PARALLEL RECURSIVE LEAST SQUARES ALGORITHMS FOR ADAPTIVE VOLTERRA FILTERS

Ajit K. Chaturvedi and Govind Sharma

Department of Electrical Engineering
Indian Institute of Technology
Kanpur – 208016 INDIA

ABSTRACT – The paper presents a parallel but approximate version of the exact recursive least squares algorithm. This algorithm, which we call the Parallel Recursive Least Squares (PRLS) algorithm has been applied to adaptive Volterra filters. It has advantages of reduced cost per iteration and substantial reduction in computational time per iteration if more than one processor is used. The algorithm has the potential of providing a flexible trade off in terms of rate of convergence and computational cost.

INTRODUCTION

The optimum filter in any given situation is generally nonlinear. The Volterra filter, due to its ability to represent a large subset of the nonlinearities possible, has immense potential in the realization of nonlinear filters. Adaptive discrete Volterra filters are useful in areas like nonlinear system modeling and identification,channel equalization, adaptive echo and noise cancellation etc. . Previous contributions for e.g. [1 –3] have succeeded in enhancing our understanding of adaptive Volterra filters. However it appears that we still have to cover some distance before these adaptive filters can become tools of common use.
In [4] we have derived a Recursive Least Squares (RLS) algorithm and its fast version (FRLS) both of which are applicable to Volterra filters of any degree.and have rapid convergence compared to LMS algorithms.
In this paper the approach we have followed in [4] is carried forward. However [4] is not a prerequisite to understand this paper The main contribution of this paper lies in giving a parallel version of the RLS algorithm for Volterra filters of any degree. *The Parallel RLS algorithm (PRLS) provides a high degree of parallelism,due to which the time per iteration can be reduced drastically and it becomes possible to operate Volterra filters in real time, if we are allowed to have more than one processor.* The time per iteration and the degree of parallelism possible depend upon the filter structure under consideration. *The computational cost per iteration of the fast version of PRLS i.e. FPRLS is less than that of the FRLS algorithm derived in [4], independent of whether we exploit the parallelism or not.* Besides, unlike FRLS there is no need to invert any matrix or create rescue devices in the PRLS. Simulations have shown that the rate of convergence of PRLS is much better than the LMS although not as good as that of RLS or FRLS of [4].
Another useful virtue of PRLS comes out when we apply it to *linear filters*. Apart from providing us with an algorithm, *the cost of which is the least of all the available algorithms of the RLS family,* it provides us with a sort of 'bridge'(valid for the nonlinear case also) which connects the LMS to the RLS algorithm. *This 'bridge' (explained later) has practical utility and gives an insight which improves our current understanding of the relationship between the LMS and the RLS family of algorithms.* In the sequel lower case letters have been used for scalars, upper case letters for vectors and bold upper case for matrices

THE PRLS ALGORITHM

The Volterra filter is defined as :

$$d(n) = h_0 + \sum_{j=0}^{p} a(j)x(n-j) + \sum_{j,k=0}^{q}\sum^{r} b(j,k)x(n-j)x(n-k) + \ldots \quad \ldots(1)$$

where d(n) is the desired output sequence and x(n) is the input sequence. h_0 , a(j), b(j,k) etc. are the coefficients which have to be determined.

(1) can be written as $d(n) = W^T X(n)$. Here W is the Nx1 coefficient vector and X(n) is the Nx1 input vector where N is the order of the filter.

$$X^T(n) = [1, x(n),\ldots,x(n-p), x^2(n),\ldots,x(n-q)x(n-r),\ldots\ldots]$$

$$W^T = [h_0, a(0),\ldots,a(p), b(0,0),\ldots,b(q,r),\ldots\ldots]$$

In X(n) above p is the order of the first degree , (q,r) is the order of the second degree and so on. Within any particular degree, terms which are permutations of each other are clubbed together. To find the coefficient vector W we follow the usual approach of minimising the sum of squared errors given by

$$\epsilon(n) = \sum_{i=0}^{n} \lambda^{n-i} e^2(i) \qquad [\text{ Here } e(i) = d(i) - y(i) \text{ }]$$

Here d(i) is the desired sequence and λ is the forgetting factor to account for nonstationarity. The optimum value of W(n) is given by the normal equation:

$$\mathbf{R}(n)W(n) = P_1(n) \qquad (2)$$

$$\text{where } \mathbf{R}(n) = \lambda\, \mathbf{R}(n-1) + X(n)X^T(n) \qquad (3)$$
$$P_1(n) = \lambda\, P_1(n-1) + d(n)X(n)$$

W(n) should approach W as n increases (for a useful algorithm).

We first partition X(n) into different kinds of nonlinearities. Two nonlinear terms are defined to be of

the same kind if they differ only by some delay. Since $x^2(n)$ and $x(n)x(n-1)$ are not time delayed versions of each other we treat them as different kind of nonlinearities even though both are of the second degree.
Now assume $d(n) = d_1(n) + d_2(n) + d_3(n) + \ldots + d_r(n)$ where r is the total number of different kinds of nonlinearities in the filter structure under consideration. $d_i(n)$ represents that fraction of the desired signal d(n) that comes from the ith kind of nonlinearity. In a given situation we are given only d(n) and we do not know the different $d_i(n)$. Assume for the time being that we know the different $d_i(n)$. Then we can implement the nonlinear filter as r parallel filters each operating independent of all the others. It will be equivalent to saying that we have broken an Nth order filter into r smaller filters each operating on the same input sequence but using different desired sequences. Each will have its own error sequence and weight vectors. Because only the input sequence is common the filters will operate independent of each other. In this way not only the filters can be operated in parallel but the overall computational cost will also come down as now we will be operating on smaller order filters compared to a single filter of higher order. It is worthwhile to note that the use of $d_i(n)$ lies in computing $e_i(n)$ [the error of the ith filter],which in turn is used to update the weight vectors of the ith filter. If in place of $d_i(n)$ we are given $e_i(n)$ it will as well serve our purpose.
To circumvent the problem of not knowing $d_i(n)$ or $e_i(n)$ we do the following:
Treat all the r filters as separate entities. The output of the overall filter is computed by summing the outputs of all the individual filters. Subtract this from d(n) to compute e(n). Now update all the r filters using this e(n) and their respective input and weight vectors. Instead of running the filters with their $e_i(n)$ we are running all of them with e(n). We have to make this approximation due to our ignorance of both the $d_i(n)$s and the $e_i(n)$s.
In effect this implies that each filter assumes all the other filters (except itself) to have converged to their true values.
Mathematically, this approximation implies that the following equality holds :

$$d_i(n) = d(n) - \sum_{\substack{j=1 \\ j\neq i}}^{r} W_j^T(n)X_j(n) \qquad 1 \leq i \leq r$$

The above equality is true ONLY when $W_j(n)$ [weight vector of the jth filter] for all j (except i) have converged to their true values. Thus for each of the r filters the approximation made is different although essentially it is of the same nature
It appears paradoxical that when we are updating the ith filter at time n we are assuming that all the other filters have converged to their true values at time (n–1) and when we are updating the jth filter (j≠i) we are assuming all filters (excluding j) including the ith filter have converged to their true values at time (n–1). However, a little reflection shows that it is nothing more than a convenient and useful approximation.
The error due to the approximation can be modeled as correlated noise which is added to each $e_i(n)$ to produce e(n) and hence instead of $e_i(n)$ we are using e(n).
Another way to look at this approximation is that since we are ignoring the couplings between the parallel filters by treating them as separate entities, we are sort of compensating it by using e(n) which not only depends on the output of all the filters but also on d(n), the sequence which the overall filter is trying to produce.
As a simple example let there be a Volterra filter defined by the following relation :

$$d(n) = b(0,0)x^2(n) + b(1,1)x^2(n-1) + b(0,1)x(n)x(n-1) + b(1,2)x(n-1)x(n-2)$$

According to our definition, this consists of two kinds of nonlinearities. So we will operate two filters in parallel. The output of the PRLS filter is the sum of the outputs of the two filters. The PRLS for the case when p filters are operating in parallel is as follows:
Initialize $W_i(n)$ and $P_i(n)$ for i=1,p

$$\alpha(n) = d(n) - \sum_{j=1}^{p} W_j^T(n)X_j(n)$$

Do for i=1,p

$$k_i(n) = \lambda^{-1}P_i(n-1)X_i(n)\,[1+ \lambda^{-1}X_i^T(n)P_i(n-1)X_i(n)]^{-1}$$

$$W_i(n) = W_i(n-1) + k_i(n)\alpha(n)$$

$$P_i(n) = \lambda^{-1}P_i(n-1) - \lambda^{-1}k_i(n)X_i^T(n)P_i(n-1)$$

where α denotes the a priori error while W,P,X,k and λ have the usual meaning.
Exploiting the serial shifting of data, inherent in each of the filters because of the way they are defined, by using the Fast RLS algorithm derived for linear filters in [5] we have derived the fast version of PRLS i.e. FPRLS. It is quite straightforward and we are not giving the algorithm here.
The cost of FPRLS is O(N) while that of FRLS is O(rN) where r is the number of nonlinearities and the cost of PRLS varies from O(N) to O(N^2) depending on the configuration chosen (we are concerned with the part of the processing required after the input vectors have been procured).If we exploit the parallelism the time per iteration can be reduced to α[O(m)] where m is the length of the longest filter amongst all the filters that are operating in parallel and α is the time taken for one operation.
We have run PRLS and FPRLS for a large variety of Volterra filters. In all of the nearly fifty examples done they converged.
The rate of convergence and the point around which the algorithm walks after it has stabilised are functions of the filter structure. As an example we have given the $\log_{10}[e^2(n)]$ vs. the no. of iterations of PRLS in Fig. 1 for the case when

$$\begin{aligned} d(n) = \; & b(0,1)x(n)x(n-1) + b(1,2)x(n-1)x(n-2) + \\ & b(2,3)x(n-2)x(n-3) + d(0,0,0,0)x^4(n) + \\ & d(1,1,1,1)x^4(n-1) + d(2,2,2,2)x^4(n-2) \end{aligned}$$

[b and d denote the coefficients]
However no example of PRLS should be treated as representative since the convergence and steady state error are dependent on the filter structure.

LINEAR FILTERS

Apart from the advantages of parallelism, reduced time per iteration and the lowest computational cost amongst the family of recursive algorithms for linear filters (when PRLS is used for linear filters); a useful feature of PRLS can be explained using the linear filters as an example. If we have a linear filter of order N then, following the method explained above we can operate it as N parallel filters each of length 1. We can as well allow the filters to be of length 2. In that case we will have only N/2 parallel filters. Thus depending on the value of N we have a number of different possibilities in which in which the filter can be configured.

In terms of cost and convergence there is a steep difference between LMS and RLS. However PRLS provides with a continuous gradation from one end to the other. If the filter order is 4, PRLS gives rise to 15 possible configurations. If we use 4 filters in parallel the behaviour of the algorithm is nearer to LMS (it can be shown using the update equations) while if we use only 1 filter PRLS is the same as RLS. In between as the number of parallel filters come down the cost rises while the convergence improves. Thus there is a step by step increase in the cost and improvement in the performance as we go from one end to the other. *This is why it can be said to have provided a 'bridge' from the LMS to RLS.* Depending on the requirement and the affordable cost one can chose the appropriate configuration of PRLS. If LMS is not suitable there is no need to jump to RLS directly. When chosing a configuration of PRLS one should take care to club together those terms, in a particular branch of the filter, which have significant correlation amongst themselves. This will help in optimising the cost and convergence together. In fact if we do not chose the terms in a branch according to this criterion we may get a situation in which the performance of a configuration with higher cost is inferior to that of a configuration with lower cost. Obviously this is undesirable.

CONCLUSION

Even though the fraction of the desired signal that is coming from a particular branch of a Volterra filter is not known but by making an approximation in the error signal we have been able to simplify considerably the adaptation of an adaptive Volterra filter. This has provided significant advantages in cost with a little sacrifice in performance.

REFERENCES

[1] Koh T. and Powers E. J.,"Second Order Volterra Filtering and its Application to Nonlinear System Identification", *IEEE Trans. ASSP* –33 No.6 Dec.'85 pp. 1445–54.

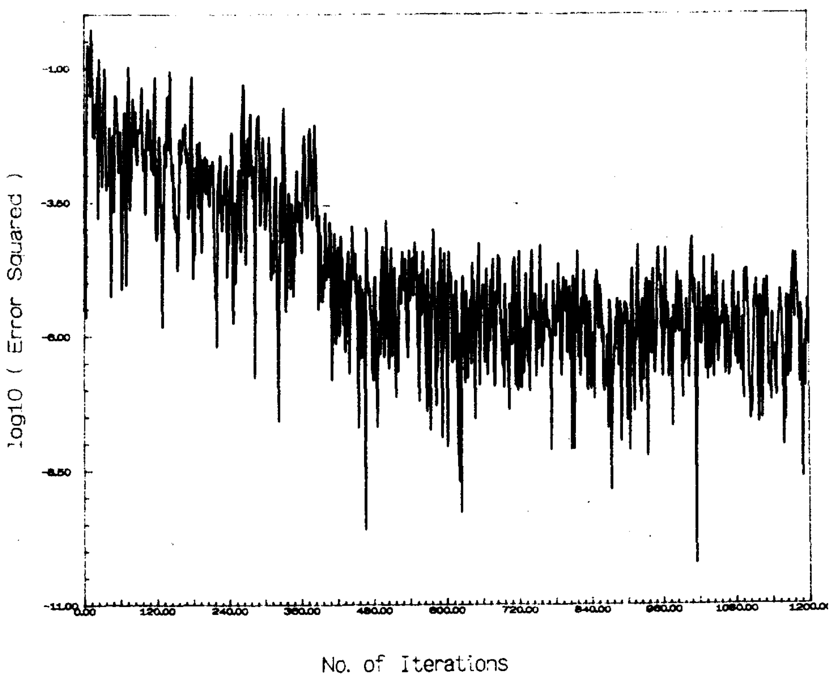
[2] Davila C. E. et al,"A Second Order Adaptive Volterra Filter with Rapid Convergence", *IEEE Trans. ASSP* –35 no.9 Sept.'87 pp. 1259–63.

[3] Duvaut P.,"A Unifying and General Approach To Adaptive Linear –Quadratic Discrete Time Volterra Filtering", *Proc. IEEE ICASSP* 1989 pp. 1166–70.

[4] Chaturvedi Ajit K. and Sharma Govind ,"RLS and FAST RLS Algorithms for Adaptive Nonlinear Volterra Filters of any Degree", *Proc. International Conference on Information and Systems,* held at HANGZHOU, CHINA during Oct. 9 – 11 ,1991.

[5] Ljung L., Morf M. and Falconer D.,"Fast Calculation of Gain Matrices for Recursive Estimation Schemes", *Int J. Control*, Vol. 27,No. 1, Jan.'78 pp. 1–19.

PRLS (Fig. 1)


log10 (Error Squared)
No. of Iterations