On the Methods for Solving Yule–Walker Equations

Hui-Min Zhang and Pierre Duhamel, *Senior Member, IEEE*

***Abstract*—We review the three well-known fast algorithms for the solution of Yule–Walker (YW) equations: the Levinson, Euclidean, and Berlekamp–Massey algorithms, and show the relation between each of them and the Padé approximation problem. This connection has already been noticed for some of them, but we intend here to offer a synthetic view of these fast algorithms.**

We classify the algorithms solving YW equations with reference to three criteria, namely: 1) the path they follow in the Padé table; 2) the organization of the computation: we distinguish between one-pass and two-pass algorithms; and 3) the auxiliary variables used: some algorithms use the backward predictor of same degree as intermediate variable for computing the forward predictor (or vice versa), while others use two predictors of the same type but of successive degrees.

This classification shows that the set of known classical algorithms is not complete, and we propose the missing variants. With these variants of the Berlekamp–Massey and Euclid algorithms, we are able to obtain both forward and backward predictors without additional cost.

Furthermore, we give a unified representation of the two-pass algorithms, in such a way that the application of the divide and conquer strategy becomes straightforward. A general doubling algorithm which represents all the associated doubling algorithms in an exhaustive way is provided.

I. Introduction

SPEEDING up the solution of Yule–Walker (YW) equations is very important, since this problem appears frequently in various fields such as error-correcting codes, and digital signal processing, to name only a few. For example, the key equation for decoding Bose–Chaudhuri–Hocquenghem (BCH)/Reed–Solomon (RS)/Goppa codes [4], and the equation involved in the identification of coefficients of an autoregressive model, are Yule–Walker equations. The Levinson algorithm, Berlekamp–Massey algorithm, and Euclidean algorithm are the three well-known fast algorithms for this purpose, each of them being linked initially with one special application, although their usefulness has been recognized for the other ones later on.

By exploring the Toeplitz structure of the coefficient matrix, the Levinson algorithm [18] solves a Yule–Walker equation with $O(n^2)$ arithmetic operations, where n is the size of the matrix. The Berlekamp–Massey algorithm was developed by Berlekamp [4] for solving precisely the key equation for decoding BCH codes, and was interpreted by Massey as a linear feedback shift-register synthesis having the shortest length [19]. The relation between a Yule–Walker equation and a Padé approximation [6] makes also possible the Euclidean algorithm [17], [20] for solving the Yule–Walker equation. These algorithms also require $O(n^2)$ arithmetic operations.

Manuscript received June 11, 1990; revised August 21, 1991.
The authors are with CNET/PAB/RPE, 92131 Issy-Les-Moulineaux, France.
IEEE Log Number 9203228.

Although proposed for solving the same equation, these algorithms are based initially on various concepts. Our main purpose is therefore to compare them and to offer a synthetic view of these fast algorithms.

First, we recall that all these algorithms can be interpreted as Padé approximation algorithms [6], [7], [15], which provides a better understanding of the essential of the problem. This interpretation clarifies the relation between different algorithms, gives an explanation of the diversity of algorithms, and provides possibilities for the development of new ones.

Then, we show that all these algorithms have two different versions, belonging, respectively, to the following classes defined in [8]: 1) the one-pass algorithms, which evaluate directly the denominators of Padé approximants; and 2) the two-pass algorithms, which compute first the numerators or the remainders of Padé approximants via a sequence of elementary operations and then repeat the same sequence of operations to obtain the denominators.

It is shown that the one-pass algorithms are computationally more efficient than the two-pass algorithms, while the two-pass algorithms which provide more information, have the essential advantage of allowing the derivation of very powerful signal processing algorithms, namely, the doubling algorithms ("superfast algorithms") and the parallel algorithms [24]. Thus, for each algorithm belonging to one class, we develop the corresponding one belonging to the other class. The Levinson algorithm being of the one-pass type, we obtain its two-pass equivalent by a simple combination of this algorithm with that of Schur [25], which is already known. For the Berlekamp–Massey and Euclidean algorithms, we develop two new corresponding algorithms: the two-pass Berlekamp–Massey algorithm and the one-pass Euclidean algorithm. We complete therefore these algorithms in both classes. Note that one of these algorithms (the two-pass Berlekamp–Massey algorithm) seems to have been derived independently by Zarowski [28].

When the polynomials to be processed in the algorithms are seen as forward or backward prediction polynomials in the prediction theory, it is seen that the Levinson algorithm works on both forward and backward prediction polynomials, while the Euclidean works only on the forward one, and the Berlekamp–Massey only on the backward one. We develop for each of the last two algorithms a variant which is able to provide both types of predictors while maintaining the computational complexity almost unchanged.

Throughout the paper, the arithmetic complexity is evaluated for each algorithm. We show that the algorithms in the same class require nearly the same number of arithmetic operations, and that the Levinson algorithm is the only one for which the arithmetic complexity can be reduced when the matrix is symmetric.

Finally, we study the common structure of the two-pass algorithms allowing to apply the divide and conquer strategy. The application of this strategy converts the classical algorithms into the class of so-called doubling algorithms which is faster asymptotically [1]-[3], [15], [16], [22], [23]. We then present a general doubling algorithm which represents all the doubling algorithms in an exhaustive manner.

The preliminary section (Section II) recalls the Padé approximation and describes the Yule-Walker equation as a Padé approximation problem. Section III on classical algorithms is divided into three subsections, presenting successively the three algorithms for which we discuss the following points: the Padé approximants to be evaluated; the number of multiplications required; and the class to which the algorithm belongs.

In Section IV, we give a summary of these algorithms and point out the missing variants to be developed in Section V. Then, based on the common structure of the classical two-pass algorithms, Section VI provides a general principle for the application of the doubling strategy to the classical algorithms. Finally, we conclude the paper with a short summary. Note that, throughout this paper, we shall always consider the nonsingular case. How to deal with singularities during the recursions is out of the scope of this paper and can be found elsewhere [27], [9].

II. Preliminaries

Solving a Yule-Walker equation is shown to be closely related to the Padé approximation of a formal power series (see [6]). We recall the definition of the Padé approximation problem and its connection with the Yule-Walker equation.

A. Padé Approximant

Definition [13], [14], [21]: Let $C(z) = c_0 + c_1 z + c_2 z^2 + \cdots$ be a formal power series with $c_0 \neq 0$. A rational function of the form $u(z)/v(z)$ is an (m, n) Padé approximant for $C(z)$ if

$$\deg(u(z)) \leq m \tag{1}$$

$$\deg(v(z)) \leq n \tag{2}$$

$$C(z)v(z) - u(z) = O(z^{m+n+1}). \tag{3}$$

For a formal power series, there always exists a Padé approximant of order (m, n) represented by the rational

TABLE I
Padé Table

numerator order ↓ / denominator order →						
$r_{0,0}$	$r_{0,1}$	$r_{0,2}$	$r_{0,3}$	—	—	—
$r_{1,0}$	$r_{1,1}$	$r_{1,2}$	$r_{1,3}$	—	—	—
$r_{2,0}$	$r_{2,1}$	$r_{2,2}$	$r_{2,3}$	—	—	—
$r_{3,0}$	$r_{3,1}$	$r_{3,2}$	$r_{3,3}$	—	—	—
|	|	|	|			
|	|	|	|			

function $r_{m,n}(z)$:

$$r_{m,n}(z) = \frac{p_{m,n}(z)}{a_{m,n}(z)} \tag{4}$$

with $p_{m,n}(z)$ and $a_{m,n}(z)$ relatively prime and $p_{m,n}(0) = c_0$, $a_{m,n}(0) = 1$.

The Padé table of the formal power series $C(z)$ is a doubly infinite array of the uniquely determined rational function $r_{m,n}(z) = p_{m,n}(z)/a_{m,n}(z)$, which is represented in Table I.

By definition of $r_{m,0}(z)$, the first column contains the partial sums of $C(z)$:

$$r_{m,0}(z) = \sum_{k=0}^{m} c_k z^k.$$

The power series $C(z)$ is said to be normal if, for each pair (m, n), the Maclaurin expansion of $r_{m,n}(z)$ agrees with $C(z)$ exactly up to the power z^{m+n}. The Padé approximant $r_{m,n}$ is normal if it occurs exactly once in the Padé table. The power series $C(z)$ is normal if all its Padé approximants are normal; that is, no two are equal [14].

B. Yule-Walker Equation

The Yule-Walker (YW) equation is defined as a system of linear equations:

$$\begin{bmatrix} c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & & \cdots & c_n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \alpha_n \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

or

$$\begin{bmatrix} c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & & \cdots & c_n \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \alpha_n \end{bmatrix} \tag{5}$$

where $a(z)$ and $b(z)$ ($a(z) = a_0 + a_1 z + \cdots + a_n z^n$, $b(z) = b_0 + b_1 z + \cdots + b_n z^n$) are seen as the forward and backward prediction polynomials of the corresponding system. It is shown in [6] that the solution of the YW equations can be described as a Padé approximation problem. This interpretation is recalled below.

Equation (5) is completed as follows:

$$\begin{bmatrix} c_0 & 0 & \cdots & 0 \\ c_1 & c_0 & & \vdots \\ \vdots & & \ddots & 0 \\ c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & c_{2n-1} & \cdots & c_n \\ 0 & c_{2n} & & c_{n+1} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & c_{2n} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \bar{p}_0 \\ \bar{p}_1 \\ \vdots \\ \bar{p}_n \\ 0 \\ \vdots \\ 0 \\ \bar{q}_0 \\ \bar{q}_1 \\ \vdots \\ \bar{q}_{n-1} \end{bmatrix}$$

or

$$\begin{bmatrix} c_0 & 0 & \cdots & 0 \\ c_1 & c_0 & & \vdots \\ \vdots & & \ddots & 0 \\ c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & & \vdots \\ c_{2n} & c_{2n-1} & \cdots & c_n \\ 0 & c_{2n} & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & c_{2n} \end{bmatrix} \begin{bmatrix} b_0 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \\ 0 \\ \vdots \\ 0 \\ q_0 \\ q_1 \\ \vdots \\ q_n \end{bmatrix} \quad (6)$$

with $\bar{p}_n = q_0 = \alpha_n$. By noting $c(z) = c_0 + c_1 z + \cdots + c_{2n} z^{2n}$, $C(z) = c(z) + O(z^{2n+1})$, $\bar{p}(z) = \bar{p}_0 + \bar{p}_1 z + \cdots + \bar{p}_n z^n$, and $\bar{q}(z) = \bar{q}_0 + \bar{q}_1 z + \cdots + \bar{q}_{n-1} z^{n-1}$, $p(z) = p_0 + p_1 z + \cdots + p_{n-1} z^{n-1}$, and $q(z) = q_0 + q_1 z + \cdots + q_n z^n$, we have the polynomial description of (6)

$$C(z)a(z) = \bar{p}(z) + z^{2n+1}\bar{q}(z)$$
$$C(z)b(z) = p(z) + z^{2n}q(z). \quad (7)$$

Comparing (7) with the definition of the Padé approximant, we see that $\bar{p}(z)/a(z)$ and $p(z)/b(z)$ with deg $(\bar{p}(z)) = n$, deg $(a(z)) = n$, deg $(p(z)) = n - 1$, deg $(b(z)) = n$ are, respectively, the Padé approximants of order (m, n) and $(n - 1, n)$ for $C(z)$. The solution of the Yule-Walker equation is thus described as a Padé approximation problem.

This simple consideration already provides an interesting information: a_n is the forward predictor, while b_n is the backward one. The understanding of a_n and b_n as Padé approximants of order (n, n) and $(n - 1, n)$, respectively, shows that the backward predictor is certainly easier to obtain, since it is an approximant of lower order than the forward one. Of course, both forward and backward predictors are essentially equivalent (by simple permutation), the Padé interpretation only tells us that it may be computationally (slightly) advantageous to compute them by means of a backward predictor, if a single one is required.

III. Classical Algorithms

We present here three classical algorithms: the Levinson algorithm (and Schur algorithm), Euclidean algorithm, and Berlekamp-Massey algorithm. Our goal is not to describe the details of the algorithms, but to 1) interprete them as Padé approximation algorithms; 2) group them together in two classes, namely, one-pass algorithms and two-pass algorithms; and 3) compare their arithmetic complexity.

Considering that the algorithms require almost as many additions as multiplications, we evaluate only the latter for each algorithm. The number will be noted after each step in a square bracket.

A. Levinson Algorithm

We recall the Levinson algorithm as well as the Schur algorithm. For these algorithms, we can find in [7] their interpretation as Padé approximation algorithms. The Levinson algorithm [12], [18] solves the following equations:

$$\begin{bmatrix} c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & & \cdots & c_n \end{bmatrix} [\boldsymbol{a}_n \quad \boldsymbol{b}_n] = \begin{bmatrix} \alpha_n & 0 \\ 0 & \vdots \\ \vdots & 0 \\ 0 & \alpha_n \end{bmatrix} \quad (8)$$

which are Yule-Walker equations with $\boldsymbol{a}_n$ and $\boldsymbol{b}_n$, respectively, the forward and backward prediction vectors in the prediction theory. Let C_i be the top principal matrix of order $i + 1$ of the matrix in the above equation, which is supposed to be nonsingular. The prediction vectors of two successive orders obey the following recursion relationships:

$$C_i \left\{ \begin{bmatrix} \boldsymbol{a}_{i-1} \\ 0 \end{bmatrix} + K_{a,i} \begin{bmatrix} 0 \\ \boldsymbol{b}_{i-1} \end{bmatrix} \right\} = \begin{bmatrix} \alpha_{i-1} \\ \boldsymbol{0} \\ \beta_{a,i-1} \end{bmatrix} + K_{a,i} \begin{bmatrix} \beta_{b,i-1} \\ \boldsymbol{0} \\ \alpha_{i-1} \end{bmatrix} = \begin{bmatrix} \alpha_i \\ \boldsymbol{0} \end{bmatrix} \quad (9)$$

$$C_i \left\{ \begin{bmatrix} 0 \\ \boldsymbol{b}_{i-1} \end{bmatrix} + K_{b,i} \begin{bmatrix} \boldsymbol{a}_{i-1} \\ 0 \end{bmatrix} \right\} = \begin{bmatrix} \beta_{b,i-1} \\ \boldsymbol{0} \\ \alpha_{i-1} \end{bmatrix} + K_{b,i} \begin{bmatrix} \alpha_{i-1} \\ \boldsymbol{0} \\ \beta_{a,i-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \alpha_i \end{bmatrix}. \quad (10)$$

These two equations constitute the main recursion relationships of the Levinson algorithm. The algorithm is summarized below.

Algorithm (Levinson)

$$a_0(z) = b_0(z) = 1, \quad \alpha_0 = c_n$$

for $i = 1, \cdots, n$

$$\beta_{b,i-1} = \sum_{j=0}^{i-1} b_{i-1,j}c_{n-1-j} \qquad [i-1]$$

$$\beta_{a,i-1} = \sum_{j=0}^{i-1} a_{i-1,j}c_{n+i-j} \qquad [i-1]$$

$$K_{b,i} = -\beta_{b,i-1}/\alpha_{i-1} \qquad [1]$$

$$K_{a,i} = -\beta_{a,i-1}/\alpha_{i-1} \qquad [1]$$

$$\begin{bmatrix} b_i(z) \\ a_i(z) \end{bmatrix} = \begin{bmatrix} z & K_{b,i} \\ K_{a,i}z & 1 \end{bmatrix} \begin{bmatrix} b_{i-1}(z) \\ a_{i-1}(z) \end{bmatrix} \qquad [2(i-1)]$$

$$\alpha_i = \alpha_{i-1}(1 - K_{a,i}K_{b,i}) = \alpha_{i-1} + \beta_{b,i-1}K_{a,i} \qquad [1].$$

We can easily evaluate the number of multiplications required by this algorithm, which is equal to $2n^2 + n$. In the case where matrix C is symmetric, we have $\beta_{a,i} = \beta_{b,i}$, $K_{a,i} = K_{b,i}$, and $a_i(z) = b_i^{\#}(z) = z^i b_i(z^{-1})$, this number is reduced to $n^2 + n$.

In the above algorithm, the parameters $K_{a,i}$ and $K_{b,i}$, known as reflection coefficients, are computed via the scalar products $\beta_{a,i}$ and $\beta_{b,i}$. These parameters provide the recursion relationship for the prediction polynomials $a_i(z)$ and $b_i(z)$, and the only recursion is carried out on these prediction polynomials. This algorithm is thus classified as a one-pass algorithm.

The Schur algorithm [25] provides an alternative method for computing the reflection coefficients $\{K_{a,i}, K_{b,i}\}$ via the recursion of the polynomials $p_i(z)$, $q_i(z)$, $\bar{p}_i(z)$, and $\bar{q}_i(z)$:

Algorithm 2 (Schur)

$$\begin{bmatrix} p_i(z) \\ \bar{p}_i(z) \end{bmatrix} = \begin{bmatrix} z & K_{b,i} \\ K_{a,i}z & 1 \end{bmatrix} \begin{bmatrix} p_{i-1}(z) \\ \bar{p}_{i-1}(z) \end{bmatrix}$$

$$z\begin{bmatrix} q_i(z) \\ z\bar{q}_i(z) \end{bmatrix} = \begin{bmatrix} z & K_{b,i} \\ K_{a,i}z & 1 \end{bmatrix} \begin{bmatrix} q_{i-1}(z) \\ z\bar{q}_i(z) \end{bmatrix}$$

with

$$K_{b,i+1} = -p_{i,n-1}/\bar{p}_{i,n}$$

$$K_{a,i+1} = -\bar{q}_{i,0}/q_{i,0}$$

where

$$p_i(z) = \sum_{j=0}^{n-1} p_{i,j}z^j, \quad \bar{p}_i(z) = \sum_{j=0}^{n} \bar{p}_{i,j}z^j$$

$$q_i(z) = \sum_{j=0}^{n} q_{i,j}z^j, \quad \bar{q}_i(z) = \sum_{j=0}^{n-1} \bar{q}_{i,j}z^j.$$

The variables $p_{i,j}$, $q_{i,j}$, $\bar{p}_{i,j}$, $\bar{q}_{i,j}$ involved in the algorithm are found in the following linear equations:

$$\begin{bmatrix} c_0 & 0 & \cdots & 0 \\ c_1 & c_0 & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & \ddots & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & c_{2n-1} & \cdots & c_n \\ 0 & c_{2n} & & c_{n+1} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & c_{2n} \end{bmatrix} \left[\begin{array}{ccccc|ccccc} b_{0,0} & \cdots & b_{i,0} & \cdots & b_{n,0} & a_{0,0} & \cdots & a_{i,0} & \cdots & a_{n,0} \\ 0 & \cdot & \cdots & & b_{n,1} & 0 & \cdot & & & a_{n,1} \\ \vdots & & b_{i,i} & & \vdots & \vdots & & a_{i,i} & \cdots & \vdots \\ \vdots & & \cdots & \cdot & \vdots & \vdots & & & \cdot & \vdots \\ 0 & \cdots & \cdots & 0 & b_{n,n} & 0 & \cdots & \cdots & 0 & a_{n,n} \end{array}\right]$$

$$= \left[\begin{array}{cccc|cccc} p_{0,0} & & \cdots & p_{n,0} & \bar{p}_{0,0} & & \cdots & \bar{p}_{n,0} \\ \vdots & & & \vdots & \vdots & & & \vdots \\ p_{0,n-1} & & & p_{n,n-1} & & & & \\ q_{0,0} & 0 & \cdots & 0 & \bar{p}_{0,n} & & & \bar{p}_{n,n} \\ \vdots & \ddots & & \vdots & \bar{q}_{0,0} & 0 & \cdots & 0 \\ \vdots & & \ddots & 0 & \vdots & & & \vdots \\ q_{0,n} & & \cdots & q_{n,0} & \bar{q}_{0,n-1} & & \ddots & 0 \\ 0 & & & \vdots & 0 & \bar{q}_{1,0} & & \bar{q}_{n,0} \\ \vdots & \ddots & & \vdots & \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & q_{n,n} & 0 & \cdots & 0 & \bar{q}_{n,n-1} \end{array}\right]. \qquad (11)$$

Obviously, we have $p_{i,n-1} = \beta_{b,i}$, $\bar{q}_{i,0} = \beta_{a,i}$, and $q_{i,0} = \bar{p}_{i,n} = \alpha_i$.

The Levinson type two-pass algorithm is obtained by combining the computation of the reflection coefficients $K_{a,i}$ and $K_{b,i}$ in Schur's manner, and the prediction polynomials $a_i(z)$ and $b_i(z)$ in Levinson's manner, in such a way that the recursive iterations are performed altogether on $p_i(z)$, $q_i(z)$, $\bar{p}_i(z)$, $\bar{q}_i(z)$ and $a_i(z)$, $b_i(z)$. The combined algorithm called Levinson/Schur algorithm is written below.

Algorithm 3 (Levinson/Schur)

```
First pass:
    $p_0(z) = c_0 + c_1 z + \cdots + c_{n-1} z^{n-1}$,
    $\bar{p}_0(z) = c_0 + c_1 z + \cdots + c_n z^n$
    $q_0(z) = c_n + c_{n+1} z + \cdots + c_{2n} z^n$,
    $\bar{q}_0(z) = c_{n+1} + \cdots + c_{2n} z^{n-1}$
  for $i = 1, \cdots, n$
    $K_{b,i} = -p_{i-1,n-1}/\bar{p}_{i-1,n}$    [1]
    $K_{a,i} = -\bar{q}_{i-1,0}/q_{i-1,0}$    [1]
    $\begin{bmatrix} p_i(z) \\ \bar{p}_i(z) \end{bmatrix} = \begin{bmatrix} z & K_{b,i} \\ K_{a,i} z & 1 \end{bmatrix} \begin{bmatrix} p_{i-1}(z) \\ \bar{p}_{i-1}(z) \end{bmatrix}$    [2(n - i)]
    $z \begin{bmatrix} q_i(z) \\ z\bar{q}_i(z) \end{bmatrix} = \begin{bmatrix} z & K_{b,i} \\ K_{a,i} z & 1 \end{bmatrix} \begin{bmatrix} q_{i-1}(z) \\ z\bar{q}_{i-1}(z) \end{bmatrix}$.    [2(n - i)]
Second pass:
    $a_0(z) = b_0(z) = 1$
  for $i = 1, \cdots, n$
    $\begin{bmatrix} b_i(z) \\ a_i(z) \end{bmatrix} = \begin{bmatrix} z & K_{b,i} \\ K_{a,i} & 1 \end{bmatrix} \begin{bmatrix} b_{i-1}(z) \\ a_{i-1}(z) \end{bmatrix}$.    [2(i - 1)]
```

The number of multiplications required for evaluating this algorithm is $3n^2 - n$, which is more than the corresponding one-pass algorithm (Levinson algorithm). If the matrix is symmetric, this number may be reduced straightforwardly to $1.5n^2 - 0.5n$.

As seen in Section II, the ratio $p_i(z)/b_i(z)$ and $\bar{p}_i(z)/a_i(z)$, with $p_i(z) = p_{i,0} + p_{i,1}z + \cdots + p_{i,n-1}z^{n-1}$, and $\bar{p}_i(z) = \bar{p}_{i,0} + \bar{p}_{i,1}z + \cdots + \bar{p}_{i,n}z^n$, are respectively, the Padé approximants of types $(n - 1, i)$ and (n, i) for $C(z)$. Thus, for i from 0 to n, it is seen that this algorithm computes the Padé approximants of types $(n - 1, 0)$, $(n, 0)$, $(n - 1, 1)$, $(n, 1)$, $\cdots$, $(n - 1, n)$, (n, n), which has the form of a sawtooth related to a row in the Padé table (see Fig. 1). This is a result given in [7].

Based on a conventional three-term recursion for polynomials which are orthogonal on the unit circle, a variant of the Levinson algorithm has been proposed in [26]. This variant computes only the polynomials $a_i(z)$ or $b_i(z)$, the forward or backward predictor, but not both of them. As a consequence, only the Padé approximants of types $(n - 1, 0)$, $(n - 1, 1)$, $\cdots$, $(n - 1, n)$ (or $(n, 0)$, $(n, 1)$, $\cdots$, (n, n)) are computed, which constitute a simple row in the Padé table.

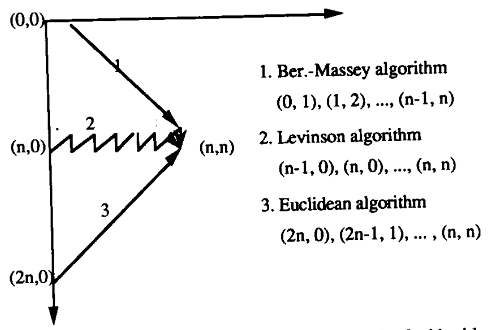


Fig. 1. Evolution of the various algorithms in the Padé table.

We note that all these types of algorithms (Levinson, Schur, Levinson/Schur) require that all the principal minors of matrix C be not singular, that is, the polynomial $c(z)$ must be normal. The generalization of this algorithm to the case where matrix C has any rank profile can be found in [9], [26].

B. Euclidean Algorithm

The Euclidean algorithm was originally developed for computing the greatest common divisor (GCD) of two entries or two polynomials [5], [20]. The application of this algorithm to the computation of Padé approximants is an original suggestion of Kronecker [17]. Using this algorithm for solving a YW equation is related directly to its application for Padé approximation [5]. We first recall the original algorithm for the computation of the GCD of two polynomials, and its extension for solving YW equation. The relation between this algorithm and the Padé approximation has been described in [6].

1) Extended Euclidean Algorithm: Let us consider the problem of computing the GCD of two polynomials $s(z)$ and $t(z)$: GCD (s, t), and the "comultiplicators" $u(z)$ and $v(z)$:

$$s(z)u(z) + t(z)v(z) = \text{GCD}\,(s, t). \tag{12}$$

Suppose that deg $(s(z)) \geq$ deg $(t(z))$, the Euclidean algorithm computes GCD (s, t) by a sequence of recursive divisions: let $s_0(z) = s(z)$, $s_1(z) = t(z)$, we construct a sequence of remainder polynomials $s_i(z)$ for $1 \leq i \leq n$ by Euclidean division:

$$s_{i-1}(z) = qu_i(z)s_i(z) + s_{i+1}(z). \tag{13}$$

The recursion being continued till $i = n$ when $s_{n+1}(z) = 0$, the resulting $s_n(z)$ is the GCD (s, t).

Once the quotients $qu_i(z)$ are known, we can evaluate $u(z)$ and $v(z)$. The algorithm involving these two passes (computation of GCD then evaluation of the comultiplicators) is called the "extended Euclidean algorithm" [8], and is summarized below.

```
Algorithm 4 (Extended Euclidean)
  First pass:
    $s_0(z) = s(z); \quad s_1(z) = t(z)$
    for $1 \le i \le n$
      $qu_i(z)$ = quotient of the Euclidean division $s_{i-1}(z)/s_i(z)$
      $\begin{bmatrix} s_i(z) \\ s_{i+1}(z) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -qu_i(z) \end{bmatrix} \begin{bmatrix} s_{i-1}(z) \\ s_i(z) \end{bmatrix}$
    $\begin{bmatrix} s_n(z) \\ s_{n+1}(z) \end{bmatrix} = \begin{bmatrix} \mathrm{GCD}(s, t) \\ 0 \end{bmatrix}.$
  Second pass:
    $\begin{bmatrix} u_0 & v_0 \\ u_1 & v_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
    for $i = 1, \cdots, n$
      $\begin{bmatrix} u_i & v_i \\ u_{i+1} & v_{i+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -qu_i(z) \end{bmatrix} \begin{bmatrix} u_{i-1} & v_{i-1} \\ u_i & v_i \end{bmatrix}.$
```

2) Extended Euclidean Algorithm for Solving the YW Equation: To apply the extended Euclidean algorithm for solving a YW equation, we consider the polynomial description of the problem, i.e., the first equation of (7):

$$-z^{2n+1}\,\bar{q}(z) + c(z)a(z) = \bar{p}(z). \tag{14}$$

By taking $s(z) = -z^{2n+1}$, $t(z) = c(z)$, we remark that (11) and (13) are of the same form with $\bar{q}(z) = u(z)$ and $a(z) = v(z)$, if $\bar{p}(z)$ is the GCD of $s(z)$ and $t(z)$. In fact, if we apply the extended Euclidean algorithm to the polynomials $s(z) = -z^{2n+1}$ and $t(z) = c(z)$, the GCD would be found as a result of the recursion when deg $(\bar{p}(z)) = 0$. However, we are interested in the solution of the YW equations. If $c(z)$ is normal, this solution is found at the nth recursion, the solution $a(z)$ is of degree n, and deg $(p(z))$ is also equal to n. If $c(z)$ is not normal, then the solution of the YW equations is found at an intermediate step where deg $(a(z)) \le n$ and deg $(\bar{p}(z)) \le n$. Thus the Euclidean algorithm for solving (13) is as follows (see [5] for a detailed demonstration).

```
Algorithm 5 (Euclidean)
  First pass:
    $\bar{p}_{-1}(z) = -z^{2n+1}, \quad \bar{p}_0(z) = c(z)$
    for $i = 0, \cdots, r$
      $qu_i(z)$ = quotient of the Euclidean division $\bar{p}_{i-1}(z)/\bar{p}_i(z)$
      $\begin{bmatrix} \bar{p}_i(z) \\ \bar{p}_{i+1}(z) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -qu_i(z) \end{bmatrix} \begin{bmatrix} \bar{p}_{i-1}(z) \\ \bar{p}_i(z) \end{bmatrix}$   [4n - 4i - 2]
    deg $(\bar{p}_r(z)) \ge n + 1$
    deg $(\bar{p}_{r+1}(z)) \le n$.
  Second pass:
    $a_{-1}(z) = 0, \quad a_0(z) = 1$
    for $i = 0, \cdots, r$
      $\begin{bmatrix} a_i(z) \\ a_{i+1}(z) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -qu_i(z) \end{bmatrix} \begin{bmatrix} a_{i-1}(z) \\ a_i(z) \end{bmatrix}.$   [2(i + 1)]
  Solution:
    $a(z) = a_{r+1}(z)/a_{r+1}(0)$.
```

In the case where $c(z)$ is normal, $r = n - 1$, and the number of multiplications required by this algorithm is $3n^2 + 4n$.

The evolution of the parameters involved in this algorithm is shown below in matrix form:

$$\begin{bmatrix} c_0 & 0 & \cdots & 0 \\ c_1 & c_0 & & \vdots \\ \vdots & & \ddots & 0 \\ c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & c_{2n-1} & \cdots & c_n \\ 0 & c_{2n} & & c_{n+1} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & c_{2n} \end{bmatrix} \begin{bmatrix} a_{0,0} & \cdots & a_{i,0} & \cdots & a_{n,0} \\ 0 & \vdots & & & \\ \vdots & & a_{i,i} & & \vdots \\ \vdots & & & \cdots & \vdots \\ 0 & \cdots & & 0 & a_{n,n} \end{bmatrix} = \begin{bmatrix} \bar{p}_{0,0} & \bar{p}_{1,0} & \cdots & \bar{p}_{n,0} \\ \bar{p}_{0,1} & & & \bar{p}_{n,1} \\ \vdots & & & \vdots \\ \bar{p}_{0,n} & & & \bar{p}_{n,n} \\ \vdots & & & 0 \\ \vdots & & \cdot^{\cdot^{\cdot}} & \vdots \\ \bar{p}_{0,2n} & 0 & \cdots & 0 \\ 0 & \bar{q}_{1,0} & \cdots & \bar{q}_{n,0} \\ \vdots & & & \vdots \\ 0 & \cdots & 0 & \bar{q}_{n,n-1} \end{bmatrix}. \tag{15}$$

This algorithm is of the two-pass type. It computes successively the Padé approximants $(2n, 0)$, $(2n - 1, 1)$, $(2n - 2, 2)$, $\cdots$, (n, n), which form an antidiagonal of the Padé table (Fig. 1), a result having been given in [6]. If $c(z)$ is a normal power series, all these approximants are distinct and deg $(a(z))$ = deg $(\bar{p}(z)) = n$.

We observe that this algorithm computes successively the solution of the following linear equations (16), in which $a_i(z) = a_{i0} + a_{i1}z + \cdots + a_{ii}z^i$ are seen as the forward prediction polynomials:

$$\begin{bmatrix} c_{2n-i} & c_{2n-i-1} & \cdots & c_{2n-2i} \\ c_{2n-i+1} & c_{2n-i} & & c_{2n-2i+1} \\ \vdots & & \ddots & \vdots \\ c_{2n} & & \cdots & c_{2n-i} \end{bmatrix} \begin{bmatrix} a_{i,0} \\ a_{i,1} \\ \vdots \\ a_{i,i} \end{bmatrix} = \begin{bmatrix} \bar{p}_{i,2n-i} \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{16}$$

It will be shown in Section V that it is possible to obtain, in addition, the backward prediction polynomials without increasing the computational complexity (algorithm 8 below).

C. Berlekamp-Massey Algorithm

This algorithm [4] was originally developed for decoding the BCH codes. It was interpreted as the synthesis of an autoregressive filter [19], with the following system of equations:

$$\begin{bmatrix} c_{n-1} & c_{n-2} & \cdots & c_0 \\ c_n & c_{n-i} & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n-2} & & \cdots & c_{n-1} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} -c_n \\ \\ \vdots \\ -c_{2n-1} \end{bmatrix} \tag{17}$$

which is equivalent to

$$\begin{bmatrix} c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & & \cdots & c_n \end{bmatrix} \begin{bmatrix} 1 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \alpha_n \end{bmatrix}. \tag{18}$$

This algorithm computes therefore the backward prediction vector in the prediction theory, while the Euclidean algorithm computes the forward one, and the Levinson algorithm computes both of them. Nevertheless, changing the initial algorithm to obtain the forward one is trivial since we need only to carry out the algorithm on the reverse of the polynomial $c(z)$. A variant which computes both forward and backward prediction vectors is developed in Section V (algorithm 10).

We give here only the algorithm. For its derivation, the reader is referred to [5], [4], [19].

Algorithm 6 (Berlekamp-Massey)

```
$b_0(z) = t_0(z) = 1$
$L = 0, \quad dt = 1$
for $r = 0, \cdots, 2n-1$
    $d_r = \sum_{j=0}^{L} b_j c_{r-j}$                                   [L]
    if ($d_r = 0$), then
        $\left\{ \begin{bmatrix} b_{r+1}(z) \\ t_{r+1}(z) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & z \end{bmatrix} \begin{bmatrix} b_r(z) \\ t_r(z) \end{bmatrix} \right\}$;
    if (($d_r \neq 0$) and ($2L > r$)), then
        $\left\{ \begin{bmatrix} b_{r+1}(z) \\ t_{r+1}(z) \end{bmatrix} = \begin{bmatrix} 1 & -d_r dt^{-1} z \\ 0 & z \end{bmatrix} \begin{bmatrix} b_r(z) \\ t_r(z) \end{bmatrix} \right\}$       [L]
    else (($d_r \neq 0$) and ($2L \leq r$))
        { $L = r + 1 - L$
        $\begin{bmatrix} b_{r+1}(z) \\ t_{r+1}(z) \end{bmatrix} = \begin{bmatrix} 1 & -d_r dt^{-1} z \\ 1 & 0 \end{bmatrix} \begin{bmatrix} b_r(z) \\ t_r(z) \end{bmatrix}$
        $dt = d_r$ }.                                                    [L]
Solution $b(z) = b_r(z)$.
```

We give here some explanation about the parameters used in the algorithm: L is the order of polynomials $b_r(z)$, which is different from the recursion order r. The recursions are carried out on polynomials $b_r(z)$, $r = 0, 1, \cdots, 2n-1$, and some of them are stored in the auxiliary polynomials $t_r(z)$ when they are denominators of a Padé approximants of $c(z)$. In the case where $c(z)$ is normal, only the polynomials at even recursion orders will be stored. It is easily seen that this algorithm is of the one-pass type, since the polynomials $b_r(z)$ are obtained in a nonrecursive manner.

The number of multiplications required by this algorithm in the case where $c(z)$ is normal, is $2n^2 + 2n$. We remark that it is of the same order as that of the other one-pass algorithm (Levinson algorithm) stated before.

We emphasize the fact that all the $b_r(z)$ are not denominators of Padé approximants of $c(z)$, but only the ones that are stored in $t_r(z)$ are the denominators of the Padé approximant of order $(r/2, r/2)$ with r even in the case where $c(z)$ is normal. Thus, for $r = 0, \cdots, 2n$, this algorithm computes the denominators of the Padé approximants of types $(0, 1), \cdots, (n-1, n)$, which constitutes a main diagonal in the Padé table.

IV. Summary of the Classical Algorithms

As seen above, most of the results provided in Section III have been explained elsewhere in the literature, at least partially. Nevertheless, only a global view of YW equations as a Padé approximation problem will allow the derivation of new results:

1) All these three algorithms evaluate the denominator of the Padé approximant of type (n, n) or $(n-1, n)$. They cover various paths in the Padé table, as shown in Fig. 1.

TABLE II
VARIANTS OF THE ALGORITHMS KNOWN OR MISSING

	One-Pass	Two-Pass	Forward or Backward	Forward and Backward
Levinson	Known	Known	Known	Known
Euclidean	Missing	Known	Known	Missing
Berlekamp	Known	Missing	Known	Missing

2) The Levinson algorithm is of the one-pass type. We obtain the corresponding two-pass version by a combination with the Schur algorithm. The Euclidean algorithm is shown to be of the two-pass type, its one-pass version is missing. The Berlekamp-Massey algorithm is of the one-pass type, its two-pass version is to be developed.

3) The Levinson algorithm computes both forward and backward predictors of the associated system. The Euclidean algorithm and the Berlekamp-Massey algorithm compute, respectively, the forward and backward predictor, the variants of these two algorithms for computing both predictors altogether are missing.

Table II shows variants of the algorithms already known, and the missing variants to be developed in the next section.

V. Development of the Missing Variants

A. The Variants of the Euclidean Algorithm

By comparing the Euclidean algorithm with that of Levinson/Schur, we remark that the first pass looks like Schur's algorithm, and the second one looks like Levinson's. We are therefore motivated to develop a one-pass algorithm which would be similar to that of Levinson, by hoping that it will be more efficient than the initial (two-pass) Euclidean algorithm.

As for the Levinson algorithm, we suppose that the formal power series $c(z)$ associated with the Toeplitz matrix is normal. The idea is to replace the computation of quotients $Q_i(z)$ in the initial algorithm by a scalar product which is a non recursive computation, just like the computation of the reflection coefficients in the Levinson algorithm. For this, we examine the explicit expression of $Q_i(z)$:

$$Q_i(z) = \text{quotient of the Euclidean division } \bar{p}_{i-1}(z)/\bar{p}_i(z) \tag{19}$$

with

$$\bar{p}_{i-1}(z) = \bar{p}_{i-1,0} + \bar{p}_{i-1,1}z + \cdots + \bar{p}_{i-1,2n-i+1}z^{2n-i+1}$$
$$\bar{p}_i(z) = \bar{p}_{i,0} + \bar{p}_{i,1}z + \cdots + \bar{p}_{i,2n-i}z^{2n-i}$$

$c(z)$ is normal, then $\bar{p}_{i,2n-i} \neq 0$, we obtain

$$Q_i(z) = Q_{i,0} + Q_{i,1}z \tag{20}$$

with

$$Q_{i,1} = \bar{p}_{i-1,2n-i+1}/\bar{p}_{i,2n-i}$$
$$Q_{i,0} = (\bar{p}_{i-1,2n-i} - Q_{i,1}\bar{p}_{i,2n-i-1})/\bar{p}_{i,2n-i}.$$

And, the recursion of $\bar{p}_i(z)$ is replaced by the computation of its last two elements:

$$\bar{p}_{i,2n-i} = [c_{2n-i}, \cdots, c_{2n-2i}][a_{i,0}, \cdots, a_{i,i}]^T \tag{21}$$
$$\bar{p}_{i,2n-i-1} = [c_{2n-i-1}, \cdots, c_{2n-2i-1}][a_{i,0}, \cdots, a_{i,i}]^T. \tag{22}$$

This algorithm is summarized below:

Algorithm 7 (One-Pass Euclidean)

$$\begin{bmatrix} \bar{p}_{-1,2n} \\ \bar{p}_{-1,2n+1} \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} a_{-1}(z) \\ a_0(z) \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

```
for $i = 0, \cdots, n-1$
    $\bar{p}_{i,2n-i} = \sum_{j=0}^{i} a_{i,j}c_{2n-i-j}$    $[i+1]$
    $\bar{p}_{i,2n-i-1} = \sum_{j=0}^{i} a_{i,j}c_{2n-i-1-j}$    $[i+1]$
    $Q_{i,1} = \bar{p}_{i-1,2n-i+1}/\bar{p}_{i,2n-i}$    $[1]$
    $Q_{i,0} = (\bar{p}_{i-1,2n-i} - Q_{i,1}\bar{p}_{i,2n-i-1})/\bar{p}_{i,2n-i}$    $[2]$
    $a_{i+1}(z) = a_{i-1}(z) - (Q_{i,0} + Q_{i,1}z)a_i(z).$    $[2i+2]$
Solution:
    $a(z) = a_n(z)/a_n(0).$
```

The resulting algorithm requires $(2n^2 + 5n)$ multiplications. Indeed, this one-pass algorithm is computationally more efficient than the initial two-pass algorithm. It should be noted that this algorithm relies on the assumption that the polynomial $c(z)$ is normal, but it can be generalized for the abnormal case without difficulty.

As we have mentioned before, the algorithm computes only the forward predictor solution. The variant which provides both the forward and backward one is developed below.

First, we introduce an auxiliary polynomial $b_i(z) = b_{i0} + b_{i1}z + \cdots + b_{ii}z^i$, which represents a backward prediction polynomial. It is the solution of the following linear equations:

$$\begin{bmatrix} c_{2n-i} & c_{2n-i-1} & \cdots & c_{2n-2i} \\ c_{2n-i+1} & c_{2n-i} & & c_{2n-2i+1} \\ \vdots & & \ddots & \vdots \\ c_{2n} & & \cdots & c_{2n-i} \end{bmatrix} \begin{bmatrix} b_{i,0} \\ b_{i,1} \\ \vdots \\ b_{i,i} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ q_{i,0} \end{bmatrix}. \tag{23}$$

Then, we organize the variables involved in the algorithm in the following matrices:

$$\begin{bmatrix} c_0 & 0 & \cdots & 0 \\ c_1 & c_0 & & \vdots \\ \vdots & & \ddots & 0 \\ c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & \ddots & \vdots \\ c_{2n} & c_{2n-1} & \cdots & c_n \\ 0 & c_{2n} & & c_{n+1} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & c_{2n} \end{bmatrix} \begin{bmatrix} a_{0,0} & \cdots & a_{i,0} & \cdots & a_{n,0} & b_{0,0} & \cdots & b_{1,0} & \cdots & b_{n,0} \\ 0 & & & & & 0 & & & & \\ \vdots & & a_{i,i} & & \vdots & \vdots & & b_{i,i} & & \vdots \\ & & & & \cdots & & & & \cdots & \\ 0 & \cdots & & 0 & a_{n,n} & 0 & \cdots & & 0 & b_{n,n} \end{bmatrix}$$

$$= \begin{bmatrix} \overline{p}_{0,0} & \overline{p}_{1,0} & & \overline{p}_{n,0} & p_{0,0} & p_{1,0} & & p_{n,0} \\ \overline{p}_{0,1} & & & & p_{0,1} & & & \\ \vdots & & & & \vdots & & & p_{n,n-1} \\ \overline{p}_{0,n} & & & \overline{p}_{n,n} & p_{0,n} & & & 0 \\ \vdots & & & 0 & & & \cdot^{\cdot^{\cdot}} & \vdots \\ \vdots & & \cdot^{\cdot^{\cdot}} & \vdots & p_{0,2n-1} & 0 & \cdots & 0 \\ \overline{p}_{0,2n} & 0 & \cdots & 0 & q_{0,0} & & \cdots & q_{n,0} \\ 0 & \overline{q}_{1,0} & & \overline{q}_{n,0} & 0 & & & \vdots \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \overline{q}_{n,n-1} & 0 & \cdots & 0 & q_{n,n-1} \end{bmatrix}. \tag{24}$$

The recursions for $a_i(z)$ and $b_i(z)$ are the same ones as for $\overline{p}_i(z)$ and $p_i(z)$, which are easily deduced by decreasing at each step one degree of the polynomials $\overline{p}_i(z)$ and $p_i(z)$. This recursion depends only on the last coefficients of $\overline{p}_i(z)$ and $p_i(z)$ ($\overline{p}_{i,2n-i}$ and $p_{i,2n-i+1}$). For example, it may be as follows:

$$a_i(z) = -(p_{i-1,2n-i}/\overline{p}_{i-1,2n-i+1})a_{i-1}(z) + zb_{i-1}(z)$$

$$b_i(z) = -(p_{i-1,2n-i}/\overline{p}_{i,2n-i})a_i(z) + b_{i-1}(z). \tag{25}$$

If these last coefficients are computed via the recursion of the polynomials $\overline{p}_i(z)$ and $p_i(z)$, then the resulting algorithm is of the two-pass type. If they are computed via scalar products, we obtain a one-pass algorithm. We give here only the one-pass version.

Algorithm 8 (Forward and Backward Euclidean)

```
$\begin{bmatrix} a_0(z) \\ b_0(z) \end{bmatrix} \leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$p_{0,2n} = c_{2n}$
for $i = 1, \cdots, n$
    $p_{i-1,2n-i} = \sum_{j=0}^{i-1} b_{i-1,j}c_{2n-i-j}$                          [i]
    $a_i(z) = -(p_{i-1,2n-i}/\overline{p}_{i-1,2n-i+1})a_{i-1}(z)$
              $+ zb_{i-1}(z)$                                             [i + 1]
    $\overline{p}_{i,2n-i} = \sum_{j=0}^{i} a_{i,j}c_{2n-i-j}$                            [i + 1]
    $b_i(z) = -(p_{i-1,2n-i}/\overline{p}_{i,2n-i})a_i(z) + b_{i-1}(z).$
                                                                  [i + 2]
```

Solution:

$$a(z) = a_n(z)/a_n(0)$$

$$b_n(z) = b_n(z)/b_n(0).$$

The resulting algorithm requires $(2n^2 + 6n)$ multiplications, which is almost the same number as that obtained for the one-pass Euclidean algorithm which provides only the forward prediction polynomial, i.e., the backward predictor is given nearly for free in this new algorithm.

B. The Variants of the Berlekamp–Massey Algorithm

A two-pass Berlekamp–Massey algorithm is developed for completing the classical algorithms in both classes, and also for obtaining an algorithm which makes the development of its associated doubling algorithm easier. Observe that the parameter d_r is the first element of the vector shown in the following matrix description:

$$\begin{bmatrix} c_0 & 0 & \cdots & 0 \\ c_1 & c_0 & & \vdots \\ \vdots & & \ddots & 0 \\ c_n & c_{n-1} & \cdots & c_0 \\ c_{n+1} & c_n & & c_1 \\ \vdots & & & \vdots \\ c_{2n} & & \cdots & c_n \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & b_{1,1} & b_{2,1} & & b_{2n,1} \\ & 0 & 0 & b_{3,2} & \\ & & & 0 \quad 0 & \vdots \\ 0 & & \cdots & 0 & b_{2n,n} \end{bmatrix} = \begin{bmatrix} d_0 & 0 & \cdots & & 0 \\ d_{0,1} & d_1 & & & \\ & d_{1,1} & & & \vdots \\ & & \ddots & & \\ & & & d_r & \\ & & & d_{r,1} & \ddots \quad 0 \\ d_{0,2n} & & & d_{r,2n-r} & d_{2n} \end{bmatrix}. \quad (26)$$

Hence, the two-pass algorithm is obtained by computing these parameters via the recursion of the vectors $\boldsymbol{d}_r$, which constitutes the first pass of the algorithm. The recursion relationship is exactly the same one as that for $\boldsymbol{b}_r$ (or $b_r(z)$). The algorithm is described below.

Algorithm 9 (Two-Pass Berlekamp–Massey)

```
First pass:
    $d_0(z) = c(z),\quad dt = 1,\quad d_0'(z) = c(z),\qquad L = 0$
    for $r = 0, 1, \cdots, 2n-1$
        $d_r = d_{r,r}$   (the first element of $d_r(z)$
        if $(d_r = 0)$, then
            $\left\{\begin{bmatrix} d_{r+1}(z) \\ d_{r+1}'(z)\end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & z\end{bmatrix}\begin{bmatrix} d_r(z) \\ d_r'(z)\end{bmatrix}\right\}$
        if $[(d_r \neq 0)$ and $(2L > r)]$, then
            $\left\{\begin{bmatrix} d_{r+1}(z) \\ d_{r+1}'(z)\end{bmatrix} = \begin{bmatrix} 1 & -d_r\,dt^{-1}z \\ 0 & z\end{bmatrix}\begin{bmatrix} d_r(z) \\ d_r'(z)\end{bmatrix}\right\}$
                                                        $[2n-r+1]$
        if $[(d_r \neq 0)$ and $(2L \leq r)]$, then
            $\left\{\begin{bmatrix} d_{r+1}(z) \\ d_{r+1}'(z)\end{bmatrix} = \begin{bmatrix} 1 & -d_r\,dt^{-1}z \\ 0 & 0\end{bmatrix}\begin{bmatrix} d_r(z) \\ d_r'(z)\end{bmatrix}\right.$
            $L = r + 1 - L$
            $\left. dt = d_r \right\}.$                      $[2n-r+1]$
Second pass:
    $b_0(z) = t_0(z) = 1$
    $L = 0,\quad dt = 1$
    for $r = 0, \cdots, 2n-1$
        if $(d_r = 0)$, then
            $\left\{\begin{bmatrix} b_{r+1}(z) \\ t_{r+1}(z)\end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & z\end{bmatrix}\begin{bmatrix} b_r(z) \\ t_r(z)\end{bmatrix}\right\};$
        if $((d_r \neq 0)$ and $(2L > r))$, then
            $\left\{\begin{bmatrix} b_{r+1}(z) \\ t_{r+1}(z)\end{bmatrix} = \begin{bmatrix} 1 & -d_r\,dt^{-1}z \\ 0 & z\end{bmatrix}\begin{bmatrix} b_r(z) \\ t_r(z)\end{bmatrix}\right\}$
                                                        $[L-1]$
        else $((d_r \neq 0)$ and $(2L \leq r))$
            $\left\{ L = r + 1 - L \right.$
            $\begin{bmatrix} b_{r+1}(z) \\ t_{r+1}(z)\end{bmatrix} = \begin{bmatrix} 1 & -d_r\,dt^{-1}z \\ 1 & 0\end{bmatrix}\begin{bmatrix} b_r(z) \\ t_r(z)\end{bmatrix}$
            $\left. dt = d_r \right\}.$                      $[L-1]$
```

It is easily checked that the number of multiplications required for evaluating this algorithm is equal to $3n^2 + 2n$ in the case where $c(z)$ is normal. It is thus of the same order of magnitude as that of the other two-pass algorithms described before.

TABLE III
NUMBERS OF MULTIPLICATIONS REQUIRED FOR EACH ALGORITHM WHERE a/b: FORWARD OR BACKWARD PREDICTOR TO BE COMPUTED AND $a + b$: FORWARD AND BACKWARD PREDICTORS TO BE COMPUTED

	Levinson		Euclidean		Berlekamp	
	One-Pass	Two-Pass	One-Pass	Two-Pass	One-Pass	Two-Pass
a/b	$2n^2$	$3n^2 - 3n$	$2n^2 + 5n$	$3n^2 + 4n$	$2n^2 + 2n$	$3n^2 + 2n$
$a + b$	$2n^2 + n$	$3n^2 - n$	$2n^2 + 6n$	$3n^2 + 3n$	$2n^2 + n$	$3n^2$

As for the Euclidean algorithm, we can also modify a Berlekamp–Massey algorithm for computing both the forward and backward prediction polynomials. The development of this algorithm follows the same lines and will not be repeated. The one-pass version is provided below.

Algorithm 10 (Forward and Backward Berlekamp–Massey)

```
$\begin{bmatrix} a_0(z) \\ b_0(z) \end{bmatrix} \leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
$d_{0,0} = c_0, \quad r = 0$
for $i = 1, \cdots, n$
    $r = r + 1$
    $\bar{d}_{i-1,0} = \sum_{j=0}^{i-1} b_{i-1,j} c_{r-j}$    $[i - 1]$
    $a_i(z) = b_{i-1}(z) - (\bar{d}_{i-1,0}/d_{i-1,0}) z a_{i-1}(z)$    $[i]$
    $r = r + 1$
    $d_{i,0} = \sum_{j=0}^{i} a_{i,j} c_{r-j}$    $[i]$
    $b_i(z) = a_i(z) - (d_{i,0}/\bar{d}_{i-1,0}) z b_{i-1}(z).$    $[i]$
Solution:
    $a(z) = a_n(z).$
```

The number of multiplications required is $(2n^2 + n)$, which is even less than that of the original algorithm, while providing a set of additional forward prediction polynomials $(a_0(z), \cdots, a_n(z))$.

We have thus completed the missing variants for the three classical algorithms. In Table III, we give the number of multiplications for each version of the algorithms described before.

We remark that the algorithms providing both forward and backward predictors have almost the same computational complexity as those providing only the forward or backward predictor. All the algorithms of one-pass type require fewer arithmetic operations than the two-pass algorithms ($2n^2 + O(n)$ versus $3n^2 + O(n)$). Nevertheless, the structure of the two-pass algorithms is fundamental for the development of the doubling algorithms which we will describe in the next section.

Some differences are to be mentioned between the three classical algorithms. First, the Levinson algorithm is the only one which can be simplified for a symmetric matrix, since the other algorithms perform the recursions on partial matrices that are not symmetric, excepted at the very last step. Next, the algorithms will have different finite precision properties since the recursion relationships are related to the inverse of different minors of the Toeplitz matrix (the principal minors for Levinson algorithm, the minors along the second diagonal for Euclidean's (from bottom to top) and Berlekamp's (from top to bottom)), and the condition number of different minors are different. The detailed analysis is out of the scope of this paper.

VI. Doubling Algorithms

All the algorithms that we have discussed so far require a number of multiplications which is proportional to n^2. In this section we show how the divide and conquer strategy [5] can be used to reduce the computational complexity of all these algorithms for large n. Our goal is not to study each algorithm in detail, but to retrieve the essential of the problem, and to provide a unified presentation of the doubling algorithms.

A. A Unified Presentation of the Classical Algorithms

Despite the diversity of the classical algorithms, they have a common structure on which the doubling strategy relies.

First, we note that most known applications of the so-called divide and conquer strategy to the solution of Yule–Walker equations is derived from algorithms of the two-pass type (e.g., the development of the doubling Euclidean algorithm [5]). In the case where they seem to be derived from the one pass version (cf. the recursive Berlekamp–Massey algorithm in [5]), one can observe that it relies implicitly on the corresponding two-pass version: a scalar product in the algorithm can be seen as an element of the result of some polynomial product. The computation of the scalar product is then replaced by the computation of the corresponding polynomial, which can be found in the two-pass version of the algorithm. Hence, in the remainder of this section, we shall concentrate on two-pass versions.

Then, we note that the recursion relationships found in two-pass algorithms are the same ones for the polynomials in the first pass and in the second pass, and can be rewritten in matrix form:

$$\begin{bmatrix} s_i(z) \\ t_i(z) \end{bmatrix} = F_i(z) \begin{bmatrix} s_{i-1}(z) \\ t_{i-1}(z) \end{bmatrix} \qquad (27)$$

with $F_i(z)$ a polynomial matrix of dimensions 2×2. Moreoever, in some algorithms, the recursion in each pass is carried out on three successive terms of a single polynomial recurrence (in which case we have $t_i(z) = s_{i-1}(z)$) while in the other ones (those involving both forward and backward predictors) they involve two successive terms of two polynomial sequences.

Thus, by further noting that one of these recursions involves polynomials of decreasing order (outgoing recursion) while the other one increases the degree of the involved polynomials by 1 at each step (incoming recursion), we have picked up the common structure of the classical algorithms for the development of the corresponding doubling algorithms. The following general presentation of the algorithms summarizes this structure:

```
First pass:
  Initialization of $u_0(z)$ and $v_0(z)$
  For $i = 1, 2, \cdots, n$
    $F_i(z)$ = function of $(u_{i-1}(z), v_{i-1}(z))$
    $\begin{bmatrix} u_i(z) \\ v_i(z) \end{bmatrix} = F_i(z) \begin{bmatrix} u_{i-1}(z) \\ v_{i-1}(z) \end{bmatrix}$
                                        /* outgoing recursion */
Second pass:
  Initialization of $a_0(z)$ and $b_0(z)$
  for $i = 1, 2, \cdots, n$
    $\begin{bmatrix} a_i(z) \\ b_i(z) \end{bmatrix} = F_i(z) \begin{bmatrix} a_{i-1}(z) \\ b_{i-1}(z) \end{bmatrix}$
                                        /* incoming recursion */.
```

Now, this general representation allows us to explain the principle of the doubling algorithms for all these classical algorithms in a unified manner.

B. Application of the Doubling Strategy

We start with the definition of a polynomial matrix of dimensions 2×2:

$$F_{n,1}(z) = \prod_{i=n}^{1} F_i(z). \tag{28}$$

Using this definition, the polynomials $a_i(z)$ and $b_i(z)$ in the general algorithm can be expressed as follows:

$$\begin{bmatrix} a_n(z) \\ b_n(z) \end{bmatrix} = F_{n,1}(z) \begin{bmatrix} a_0(z) \\ b_0(z) \end{bmatrix}. \tag{29}$$

For splitting the algorithm, we suppose that n is even, thus we have:

$$F_{n,1}(z) = F_{n,n/2+1}(z) F_{n/2,1}(z). \tag{30}$$

Then,

$$\begin{bmatrix} a_k(z) \\ b_k(z) \end{bmatrix} = F_{n,n/2+1}(z) \begin{bmatrix} a_{n/2}(z) \\ b_{n/2}(z) \end{bmatrix}. \tag{31}$$

The second pass has been splitted in two parts, the first one being the computation of $a_{n/2}(z)$ and $b_{n/2}(z)$, while the second half provides $F_{n,n/2+1}(z)$.

The second pass depends on the result of the first pass, $F_i(z)$, $i = 1, 2, \cdots, n$, and the computation of these polynomial matrices should be also splitted. For this, we examine the expression of $u_i(z)$ and $v_i(z)$ for $i > n/2$:

$$\begin{bmatrix} u_i(z) \\ v_i(z) \end{bmatrix} = F_{i,n/2+1}(z) F_{n/2,1}(z) \begin{bmatrix} u_0(z) \\ v_0(z) \end{bmatrix} = F_{i,n/2+1}(z) \begin{bmatrix} u_{n/2}(z) \\ v_{n/2}(z) \end{bmatrix} \tag{32}$$

with

$$\begin{bmatrix} u_{n/2}(z) \\ v_{n/2}(z) \end{bmatrix} = F_{n/2,1}(z) \begin{bmatrix} u_0(z) \\ v_0(z) \end{bmatrix}. \tag{33}$$

We note that the recursion is initialized at order $n/2$, by $u_{n/2}(z)$ and $v_{n/2}(z)$. The recursion here is of outgoing type: the order of the polynomials $u_i(z)$ and $v_i(z)$ decrease when i increases. Therefore, the order of $u_{n/2}(z)$ and $v_{n/2}(z)$ are lower than those of $u_0(z)$ and $v_0(z)$. As for $F_i(z)$, $i = 1, 2, \cdots, n/2$, it is easily seen that it does not depend on every coefficient of $u_0(z)$ and $v_0(z)$. In fact, the initial polynomials for the first half are of the same orders as that of $u_{n/2}(z)$ and $v_{n/2}(z)$, which are the initial polynomials of the second half.

Therefore, we have also splitted the second pass. The resulting algorithm is thus entirely splitted. It is summarized below.

The Splitted General Algorithm

```
First half:
  Initialization:
    $\begin{bmatrix} u_0'(z) \\ v_0'(z) \end{bmatrix} \leftarrow \begin{bmatrix} \text{a part of } u_0(z) \\ \text{a part of } v_0(z) \end{bmatrix}$
    $F_{0,1}'(z) \leftarrow I$
  For $i = 1, 2, \cdots, n/2$
    $F_i'(z)$ = function of $(u_{i-1}'(z), v_{i-1}'(z))$
    $\begin{bmatrix} u_i'(z) \\ v_i'(z) \end{bmatrix} = F_i'(z) \begin{bmatrix} u_{i-1}'(z) \\ v_{i-1}'(z) \end{bmatrix}$
    $F_{i,1}'(z) = F_i'(z) F_{i-1,1}'(z)$
  Output: $F_{n/2,1}'(z)$
    $\begin{bmatrix} a_{n/2}(z) \\ b_{n/2}(z) \end{bmatrix} = F_{n/2,1}'(z) \begin{bmatrix} a_0(z) \\ b_0(z) \end{bmatrix}$.
Second half:
  Initialization:
    $\begin{bmatrix} u_0''(z) \\ v_0''(z) \end{bmatrix} \leftarrow F_{n/2,1}'(z) \begin{bmatrix} u_0(z) \\ v_0(z) \end{bmatrix}$
    $F_{0,1}''(z) \leftarrow I$
```

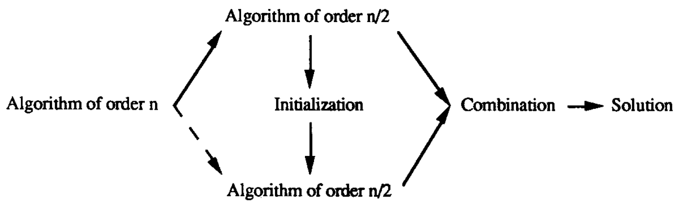


Fig. 2. Organization of the divide and conquer algorithms.

```
For $i = 1, 2, \cdots, n/2$
    $F_i''(z)$ = function of $(u_{i-1}''(z), v_{i-1}''(z))$
    $\begin{bmatrix} u_i''(z) \\ v_i''(z) \end{bmatrix} = F_i'(z) \begin{bmatrix} u_{i-1}''(z) \\ v_{i-1}''(z) \end{bmatrix}$
    $F_{i,1}''(z) = F_i''(z) F_{i-1,1}''(z)$
  Output: $F_{n/2,1}''(z)$
Combination:
    $\begin{bmatrix} a_n(z) \\ b_n(z) \end{bmatrix} = F_{n/2,1}''(z) \begin{bmatrix} a_{n/2}(z) \\ b_{n/2}(z) \end{bmatrix}.$
```

If we compare each half of the splitted algorithm with the initial algorithm, we remark that the latter computes directly the polynomials $a_i(z)$ and $b_i(z)$, while the former computes the polynomial matrices $F_{i,1}'(z)$ (or $F_{i,1}''(z)$), which are of dimension 2×2 and contain 4 polynomials instead of 2. The computational complexity is therefore increased. However, this initial increase in the arithmetic complexity makes the application of the divide and conquer strategy feasible, which allows a large compensation for the additional computations.

Fig. 2 shows the structure of the splitted algorithm.

We remark that the splitted algorithm contains twice the same algorithms of half order, with some additional computations for the initialization of the second half and the combination of the partial results obtained in each half. The number of multiplications required for this algorithm, $M(n)$, follows the relationship below:

$$M(n) = 2M(n/2) + Ms(n) \tag{34}$$

where $Ms(n)$ is the number of multiplications required for the additional computations (initialization and combination).

It is easily seen that the initialization of the second part and the final combination are in fact polynomial products. This type of operation is essentially a linear convolution. Therefore, we can use any fast convolution algorithm, for example, the FFT method [10], [11], and we obtain the computational complexity of the splitted algorithm:

$$Ms(n) = C^{te} n \log_2 n. \tag{35}$$

If we continue this splitting operation for each half of the algorithm, (this is feasible, since the splitted algorithm preserves the same structure as that of the initial one), for $n = 2^v$, we obtain

$$M(n) = A\, n(\log_2 n)^2 \tag{36}$$

with A a given constant.

We have thus succeeded in applying the doubling strategy to the general algorithm, and obtain a doubling algorithm which requires a number of multiplications proportional to $n(\log_2 n)^2$. Since all the classical algorithms belong to the class described in Section VI-A on which the divide and conquer strategy has been applied, this doubling algorithm represents exhaustively their associated doubling algorithms. A precise derivation of these doubling algorithms is easily obtained by applying the strategy explained in Section VI on any of the two-pass algorithms, either computing both forward and backward predictors, or based on a three-term recurrence.

The case of the doubling Levinson/Schur algorithm has been explained with some detail in [1]-[3] by Ammar and Gragg, and is called the "generalized Schur algorithm." In [27], we have developed the algorithm from the above general framework, the resulting algorithm having a more concise presentation, and a lower arithmetic complexity in small dimension cases (we show that the number of multiplications can be reduced already for such small dimensions as $N = 32$). Since the above approach can be applied to any two-pass algorithm, it can also be applied to the split-Levinson algorithm [30], which allows the computational complexity to be further divided by two when the Toeplitz matrix is symmetric.

VII. Conclusion

In this paper, we have provided a general framework for the description of the classical algorithms solving Yule-Walker equations. We provide an exhaustive presentation of these algorithms in terms of Padé approximants, which includes some partial results already published. This presentation allows the obtention of new versions of these algorithms. The arithmetic complexity of all these algorithms is seen to be nearly equivalent inside each of the two classes we have distinguished, and in any case is proportional to n^2. They will nevertheless have different properties in terms of error accumulation.

This unified presentation allows to understand the common structure underlying these algorithms, and we could thus obtain a general description of the associated doubling algorithms, which require a number of multiplications proportional to $n \log_2^2 n$.

References

[1] G. S. Ammar and W. B. Gragg, "Implementation and use of the generalized Schur algorithm," in *Computational and Combinatorial Methods in Systems Theory*, C. I. Byrnes and A. Lindquist, Eds. Amsterdam: North-Holland, 1986, pp. 265-280.

[2] G. S. Ammar and W. B. Gragg, "The generalized Schur algorithm for the superfast solution ot Toeplitz systems," in *Rational Approximation and its Applications in Mathematics and Physics*, J. Gilewicz, M. Pindor, and W. Siemaszko, Eds. (Lecture Notes in Mathematics 1237). New York, Berlin: Springer-Verlag, 1987, pp. 315-330.

[3] G. S. Ammar and W. B. Gragg, "Superfast solution of real positive definite Toeplitz systems," *SIAM J. Matrix Anal. Appl.*, vol. 9, no. 1, pp. 61-76, Jan. 1988.

[4] E. R. Berlekamp, *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.

[5] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, MA: Addison-Wesley, 1985.

[6] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun, "Fast solution of Toeplitz systems of equations and computation of Padé approximations," *J. Algorithms*, vol. 1, pp. 259-295, 1980.

[7] A. Bultheel and P. Dewilde, "On the relation between Pade approximation algorithms and Levinson/Schur recursive methods," in *Signal Processing: Theories and Applications*, M. Kunt and F. de Coulon, Eds. North-Holland, EURASIP, 1980.

[8] J. M. Delosme, "Algorithms for finite Schift-rank processes," Ph.D. dissertation, Stanford Univ., 1982.

[9] P. Delsarte, Y. Genin, and Y. G. Kamp, "A generalization of the Levinson algorithm for Hermitian Toeplitz matrices with any rank profile," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-33, no. 4, Aug. 1985.

[10] P. Duhamel, "Implementation of "split-radix" FFT algorithms for complex, real, and real-symmetric data," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 34, pp. 285-295, 1986.

[11] P. Duhamel and M. Vetterli, "Improved Fourier and Hartley transform algorithms: Application to cyclic convolution of real data," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, no. 6, June 1987.

[12] J. Durbin, "The fitting of time series models," *Rev. Inst. Int. de stat.*, vol. 28, pp. 233-244, 1960.

[13] G. Frobenius, "Uber Relationem Zwischen den Naherungsbruchen von Potenzreihen," *J. fur Math.*, vol. 90, pp. 1-17, 1881.

[14] W. B. Gragg, "The Padé table and its relation to certain algorithms of numerical analysis," *SIAM Rev.*, vol. 14, no. 1, pp. 1-62, 1972.

[15] F. G. Gustavson and D. Y. Y. Yun, "Fast algorithms for rational Hermite approximation and solution of Toeplitz system," *IEEE Trans. Circuits Syst.*, vol. CAS-26, pp. 750-755, Sept. 1979.

[16] F. De Hoog, "A new algorithm for solving Toeplitz systems of equations," *Linear Alg. Appl.*, vol. 88-89, pp. 122-138, 1987.

[17] L. Kronecker, "Zur Theorie der Elimination einer Variabelen aus zwei algebraischen Gleichungen," Monastsb. Konigl. Preuss. Akad. Wiss. Berlin, 1881, pp. 735-600.

[18] N. Levinson, "The Wiener (root mean square) error criterion in filter design and prediction," *J. Math. Phys.*, vol. 25, pp. 261-278, 1947.

[19] J. L. Massey, "Shift-register synthesis and BCH decoding," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 122-127, Jan. 1969.

[20] R. T. Moenck. "Fast computation of GCDs," in *Proc. 5th Annu. ACM Symp. Theory Computing*, 1973, pp. 142-151.

[21] H. Padé, "Sur La Representation Approchée d'une Fonction par des Fractions Rationnelles," Thesis, Ann. Ecole Nor. (3), 9, suppl., pp. 1-93.

[22] Y. Sugiyama, M. Kasahara, S. Hirasawa, and T. Namekawa, "A method for solving key equation for decoding Goppa codes," *Inform. Contr.*, vol. 27, pp. 87-99, Jan. 1975.

[23] Y. Sugiyama, "An algorithm for estimating AR coefficients based upon Euclid algorithm," in *Proc. 1981 Nat. Conf. Inform. Sci. Technol. Inst. Electron. Commun. Eng. Japan*, vol. 10, p. 10. Oct. 1981.

[24] R. P. Brent, "Old and new algorithms for Toeplitz systems," presented at the NATO ASI 1988, Louvent, Belgium.

[25] I. Gohberg, *I. Schur Methods in Operator Theory and Signal Processing*, I. Goberg, Ed. Basel, Boston, Stuttgart: Birkhäusser, 1986.

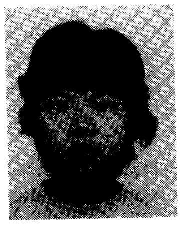
[26] S. Pombra, H. Lev-Ari, and T. Kailath, "Levinson and Schur algorithms for Toeplitz matrices with singular minors," in *Proc. ICASSP 88*, pp. 1643-1646.

[27] H. M. Zhang and P. Duhamel, "Doubling Levinson/Schur algorithm and its implementation," in *Proc. ICASSP 89*, pp. 1115-1118.

[28] C. J. Zarowski, "A Schur version of the Berlekamp-Massey algorithm," *IEEE Trans. Signal Processing*, submitted for publication.

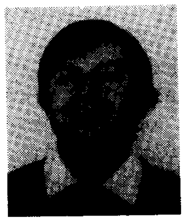
[29] G. Heinig and K. Rost, *Algebraic Methods for Toeplitz-Like Matrices and Operators*. Boston: Birkhauser, 1984.

[30] P. Delsarte and Y. Genin, "The split-Levinson algorithm," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, no. 3, pp. 470-478, June 1986.



Hui-Min Zhang, was born in Fujian, China, in 1964. She received the B.S. degree in electric automation from the University of Textiles of China, Shanghai, in 1984, the Engineer Specialization diploma in signaux, images, and formes from the Ecole Supérieure d'Electricité, Metz, France, in 1986, and the Ph.D. degree in digital signal processing from the Ecole Nationale Supérieure des Télécommunications and the Centre National d'Etudes des Télécommunications, Paris, France, in 1989.

After a year of postdoctoral research at Electricité de France, she is currently working for Matra Ericsson Telecommunications, Massy, France, on several research projects within the Research for Advanced Communication in Europe (RACE).



Pierre Duhamel (M'87-SM'87) was born in France in 1953. He received the Ingenieur degree in electrical engineering from the National Institute for Applied Sciences (INSA), Rennes, France, in 1975, the Dr. Ing. degree in 1978, and the Doctorat es Sciences in 1986, both from Orsay University, France.

From 1975 to 1980, he was with Thomson-CSF, Paris, France, where his research interests were in circuit theory and signal processing, including digital filtering and analog fault diagnosis. In 1980, he joined the National Research Center in Telecommunications (CNET), Issy les Moulineaux, France, where his activities were first concerned with the design of recursive CCD filters. He is now working on fast Fourier transforms and convolution algorithms, and on the application of similar techniques to adaptive filtering, spectral analysis, and wavelet transforms.

Dr. Duhamel is a member of the DSP Committee and was also an Associate Editor for IEEE Transactions on Signal Processing.