3[n]$ by multiplying $x^2[n]$ and $x[n]$. This way the input products can be computed using only one multiplication per parameter. For the output sample, one additional multiplication per parameter is required, which results in a total number of two multiplications per parameter. The computation of output samples using only one multiplication per parameter can be achieved with the pipelined structure proposed in [9]. By the use of frequency-domain methods [10]–[12] which utilize the fast Fourier transform, even less than one multiplication per parameter can be achieved, but only for large memory depths. For small memory depths, time-domain computation is faster.

A. Contributions

Reuse — An algorithm which computes Volterra series input products using one multiplication per input product.

Horner — An algorithm which computes Volterra series output samples using one multiplication per parameter.

The second algorithm is based on a factorization which was proposed in [9] to derive an efficient hardware structure. In the present paper, we use this factorization to derive an efficient algorithm and we compare different options for its implementation. We also highlight that this factorization generalizes Horner's method for polynomial evaluation.

Algorithm 1 Nested loop traversing of $\{i, j, m_1, \ldots, m_p\}$

```
initialize i and j
for m_1 from 0 up to M do
    for m_2 from m_1 up to M do
        ⋱
            for m_{p-1} from m_{p-2} up to M do
                for m_p from m_{p-1} up to M do
                    work with {i, j, m_1, ..., m_p}
                    increment i
                end for
                increment j
            end for
        ⋰
    end for
end for
```

Algorithm 2 Combinatoric traversing of $\{i, j, m_1, \ldots, m_p\}$

```
initialize i and j, set {m_1, ..., m_p} to 0
start:
work with {i, j, m_1, ..., m_p}
increment i
if m_p < M then
    increment m_p, go to start
end if
increment j
for i from p − 1 down to 1 do
    if m_i < M then
        increment m_i
        set {m_{i+1}, ..., m_p} to m_i, go to start
    end if
end for
```

II. Methods of Traversing

A fundamental requirement for the implementation of Volterra filters is the ability to traverse the set of non-redundant input products and their corresponding kernel parameters. This means we need an algorithm which visits every required combination of time-lags m_1 to m_p exactly once. The result of such an algorithm can be represented as a list like in Table I.

Table I contains the order p, the kernel index i, the reuse or projection index j and the time-lags m_1 to m_p. The kernel index i is incremented at each iteration and is used to address the kernel parameters in the one-dimensional representation of the Volterra kernel which is given by

$$h[i] = h_p[m_1, \ldots, m_p]\,. \tag{3}$$

The reuse or projection index j is incremented only if there is a change in the time-lags m_1 to m_{p-1} and will be used by the fast algorithms in the next section. For each order p, the time-lags m_1 to m_p represent p-multicombinations out of the $M+1$ numbers $\{0, \ldots, M\}$ in lexicographic order.

To generate Table I, one of the two algorithms on top of this page can be used. Algorithm 1 generates the values $\{i, j, m_1, \ldots, m_p\}$ for a given order p by use of a nested for-loop. This represents a direct implementation of (1) with the constraint $m_1 \leq m_2 \leq \ldots \leq m_p$ being imposed by the start indices of the nested for-loops. Algorithm 2 generates the same values by conditional increment and set operations of the variables m_1 to m_p. An advantage of Algorithm 2 is that it can be used for arbitrary orders p, whereas Algorithm 1 must be explicitly coded for each order p. However, the increased generality of Algorithm 2 comes at the price of a higher runtime which will be shown in Section IV.

The time-lags of Table I can also be visualized by the geometric representation in Fig. 1. The first order time-lags are represented by green circles along the axis m_1, the second order time-lags are represented by blue circles in the surface spanned by axes m_1 and m_2 and the third order time-lags are represented by red circles in the area spanned by axes m_1, m_2 and m_3. The numbers in Fig. 1 indicate the kernel index i.

TABLE I
Traversal Trace for $P = 3$ and $M = 2$.

p	i	j	m_1	m_2	m_3
1	0	-	0	-	-
	1	-	1	-	-
	2	-	2	-	-
2	3	0	0	0	-
	4	0	0	1	-
	5	0	0	2	-
	6	1	1	1	-
	7	1	1	2	-
	8	2	2	2	-
3	9	3	0	0	0
	10	3	0	0	1
	11	3	0	0	2
	12	4	0	1	1
	13	4	0	1	2
	14	5	0	2	2
	15	6	1	1	1
	16	6	1	1	2
	17	7	1	2	2
	18	8	2	2	2

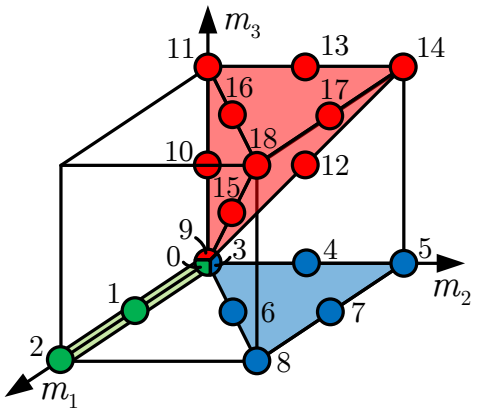

Fig. 1. Geometric representation of Table I.

III. Methods of Computation

A. Direct Computation

Based on the iteration through $\{p, i, m_1, \ldots, m_p\}$, output samples of the Volterra filter can be computed by

$$y[n] = \sum_{i=0}^{i_{\max}} h[i]\, \Phi_i[n] \tag{4}$$

where $i_{\max}$ is the highest kernel index and $\Phi_i[n]$ is the i-th input product consisting of p terms given by

$$\Phi_i[n] = x[n - m_1] \cdots x[n - m_p]\,. \tag{5}$$

Volterra filtering based on (4)-(5) requires $p-1$ multiplications for the input product, one multiplication for the weighting by the kernel and one accumulation per kernel parameter.

B. Fast Computation of Input Products

The number of operations for computing the input products of order $p > 1$ can be reduced to one multiplication, if the reuse index j and the highest order time-lag m_p are utilized to represent the i-th input product by

$$\Phi_i[n] = \Phi_j[n]\, x[n - m_p]\,. \tag{6}$$

A further reduction of multiplications is possible, if input products with $m_1 > 0$ are generated from input products of the same order with $m_1 = 0$ by delay operations. However, this requires additional storage for previous input products and introduces overhead for the selection of multiplication or delay. Therefore, it generally gives no runtime advantage.

C. Fast Computation of Output Samples

The fast computation of output samples is based on a factorization of the Volterra series which was initially proposed in [9]. It is obtained by subsequently pulling out the sum over m_l and the term $x[n - m_l]$ for $l = 1, \ldots, p$ and representing the remaining part as a time-varying kernel[1]. This results in

$$y[n] = \sum_{m_1=0}^{M} x[n - m_1]\, g_1[n, m_1] \tag{7}$$

with the first-order time-varying kernel given by

$$g_1[n, m_1] = h_1[m_1] + \sum_{m_2=m_1}^{M} x[n - m_2]\, g_2[n, m_1, m_2] \tag{8}$$

and the higher-order time-varying kernels given by

$$\begin{aligned} g_p[n, m_1, \ldots, m_p] &= h_p[m_1, \ldots, m_p] \\ &+ \sum_{m_{p+1}=m_p}^{M} x[n - m_{p+1}]\, g_{p+1}[n, m_1, \ldots, m_{p+1}]\,, \end{aligned} \tag{9}$$

up to the highest order P where the kernel is time-invariant and given by $g_P[n, m_1, \ldots, m_P] = h_P[m_1, \ldots, m_P]$.

[1]This representation of the Volterra series can also be used for the linearization of time-varying nonlinear systems as presented in [13].

TABLE II
Fast Computation of an Output Sample for $P = 3$ and $M = 2$.

p	j	m	i	Operation
3	3	0	9	$g[3] \leftarrow h[3] + x[n]\, h[9]$
	3	1	10	$g[3] \mathrel{+}= x[n-1]\, h[10]$
	3	2	11	$g[3] \mathrel{+}= x[n-2]\, h[11]$
	4	1	12	$g[4] \leftarrow h[4] + x[n-1]\, h[12]$
	4	2	13	$g[4] \mathrel{+}= x[n-2]\, h[13]$
	5	2	14	$g[5] \leftarrow h[5] + x[n-2]\, h[14]$
	6	1	15	$g[6] \leftarrow h[6] + x[n-1]\, h[15]$
	6	2	16	$g[6] \mathrel{+}= x[n-2]\, h[16]$
	7	2	17	$g[7] \leftarrow h[7] + x[n-2]\, h[17]$
	8	2	18	$g[8] \leftarrow h[8] + x[n-2]\, h[18]$
2	0	0	3	$g[0] \leftarrow h[0] + x[n]\, g[3]$
	0	1	4	$g[0] \mathrel{+}= x[n-1]\, g[4]$
	0	2	5	$g[0] \mathrel{+}= x[n-2]\, g[5]$
	1	1	6	$g[1] \leftarrow h[1] + x[n-1]\, g[6]$
	1	2	7	$g[1] \mathrel{+}= x[n-2]\, g[7]$
	2	2	8	$g[2] \leftarrow h[2] + x[n-2]\, g[8]$
1	-	0	0	$y[n] \leftarrow x[n]\, g[0]$
	-	1	1	$y[n] \mathrel{+}= x[n-1]\, g[1]$
	-	2	2	$y[n] \mathrel{+}= x[n-2]\, g[2]$

To convert (7)-(9) to an efficient algorithm, we represent the kernels in one-dimensional form and evaluate the time-varying kernel from its highest order down to one. The one-dimensional representation of the time-varying kernel is

$$g[i] = g_p[n, m_1, \ldots, m_p]\,. \tag{10}$$

The computation of the time-varying kernel of order p is given by (9), which is represented in one-dimensional form by

$$g[j] = h[j] + \sum_{m,i} x[n - m]\, g[i] \tag{11}$$

where j is a one-dimensional kernel index which corresponds to the time-lag $\{m_1, \ldots, m_p\}$, m ranges from m_p to M and i is a one-dimensional kernel index which ranges from the time-lag $\{m_1, \ldots, m_p, m_p\}$ to the time-lag $\{m_1, \ldots, m_p, M\}$.

An example for the fast computation of an output sample is given in Table II, where p, i, j and m are taken from Table I with m being the highest-order time-lag m_p in Table I. The first three rows of Table II implement (11) for $j = 3$ using one multiply-add operation and two multiply-accumulate operations. The following two rows implement (11) for $j = 4$ using one multiply-add and one multiply-accumulate operation. This process goes on until the first order is reached, where the output sample is computed by a linear convolution of the input signal with the first-order time-varying kernel.

An intuitive understanding of the operations in Table II can be obtained by comparison with Fig. 1. The operations where $p = 3$ can be interpreted as a marginalization of dimension m_3, where the red area which corresponds to the third order kernel is projected onto the blue surface which corresponds to the second order kernel. In a similar way, the operations where $p = 2$ can be interpreted as a marginalization of dimension m_2, where the blue surface is projected onto the green line.

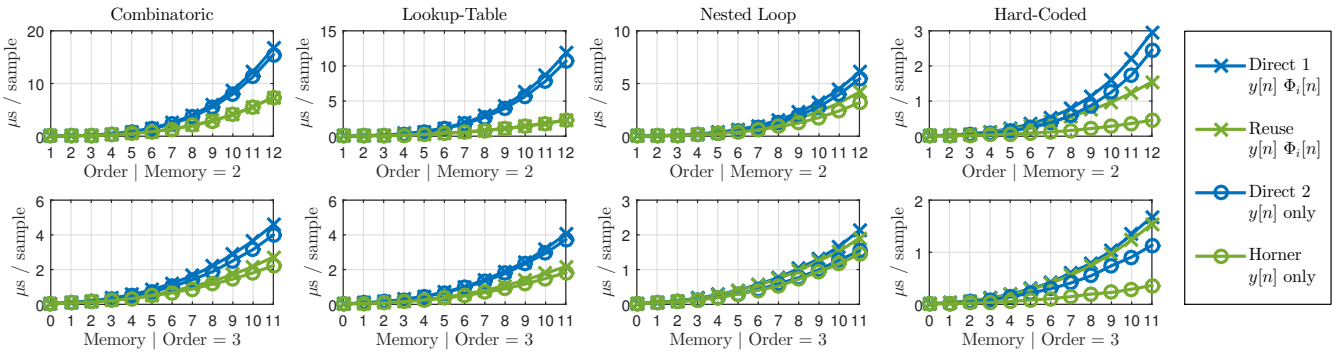

Fig. 2. Comparison of the runtime of different implementations of time-domain Volterra filtering. The first row of plots shows a sweep of the filter order $P = 1, \ldots, 12$ with the memory depth kept constant at $M = 2$. The second row of plots shows a sweep of the memory depth $M = 0, \ldots, 11$ with the filter order kept constant at $P = 3$. Both sweeps cover the same range of number of parameters which is $\{3, 9, 19, 34, 55, 83, 119, 164, 219, 285, 363, 454\}$. The methods of traversing are compared by columns of plots, ordered from slow (left) to fast (right). The methods of computation are compared within the plots.

IV. Evaluation of Runtime

To evaluate the proposed methods we implemented them in the programming language C and compared their runtime on a personal computer. In total we produced 16 implementations resulting from the combination of 4 methods of traversing with 4 methods of computation. The source code is available [14].

The methods of traversing are the nested loop traversing, the combinatoric traversing, the lookup-table traversing and the hard-coded traversing. The first two methods where presented in Section II, the other two methods are techniques to speed up the filtering by computing a traversal trace like in Table I before the filter operation. At the lookup-table traversing, the traversal trace is stored in memory, whereas at the hard-coded traversing, the traversal trace is included in the program code. The latter requires metaprogramming where the algorithm of traversal generates the source code for the filter operation.

The methods of computation are two variants of the direct computation from Section III-A and the fast methods from Section III-B (*Reuse*) and Section III-C (*Horner*). The variant *Direct 1* stores input products during the computation of an output sample and is used for comparison with the method *Reuse*. The variant *Direct 2* does not store input products and is used for comparison with the method *Horner*.

To evaluate the runtime, we used the function `clock()` from the library `time.h`. An overview of the measured runtimes is shown in Fig. 2. It can be seen that the runtime strongly depends on the method of traversing. The most generic method, the combinatoric traversing, shows the highest runtime. By use of the lookup-table traversing, the runtime can be reduced at the cost of storage for the traversal trace. The nested loop traversal is faster than the previous methods, but it is limited to a maximum filter order, since each order requires one nested for-loop. Finally the hard-coded traversing is the fastest, but also the least flexible since the source code for a given memory depth and maximum filter order must be generated in advance. By comparing the methods of computation, one can see that *Reuse* is always faster than *Direct 1* and *Horner* is always faster than *Direct 2*.

V. Conclusion

We presented several methods for fast time-domain Volterra filtering and compared their runtime. Unlike frequency-domain methods, the presented time-domain methods are always faster than direct computation. In future work the presented techniques may also be applied to baseband Volterra filters [15].

References

[1] B. Murmann, C. Vogel, and H. Koeppl, "Digitally enhanced analog circuits: System aspects," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2008, pp. 560–563.

[2] A. Guérin, G. Faucon, and R. L. Bouquin-Jeannès, "Nonlinear acoustic echo cancellation based on Volterra filters," *IEEE Transactions on Speech and Audio Processing*, vol. 11, no. 6, pp. 672–683, 2003.

[3] F. Chierchie and S. O. Aase, "Volterra models for digital PWM and their inverses," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 10, pp. 2606–2616, 2015.

[4] W. J. Rugh, *Nonlinear system theory*. Johns Hopkins Univ. Press, 1981.

[5] V. Kekatos and G. B. Giannakis, "Sparse Volterra and polynomial regression models: Recoverability and estimation," *IEEE Transactions on Signal Processing*, vol. 59, no. 12, pp. 5907–5920, 2011.

[6] G. O. A. Glentis, P. Koukoulas, and N. Kalouptsidis, "Efficient algorithms for Volterra system identification," *IEEE Transactions on Signal Processing*, vol. 47, no. 11, pp. 3042–3057, 1999.

[7] T. M. Panicker and V. J. Mathews, "Parallel-cascade realizations and approximations of truncated Volterra systems," *IEEE Transactions on Signal Processing*, vol. 46, no. 10, pp. 2829–2832, 1998.

[8] D. Schwingshackl and G. Kubin, "Polyphase representation of multirate nonlinear filters and its applications," *IEEE Transactions on Signal Processing*, vol. 55, no. 5, pp. 2145–2157, 2007.

[9] M. M. Banat, "Pipelined Volterra filter," *Electronics Letters*, vol. 28, no. 13, pp. 1276–1278, 1992.

[10] M. Morhac, "A fast algorithm of nonlinear Volterra filtering," *IEEE Transactions on Signal Processing*, vol. 39, no. 10, pp. 2353–2356, 1991.

[11] M. Reed and M. Hawksford, "Efficient implementation of the Volterra filter," *IEE - Vision, Image and Signal Processing*, vol. 147, no. 2, pp. 109–114, 2000.

[12] R. Bernardini, "A fast algorithm for general Volterra filtering," *IEEE Transactions on Communications*, vol. 48, no. 11, pp. 1853–1864, 2000.

[13] M. Hotz and C. Vogel, "Linearization of time-varying nonlinear systems using a modified linear iterative method," *IEEE Transactions on Signal Processing*, vol. 62, no. 10, pp. 2566–2579, 2014.

[14] H. Enzinger, "Fast time-domain Volterra filtering implemented in C," www.spsc.tugraz.at/tools/fast-time-domain-volterra-filtering, oct 2016.

[15] H. Enzinger, K. Freiberger, G. Kubin, and C. Vogel, "Baseband Volterra filters with even order terms: Theoretical foundation and practical implications," in *Asilomar Conference on Signals, Systems and Computers*, 2016.