A new method for rational approximation and implementation on the IBM3090 supercomputer

Marc Van Barel
Adhemar Bultheel

Report TW 135, July 1990

Department of Computer Science, K.U. Leuven

Abstract

A new method for discrete least squares rational approximation is presented. The algorithm is fast in the sense that it requires order $m\alpha$ computation time where m is the number of data points and α is the complexity of the approximant. A parallel implementation of this algorithm on the IBM3090 supercomputer is described.

Keywords : rational approximation, parallelism, orthogonal polynomials.
AMS(MOS) Classification : Primary : 41A20, Secondary : 65D15, 65W05, 33A65.

A new method for rational approximation and implementation on the IBM3090 supercomputer

Marc Van Barel and Adhemar Bultheel
Department of Computer Science
K.U. Leuven

July 26, 1990

Abstract

A new method for discrete least squares rational approximation is presented. The algorithm is fast in the sense that it requires order $m\alpha$ computation time where m is the number of data points and α is the complexity of the approximant. A parallel implementation of this algorithm on the IBM3090 supercomputer is described.

1 Introduction

In our previous publications [8, 9], we investigated the rational interpolation problem not only for scalar function values but also for the vector case. The matrix case will follow in the near future. The standpoint we took was a theoretical one. We found a parametrization of all solutions of the rational interpolation problem and fast and elegant algorithms to compute this parametrization. Fast means of order m^2 if there are m interpolation data. However, the algorithms which we designed are not numerically stable. This limits their application to theoretical investigation of the rational interpolation problem or to those cases where there are no round-off errors, e.g. when the computations are done within a finite field.

This report has two purposes. First of all, we derive a generalization of the algorithm of Reichel [5] to solve the discrete least squares polynomial approximation problem based on the Rutishauser-Gragg-Harrod algorithm [6, 2]. The generalized algorithm is a new and nice method to compute a rational approximation in a least squares sense to some measured interpolation data. It will turn out that this algorithm is fast, more specifically it is of order $m\alpha$ where α indicates the complexity of the rational approximant. Implementing this algorithm in a pipelined fashion on α processors working in parallel could reduce the computational time to order m. This brings us to the second purpose of this report. We shall describe the parallel implementation of our fast algorithm on the IBM3090 having four processors (with three vector processors). We shall indicate that this parallel architecture is not that usefull to implement our algorithm in parallel. However, we have taken the opportunity to give a short introduction on how to program the IBM3090 in parallel. That is the reason why we have not dropped this parallel implementation from the exposition. It will turn out that when we want to implement our new algorithm efficiently in parallel, we should use a parallel architecture where the synchronization time is low and with a shared memory.

2 The theory behind the algorithm

Let us assume that we have function values in the form f_i/e_i for the points $z_i, i = 1, 2, \ldots, m$. For the moment, all the entities f_i, e_i and z_i belong to a field $\mathbf{F}$. Solving the rational interpolation problem means that we look for a rational function $n(z)/d(z)$ such that this rational function satisfies the m interpolation conditions

$$\frac{n(z_i)}{d(z_i)} = \frac{f_i}{e_i}, i = 1, 2, \ldots, m.$$

It is clear that this problem setting does not completely determine the rational function $n(z)/d(z)$. Indeed, if we take $d(z)$ equal to one and $n(z)$ equal to the interpolating polynomial of degree smaller than or equal to $m-1$ satisfying the m interpolation conditions, we have one of the solutions. However, in the normal case this is not the only solution. There is a solution of the form $n(z)/d(z)$ as long as the sum of the degrees of $n(z)$ and $d(z)$ is equal to $m-1$. Also, in the nonnormal case, we have in general several solutions. For example, think about the solutions lying on the mth antidiagonal in the Newton-Padé table. This indicates that we should restrict the structure of our rational interpolant. For example, we could give all the degrees of freedom to the numerator polynomial $n(z)$ leading to an interpolating polynomial as a special case of a rational interpolant. The structure that we shall adopt in the sequel for the rational interpolant $n(z)/d(z)$ will distribute the degrees of freedom as equally as possible over the numerator and denominator polynomial. In other words, we shall be working in the neighbourhood of the main diagonal in the Newton-Padé table. For the interpolating polynomial of order m, we know that the number of degrees of freedom is equal to the number of interpolation conditions leading to a unique interpolation polynomial. However, taking the number of degrees of freedom equal to the number of interpolation conditions for the rational interpolation problem does not necessarily give a unique solution. Worse, in the nonnormal case it is possible that there is no solution at all. In practice, one wants a rational interpolant of some form and an answer like "the interpolant of the presribed form does not exist" is not good enough. Hence, we have introduced the notion of complexity. For the special case where the degrees of freedom are distributed equally between the numerator and the denominator, we define the complexity or the degree of the rational form $n(z)/d(z)$ as follows.

Definition 2.1 (degree) *The degree of the polynomial couple $(n(z), d(z))$ or the rational form $n(z)/d(z)$ with $(n, d) \neq (0, 0)$ is defined as*

$$\deg\left(n(z), d(z)\right) = \max\left\{\deg n(z), \deg d(z)\right\}.$$

In [8], we have studied the following rational interpolation problem.

Definition 2.2 (rational interpolation problem) *Given the interpolation points z_i with corresponding function values $f_i/e_i, i = 1, 2, \ldots, m$, determine all polynomial couples $(n(z), d(z))$ of minimal degree which satisfy the rational interpolation conditions*

$$\frac{n(z_i)}{d(z_i)} = \frac{f_i}{e_i}, i = 1, 2, \ldots, m.$$

Note that we want to know *all* solutions because even when using this problem setting the solution is not necessarily unique. We have shown in [8] that a nice parametrization can be computed in a very efficient way representing all possible solutions of the rational interpolation problem. The set of all rational interpolants appearing in the Newton-Padé

table forms a subset of all possible rational interpolants solving the rational interpolation problem as defined above (possibly with another distribution of the degrees of freedom between the numerator and denominator polynomial).

From now on, we take the field $\mathbf{F}$ to be the field of complex numbers $\mathbf{C}$. The rational interpolation problem that we looked at above assumes that all the entities $f_i, e_i, z_i, i = 1, 2, \ldots, m$ are known with infinite precision. Of course, this is true if we are working with finite fields but if the function values f_i/e_i are determined by measurements, this is not true at all. In the sequel, we assume that the given values f_i, e_i, z_i are not the exact values but an approximation of the exact interpolation data. Because we do not know the exact interpolation data anymore, satisfying the rational interpolation conditions exactly does not make sense. We have to use another approximation criterion. We shall adopt here a least squares approximation criterion. When we linearize the rational interpolation conditions, we get

$$f_i d(z_i) - e_i n(z_i) = 0, i = 1, 2, \ldots, m.$$

Instead of the exact equality, we have

$$f_i d(z_i) - e_i n(z_i) = r_i \approx 0, i = 1, 2, \ldots, m.$$

Our approximation criterion has to specify in which sense $r_i \approx 0$. First of all, we note that multiplying the numerator $n(z)$ and the denominator polynomial $d(z)$ by the same factor $c \neq 0$, will give the same rational function. However, the residual r_i is also multiplied by this factor c. This indicates that we should normalize the polynomial couple $(n(z), d(z))$ in one way or another. We shall fix the degree of the rational approximant $n(z)/d(z)$ to be $\alpha \ll m$. In other words $\alpha = \max\{\deg n(z), \deg d(z)\} \ll m$. We denote the coefficients of the two polynomials $d(z)$ and $n(z)$ as $n(z) = n_0 + n_1 z + n_2 z^2 + \cdots + n_\alpha z^\alpha$ and $d(z) = d_0 + d_1 z + \cdots + d_\alpha z^\alpha$. The polynomial couple $(n(z), d(z))$ will be normalized such that $n_\alpha = 1$ if $\deg n(z) = \alpha$ and $d_\alpha = 1$ if $\deg n(z) < \alpha$ and $\deg d(z) = \alpha$. Note that we have given preference to the numerator polynomial in the normalization procedure. However, by switching the role of f_i and e_i and of $n(z)$ and $d(z)$, we get the symmetric problem setting giving preference to $d(z)$ in the normalization procedure. In this report we shall solve the following rational approximation problem.

Definition 2.3 (rational approximation problem) *Given the data points z_i with corresponding estimated function values $f_i/e_i, i = 1, 2, \ldots, m$, given the degree α and the weights $w_i > 0, i = 1, 2, \ldots, m$, we look for the normalized rational form $n(z)/d(z)$ of degree α satisfying the following least squares approximation criterion:*

$$\textit{minimize } \operatorname{dist}^2(n, d) = \sum_{i=1}^{m} w_i (e_i n(z_i) - f_i d(z_i))^2 = \sum_{i=1}^{m} w_i r_i^2,$$

where $\operatorname{dist}(n, d)$ denotes the distance between the rational form $(n(z), d(z))$ and the data.

Taking the classical representation for $n(z)$ and $d(z)$ in the basis $1, z, z^2, \ldots, z^\alpha$, we could solve this rational approximation problem as a linear algebra least squares problem. We could also use a generalization of the Lanczos process to solve this problem, i.e. block Lanczos. However, it is well known that this block Lanczos process can cause numerical stability problems and to overcome these, more computational effort is required. Instead of applying these algorithms, we shall generalize a numerically stable method introduced by Gragg et al. [3, 5] to compute a polynomial approximation in the least squares sense. Before we explain

the basic computational scheme, we introduce the following notations

$$e'_i = \sqrt{w_i} e_i \text{ and } f'_i = \sqrt{w_i} f_i,$$

$$\begin{bmatrix} f' & -e' \end{bmatrix} = \begin{bmatrix} \bar{f}'_1 & \bar{f}'_2 & \cdots & \bar{f}'_m \\ -\bar{e}'_1 & -\bar{e}'_2 & \cdots & -\bar{e}'_m \end{bmatrix}^H,$$

$$Z = \mathrm{diag}(z_1, z_2, \ldots, z_m),$$

where we have used the superscript H to denote complex conjugate transpose, and the bar to denote the complex conjugate. The computational scheme starts with three arrays arranged in the following scheme.

	$\begin{bmatrix} f' & -e' \end{bmatrix}^H$
$\begin{bmatrix} f' & -e' \end{bmatrix}$	Z

We have a double column in the SW quadrant, which is reflected as complex conjugate rows in the NE quadrant, and in the SE quadrant we originally find the diagonal matrix of abscissae. We now apply Givens reflections (we could also use Givens rotations) to the left and right of this initial matrix to transform Z into $Q^H Z Q$ with $Q^H Z Q$ block upper Hessenberg (the blocks are two by two). If the points $z_i, i = 1, 2, \ldots, m$ are all real or all purely imaginary, we get

$$\begin{aligned} (Q^H Z Q)^H &= Q^H Z Q, \quad \text{if } z_i \in \mathbf{R}, \\ &= -Q^H Z Q, \quad \text{if } z_i \in j\mathbf{R} \end{aligned}$$

where j represents $\sqrt{-1}$. Therefore, in these two cases $Q^H Z Q$ will be Hermitian block tridiagonal. In the sequel, we shall assume that all data points are real. Hence, $Q^H Z Q$ will be symmetric pentadiagonal which is also two by two block tridiagonal. This will result in an elegant and efficient algorithm solving the rational approximation problem. It is easy to extend the results to the purely imaginary case. When the data points ly on the unit circle, Gragg et al. [3] solve the least squares polynomial approximation problem using a Schur parametrization of the unitary Hessenberg matrix that plays the central role in the theory. We shall treat the unit circle problem in another publication. Let us return to the explanation of the computational scheme. Entries which are not necessarily zero, are denoted by $\times$. By $\odot$, we denote the two entries determining the Givens reflection making one of the two elements equal to zero. The entries which will change when applying this Givens reflection are denoted by $\otimes$. These include the zero entries which could get a nonzero value after the Givens reflection is applied (fill-in). Let us rewrite the initial setup using this new notation where we have already indicated which two elements determine the first Givens reflection that will be applied to the left. In our example, we have chosen $m = 8$.

		$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
		$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
$\odot$	$\times$	$\times$							
$\odot$	$\times$		$\times$						
$\times$	$\times$			$\times$					
$\times$	$\times$				$\times$				
$\times$	$\times$					$\times$			
$\times$	$\times$						$\times$		
$\times$	$\times$							$\times$	
$\times$	$\times$								$\times$

Applying the Givens reflection to the left of this scheme, generates the following fill-in.

		×	×	×	×	×	×	×	×
		×	×	×	×	×	×	×	×
⊗	⊗	⊗	⊗						
0	⊗	⊗	⊗						
×	×			×					
×	×				×				
×	×					×			
×	×						×		
×	×							×	
×	×								×

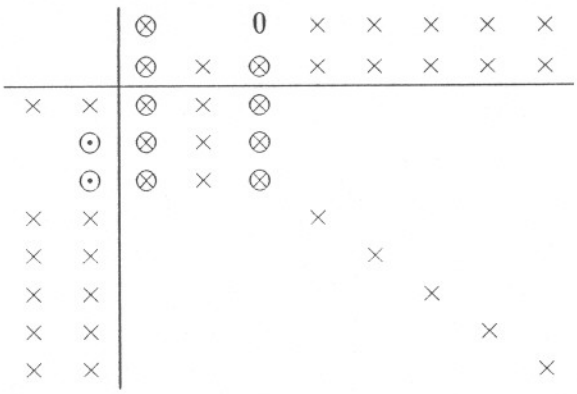
Applying this Givens reflection to the right generates a zero in the top row for reasons of symmetry. In the next scheme, we introduce the fill-in introduced by this operation and we indicate the two elements which will determine the next Givens reflection.

		⊗	0	×	×	×	×	×	×
		⊗	⊗	×	×	×	×	×	×
⊙	×	⊗	⊗						
	×	⊗	⊗						
⊙	×			×					
×	×				×				
×	×					×			
×	×						×		
×	×							×	
×	×								×

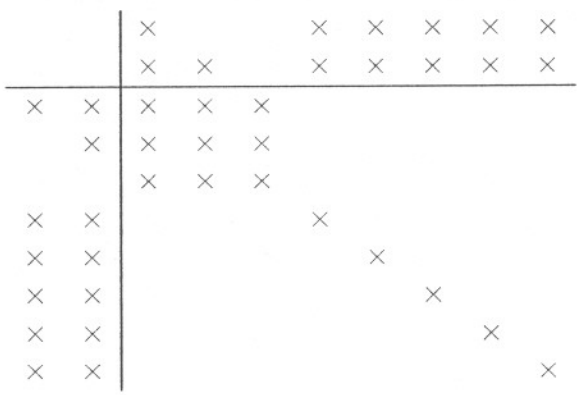
Transforming the third row of the lower block of the computational scheme leads to the following table.

		×		×	×	×	×	×	×
		×	×	×	×	×	×	×	×
⊗	⊗	⊗	⊗	⊗					
	×	×	×						
0	⊗	⊗	⊗	⊗					
×	×				×				
×	×					×			
×	×						×		
×	×							×	
×	×								×

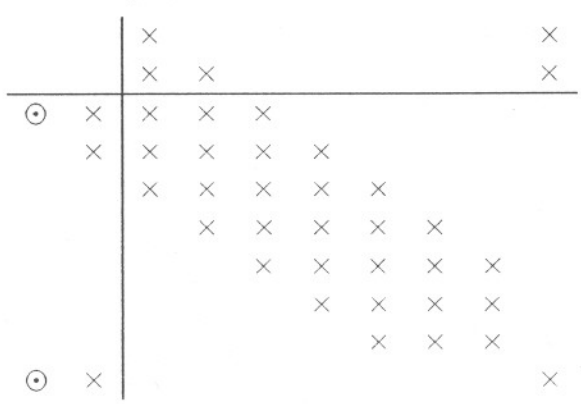
Apply the Givens reflection to the right and indicate the two elements determining the next reflection, and you get the following table.



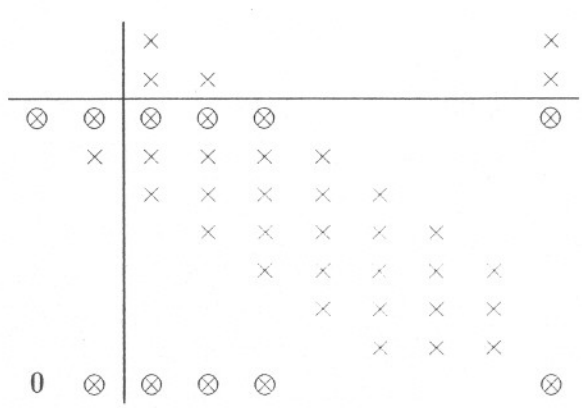
Application of the next Givens reflection to the left and right, leads to the following scheme.



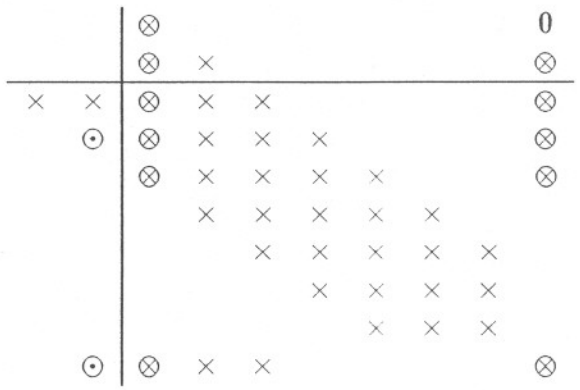
It is clear that the upper left part has become a pentadiagonal matrix. Following the same path of computation for the following rows (and symmetrically for the columns) except for the last one, the scheme eventually becomes as follows.



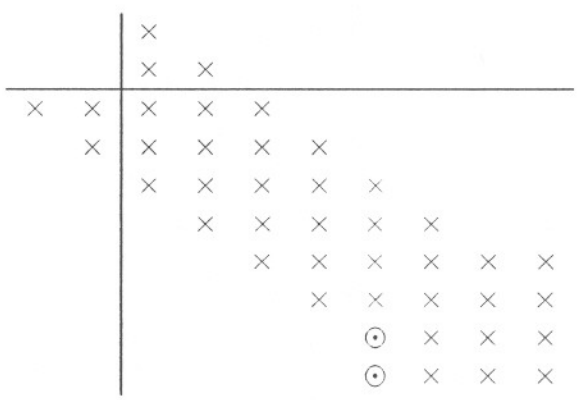
We have already indicated the two elements determining the first Givens reflection to handle the last row (and column) of the table. Applying this Givens reflection to the left, leads us to the following result.



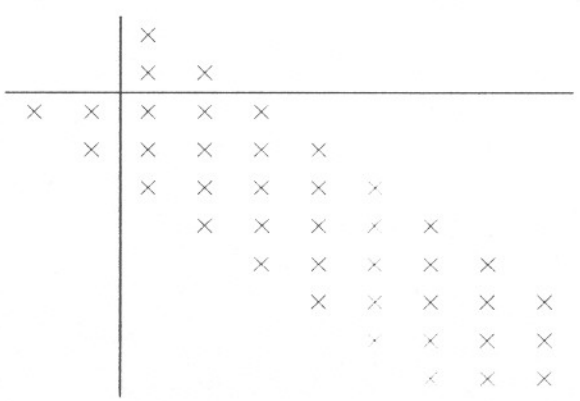
Note that we have shifted the appearance of the first nonzero element of the last row one place to the right by applying the Givens reflection on vectors of length 6. Similarly, by applying the Givens reflection to the right, the appearance of the first nonzero element of the last column will be shifted one place down again by applying the Givens reflection on vectors of length 6. The resulting scheme looks as follows.



In this way we can shift the place of the four nonzero elements of the last row and of the last column in the direction of the diagonal by applying Givens reflections to vectors of length 6. Note that the length of the vectors involved decreases when the diagonal is reached. Let us show the application of the last Givens reflection to the left and right of the computational scheme. Before we apply this reflection, the situation is as follows.



Applying the Givens reflection to the left and right involves vectors of length 4 and results in the following final pentadiagonal scheme.



It is clear that by keeping the sparse structure, the Givens reflections are applied to vectors of maximum length 6. Therefore, the computational work for this part of the algorithm is of the order m^2. Note that the sequence in which these Givens reflections are applied is very important to keep the sparse structure. Hence, we can not use Householder transformations

in this case. Note also that this transformation to a pentadiagonal form can always be carried out. One of the entries determining a Givens reflection could be zero, reducing the Givens reflection to a permutation. Even both entries could be zero. In this case, we could leave the situation unchanged by using the trivial Givens rotation

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

or, when we choose only to use reflections, we could apply the trivial Givens reflection

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Let us investigate the pentadiagonal scheme a little bit further. We divide this scheme in two by two blocks and denote these blocks as follows.

$$\begin{array}{c|c} & B_0^T \\ \hline B_0 & \left[\begin{array}{ccccc} A_1 & B_1^T & & & \\ B_1 & A_2 & \ddots & & \\ & B_2 & \ddots & \ddots & \\ & & \ddots & A_{m'-1} & B_{m'-1}^T \\ & & & B_{m'-1} & A_{m'} \end{array}\right] \end{array}$$

The superscript T denotes the transpose operation. The value of m' is equal to the integer part of $(m+1)/2$. If m is even, all blocks are two by two. However, if m is odd, $A_{m'}$ is a scalar, while $B_{m'-1}$ is a row vector with two components. Note that all blocks $A_j, j = 1, 2, \ldots, m'$ are real symmetric. Without loss of generality, we shall assume in the sequel that m is even. In this case, all $B_j, j = 0, 1, \ldots, m'-1$ are right triangular, also the last one. For the moment we also assume that all B_j are nonsingular. In this case we can define the following block polynomials, which shall play a crucial role in the solution of the rational approximation problem.

Definition 2.4 (polynomials ϕ_j) *The first two block polynomials ϕ_{-1} and ϕ_0 of the sequence are defined as follows:*

$$\phi_{-1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix},$$
$$\phi_0 = B_0^{-1}.$$

The next block polynomials $\phi_j(z), j = 1, 2, 3, \ldots, m'-1$ are defined by the following recurrence relation

$$\phi_j(z) = \phi_{j-1}(z)\left[zI - A_j\right] B_j^{-1} - \phi_{j-2}(z) B_{j-1}^T B_j^{-1}, j = 1, 2, \ldots, m' - 1,$$

based on the previous two block polynomials $\phi_{j-1}(z)$ and $\phi_{j-2}(z)$ and the two by two blocks of the block tridiagonal matrix P

$$P = \text{tridiag}\begin{bmatrix} B_1^T & B_2^T & \cdots & B_{m'-1}^T \\ A_1 & A_2 & \cdots & A_{m'} \\ B_1 & B_2 & \cdots & B_{m'-1} \end{bmatrix}$$

which has the properties described above.

We shall derive some important properties for these block polynomials $\phi_j(z)$.

Property 2.1 (degree, highest degree coefficient) *The block polynomial $\phi_j(z)$, $j = 0, 1, \ldots, m'-1$ has degree j and the highest degree coefficient is equal to the right triangular matrix $B_0^{-1}B_1^{-1}\cdots B_j^{-1}$.*

Proof. The property is certainly true for ϕ_0. From the recurrence relation, an induction proof follows readily. □

The next property will indicate the relationship between the block polynomials ϕ_j and the unitary (orthogonal) matrix Q representing the Givens reflections (rotations) applied to the right of our computational scheme, i.e. $Q^T Z Q = P$ and $Q^T [f', -e'] = \left[B_0^T, 0, \ldots, 0\right]^T$.

Property 2.2 (the values $\phi_j(z_i)$) *The orthogonal matrix Q is directly connected to the function values of the block polynomials ϕ_j in the data points,*

$$Q = D \begin{bmatrix} \phi_0(z_1) & \phi_1(z_1) & \cdots & \phi_{m'-1}(z_1) \\ \phi_0(z_2) & \phi_1(z_2) & \cdots & \phi_{m'-1}(z_2) \\ \vdots & \vdots & & \vdots \\ \phi_0(z_m) & \phi_1(z_m) & \cdots & \phi_{m'-1}(z_m) \end{bmatrix},$$

with D the block diagonal matrix

$$D = \operatorname{diag}\{[f_1', -e_1'], [f_2', -e_2'], \ldots, [f_m', -e_m']\}.$$

Proof. We partition the orthogonal matrix Q in block columns where each block column consists of two simple columns, i.e. $Q = [Q_1, Q_2, \ldots, Q_{m'}]$. From the relation

$$Q^T \begin{bmatrix} f' & -e' \end{bmatrix} = \begin{bmatrix} B_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \phi_0^{-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

it follows that the property is true for the first block column of Q: $Q_1 = [f', -e']\,\phi_0$. To prove the validity of the property for the remaining block columns $Q_j, j = 2, 3, \ldots, m'$, we use the relationship $ZQ = QP$ or

$$z_i Q_{i,j-1} = Q_{i,j-2}B_{j-2}^T + Q_{i,j-1}A_{j-1} + Q_{i,j}B_j, \quad i = 1, 2, \ldots, m,$$

where $Q_{i,j}$ denotes the ith row of the block column Q_j and $Q_0 = 0$. Hence, the jth block column Q_j can be written in function of the two previous block columns as follows

$$Q_{i,j} = Q_{i,j-1}(z_i I - A_{j-1})B_{j-1}^{-1} - Q_{i,j-2}B_{j-2}^T B_{j-1}^{-1}.$$

If we assume that the property is true for the previous block columns, we can prove that the property is also valid for the jth block column by applying the property for the block columns Q_{j-1} and Q_{j-2} resulting in

$$Q_{i,j} = [f_i', -e_i'] \left(\phi_{j-2}(z_i)(z_i I - A_{j-1})B_{j-1}^{-1} - \phi_{j-3}(z_i)B_{j-2}^T B_{j-1}^{-1}\right).$$

Using the recurrence relation for the block polynomials ϕ_j, we obtain that

$$Q_{i,j} = [f_i', -e_i']\,\phi_{j-1}(z_i)$$

and this proves the property by induction on j. □

The key to the solution of the rational approximation problem is the following orthogonality property of the block polynomials ϕ_j.

Property 2.3 (orthonormality) *The block polynomials are orthonormal in the following sense*

$$\sum_{i=1}^{m} \phi_k^T(z_i) \begin{bmatrix} f_i' \\ -e_i' \end{bmatrix} \begin{bmatrix} f_i' & -e_i' \end{bmatrix} \phi_l(z_i) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \text{if } k \neq l$$
$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \text{if } k = l.$$

Proof. From the previous property, we have the following relationship between the function value of the block polynomials ϕ_j in the points z_i

$$Q_{i,j} = [f_i', -e_i'] \, \phi_j(z_i).$$

From the orthogonality of the block columns of Q, the property follows immediately. □

Let us now return to the rational approximation problem. To solve this problem, we shall write the solution in terms of the orthonormal block polynomials ϕ_j. Because the highest degree coefficients of these block polynomials are right triangular, we can write our solution $n(z)/d(z)$ of degree α always as follows

$$\begin{bmatrix} d(z) \\ n(z) \end{bmatrix} = \sum_{j=0}^{\alpha} \phi_j(z) \begin{bmatrix} \mu_j \\ \nu_j \end{bmatrix},$$

with $(\mu_\alpha, \nu_\alpha) \neq (0,0)$. Remember that the solution we are looking for, has to be normalized. From property 2.1, we know that the highest degree coefficient of ϕ_α is equal to

$$B_0^{-1} B_1^{-1} \cdots B_\alpha^{-1} = \begin{bmatrix} 1/\prod_{j=0}^{\alpha} b_{1,j} & \times \\ 0 & 1/\prod_{j=1}^{\alpha} b_{2,j} \end{bmatrix}, \quad \text{if } B_j = \begin{bmatrix} b_{1,j} & b_{3,j} \\ 0 & b_{2,j} \end{bmatrix}. \tag{2.1}$$

The specific value of $\times$ is not important for the moment. To normalize $n(z)/d(z)$, we have to choose $\nu_\alpha = \prod_{j=0}^{\alpha} b_{2,j}$ if $n_\alpha \neq 0$ or $\mu_\alpha = \prod_{j=0}^{\alpha} b_{1,j}$ if $n_\alpha = 0$. To solve the rational approximation problem, we have to minimize

$$\sum_{i=1}^{m} \left(f_i' d(z_i) - e_i' n(z_i) \right)^2.$$

Writing $n(z)$ and $d(z)$ in function of the block polynomials ϕ_j, this is equivalent to minimizing

$$\sum_{i=1}^{m} \left(\sum_{k=0}^{\alpha} \begin{bmatrix} \mu_k & \nu_k \end{bmatrix} \phi_k^T(z_i) \right) \begin{bmatrix} f_i' \\ -e_i' \end{bmatrix} \begin{bmatrix} f_i' & -e_i' \end{bmatrix} \left(\sum_{l=0}^{\alpha} \phi_l(z_i) \begin{bmatrix} \mu_l \\ \nu_l \end{bmatrix} \right).$$

Using the orthogonality property 2.3 for the block polynomials $\phi_j(z)$, this simplifies to minimizing

$$\sum_{k=0}^{\alpha} (\mu_k^2 + \nu_k^2).$$

First of all, we can make this entity smaller by putting μ_k and ν_k equal to zero for $k = 0, 1, \ldots, \alpha - 1$. This reduces the distance of $(n(z), d(z))$ from the data to $\sqrt{\mu_\alpha^2 + \nu_\alpha^2}$. The way in which we have stated the rational approximation problem requires $n(z)/d(z)$ to have degree α. From the normalization conditions, it is clear that we should therefore take the "right" structure, i.e.

$$\nu_\alpha = \prod_{j=0}^{\alpha} b_{2,j} \text{ and } \mu_\alpha = 0, \quad \text{if } \prod_{j=0}^{\alpha} b_{2,j} \leq \prod_{j=0}^{\alpha} b_{1,j}$$

and the "left" structure, i.e.

$$\mu_\alpha = \prod_{j=0}^{\alpha} b_{1,j} \text{ and } \nu_\alpha = 0, \quad \text{if } \prod_{j=0}^{\alpha} b_{2,j} \geq \prod_{j=0}^{\alpha} b_{1,j}$$

where, in case of equality of the two distances, we can choose between the two solutions. Note that this choice corresponds to our problem setting, but that we have the possibility to solve also slightly different problems in an equally simple way. E.g. we could have required $n(z)/d(z)$ to be strictly proper, i.e. with $n_\alpha = 0$ and $d_\alpha = 1$. Then we had to choose the second solution, i.e. the one with $\nu_\alpha = 0$. It is easy to generalize this to more complicated structures than the strictly proper one. Suppose for example that we look for a solution with $n_\alpha = 0$, $n_{\alpha-1} = 0$ and $d_\alpha = 1$. From $n_\alpha = 0$, we easily derive that $\nu_\alpha = 0$. From $d_\alpha = 1$, it follows that $\mu_\alpha = \prod_{j=0}^{\alpha} b_{1,j}$. To make $n_{\alpha-1}$ equal to zero, we have to take $\nu_{\alpha-1}$ equal to a specific value, possibly different from zero. We shall not follow this path in the sequel but concentrate our attention to the original problem setting. The other cases are quite similar. Let us investigate how good the rational approximation is.

Theorem 2.4 *If we write our rational approximation $n(z)/d(z)$ in function of the parameters $\mu_k, \nu_k, k = 0, 1, 2, \ldots, \alpha$, then the approximation error made in the ith point $z_i, i = 1, 2, \ldots, m$ can be measured as follows*

$$f_i' d(z_i) - e_i' n(z_i) = \sum_{k=0}^{\alpha} Q_{i,k+1} \begin{bmatrix} \mu_k \\ \nu_k \end{bmatrix}.$$

Proof. This result follows immediately from the fact that

$$\begin{bmatrix} f_i' & -e_i' \end{bmatrix} \phi_k(z_i) = Q_{i,k+1}.$$

□

For example, if we take the strictly proper solution, all μ_k and ν_k values are zero except one, $\mu_\alpha = \prod_{j=0}^{\alpha} b_{1,j}$. The approximation error in each point z_i can thus be written as

$$f_i' d(z_i) - e_i' n(z_i) = Q_{i,\alpha+1} \begin{bmatrix} \mu_\alpha \\ 0 \end{bmatrix},$$

i.e. the error is given by a scaled version of the first column of $Q_{\alpha+1}$.

Until now, we have taken for granted that all the subdiagonal blocks B_j where nonsingular ($B_{m'-1}$ has full rank). What will happen if one of the right triangular matrices B_j is singular, i.e. if $b_{1,j}$ or $b_{2,j}$ is equal to zero? Without loss of generality, we shall assume that the first zero in the sequence $b_{1,0}, b_{2,0}, b_{1,1}, b_{2,1}, \ldots, b_{1,m'-1}, b_{2,m'-1}$ is denoted by b_α. In this case, B_α looks as follows

$$B_\alpha = \begin{bmatrix} b_{1,\alpha} & b_{3,\alpha} \\ 0 & 0 \end{bmatrix}. \tag{2.2}$$

Because the previous B_j are nonsingular, we can define all block polynomials $\phi_0, \phi_1, \ldots, \phi_{\alpha-1}$ by the recurrence relation. We can not invert B_α. Therefore, we introduce a nonsingular upper right triangular matrix B_α' based on B_α as follows

$$B_\alpha' = \begin{bmatrix} b_{1,\alpha} & b_{3,\alpha} \\ 0 & b \end{bmatrix}$$

with b a free parameter but different from zero. Using this nonsingular matrix B'_α instead of B_α in the recurrence relation, we define ϕ'_α as follows

$$\phi'_\alpha(z)B'_\alpha = \phi_{\alpha-1}(z)(zI - A_\alpha) - \phi_{\alpha-2}(z)B^T_{\alpha-1}.$$

It is clear that ϕ_α has degree α with highest degree coefficient $B_0^{-1}B_1^{-1}\cdots B_{\alpha-1}^{-1}(B'_\alpha)^{-1}$. If we denote the left and right column of ϕ'_α as

$$\phi'_\alpha(z) = \left[\begin{array}{cc} \phi'_{\alpha,l}(z) & \phi'_{\alpha,r}(z) \end{array}\right],$$

we can rewrite the previous equation evaluated in the point z_i multiplied to the left by $[f'_i, -e'_i]$ as follows

$$\left[\begin{array}{cc} f'_i & -e'_i \end{array}\right]\phi'_{\alpha,r}(z_i)\left[\begin{array}{cc} 0 & b \end{array}\right] + Q_{i,\alpha+1}\left[\begin{array}{cc} b_{1,\alpha} & b_{3,\alpha} \\ 0 & 0 \end{array}\right] = Q_{i,\alpha}(zI - A_\alpha) - Q_{i,\alpha-1}B^T_{\alpha-1}, \quad (2.3)$$

using the fact that $Q_{i,j+1} = [f'_i, -e'_i]\,\phi_j(z_i), j = 0, 1, 2, \ldots, \alpha-1$ and the left element of $Q_{i,\alpha+1}$ equals $[f'_i, -e'_i]\,\phi_{\alpha,l}(z_i)$. Because $Q^T Z Q = P$, it is easy to see that equation (2.3) reduces to

$$\left[\begin{array}{cc} f'_i & -e'_i \end{array}\right]\phi'_{\alpha,r}(z_i) = 0.$$

Hence, we immediately conclude that $n'(z)/d'(z)$ with

$$\left[\begin{array}{c} d'(z) \\ n'(z) \end{array}\right] = \phi'_{\alpha,r}(z),$$

approximates the given data exactly, i.e. $n'(z)/d'(z)$ is an interpolant of the given data. It is clear that this situation will appear rarely in practice when the data are only given with a certain finite precision. However, we shall give an example of this situation in one of the next sections. It is clear that we can always normalize the interpolant $n'(z)/d'(z)$ such that $n'_\alpha = 1$ because we know the highest degree coefficient of $\phi'_\alpha(z)$.

Let us investigate the reduction scheme in more detail. Suppose we are computing row and column l of the pentadiagonal matrix P and we have already shifted the nonzero elements $k-1$ places to the right and, $k-1$ places down. In other words, we have the following situation.

$$\begin{array}{ccccccccc}
\ddots & \ddots & \ddots & c_k & & & & & t \\
\ddots & \ddots & \ddots & b_k & \ddots & & & & u \\
 & c_k & b_k & a_k & b_{k+1} & c_{k+2} & & & v \\
 & & \ddots & b_{k+1} & \ddots & \ddots & \ddots & & w \\
 & & & c_{k+2} & \ddots & \ddots & \ddots & \ddots & \\
 & & & & & & & & \\
t & u & v & w & & & & & r
\end{array}$$

We have only indicated those entries playing a role in shifting the lth row one place to the right and the lth column one place down. First of all, we apply the Givens reflection $G_{l,k}$ to the kth and lth row such that

$$G_{l,k}\left[\begin{array}{c} c_k \\ t \end{array}\right] = \left[\begin{array}{c} c''_k \\ 0 \end{array}\right].$$

We obtain the following situation.

$$\begin{array}{ccccccc}
\ddots & \ddots & \ddots & c_k & & & t \\
\ddots & \ddots & \ddots & b_k & \ddots & & u \\
 & c_k'' & b_k'' & a_k' & b_{k+1}'' & c_{k+2}'' & u^\star \\
 & & \ddots & b_{k+1} & \ddots & \ddots & w \\
 & & & c_{k+2} & \ddots & \ddots & \ddots \\
\\
0 & t'' & u' & v'' & w'' & & r'
\end{array}$$

Applying the Givens reflection $G_{l,k}^T = G_{l,k}$ to the kth and lth column gives us the next scheme.

$$\begin{array}{ccccccc}
\ddots & \ddots & \ddots & c_k'' & & & 0 \\
\ddots & \ddots & \ddots & b_k'' & \ddots & & t'' \\
 & c_k'' & b_k'' & a_k'' & b_{k+1}'' & c_{k+2}'' & u'' \\
 & & \ddots & b_{k+1}'' & \ddots & \ddots & v'' \\
 & & & c_{k+2}'' & \ddots & \ddots & \ddots \\
\\
0 & t'' & u'' & v'' & w'' & & r''
\end{array}$$

If we drop the double quotes, we get the notation of the initial situation. If $k+1 < l$, we can increment k and repeat the computational step, i.e. applying a Givens reflection to the left and right. If $k+1 = l$, we handle the next row, if there is one, i.e. we increment l and set k equal to one.

Let us summarize how the entries can be computed.

$$\begin{bmatrix} c_k'' & b_k'' & a_k' & b_{k+1}'' & c_{k+2}'' & u^\star \\ 0 & t'' & u' & v'' & w'' & r' \end{bmatrix} = G_{l,k} \begin{bmatrix} c_k & b_k & a_k & b_{k+1} & c_{k+2} & v \\ t & u & v & w & 0 & r \end{bmatrix}$$

and

$$\begin{bmatrix} a_k'' & u'' \\ u'' & r'' \end{bmatrix} = \begin{bmatrix} a_k' & u^\star \\ u' & r' \end{bmatrix} G_{l,k}.$$

Note that we do not need to compute w'' and c_{k+2}'' if $k = l-2$ or $k = l-1$. If $k = l-1$, we also do not have to compute v'' and b_{k+1}''.

Instead of computing the block orthogonal polynomials $\phi_j(z)$ and then normalizing the corresponding polynomial couples, we are immediately computing the pseudomonic block orthogonal polynomials $\psi_j(z)$, defined as follows

Definition 2.5 (pseudomonic block orthogonal polynomials) *The pseudomonic orthogonal polynomials $\psi_j(z)$ are related to the orthonormal orthogonal polynomials $\phi_j(z)$ (if they exist) in the following way*

$$\psi_j(z) = \phi_j(z) \begin{bmatrix} \prod_{k=1}^{j+1} c_{2k-1} & 0 \\ 0 & \prod_{k=1}^{j+1} c_{2k} \end{bmatrix},$$

where c_{2k-1} and c_{2k} are the diagonal elements of B_{k-1}, i.e. $c_{2k-1} = b_{1,k-1}$ and $c_{2k} = b_{2,k-1}$ from (2.1). In other words, the highest degree coefficient of $\psi_j(z)$ is a unit upper triangular matrix.

The pseudomonic orthogonal polynomials satisfy the recurrence given in the following theorem.

Theorem 2.5 *The pseudomonic block orthogonal polynomials $\psi_j(z)$ can be defined by the following recurrence relation*

$$\psi_j(z) = \psi_{j-1}(z)(zD_j - E_j) - \psi_{j-2}(z)F_j, \quad j = 1, 2, \ldots, m' - 1$$

with the initialization

$$\psi_{-1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ and } \psi_0 = \begin{bmatrix} 1 & -b_1/c_1 \\ 0 & 1 \end{bmatrix}.$$

The matrices D_j, E_j, F_j appearing in the recurrence relation are defined as follows

$$D_j = \begin{bmatrix} 1 & d_j q_j \\ 0 & 1 \end{bmatrix},$$

$$E_j = \begin{bmatrix} a_{2j-1} & (a_{2j-1}d_j + b_{2j})q_j \\ b_{2j}/q_j & b_{2j}d_j + a_{2j} \end{bmatrix},$$

$$F_j = \begin{bmatrix} e_j & d_j e_j q_j \\ f_j/q_j & d_j f_j + g_j \end{bmatrix}$$

with

$$d_j = -b_{2j+1}/c_{2j+1}, \quad e_j = c_{2j-1}^2, \quad f_j = b_{2j-1}c_{2j}, \quad g_j = c_{2j}^2 \text{ and } q_j = \prod_{k=1}^{j} c_{2k} / \prod_{k=1}^{j} c_{2k-1}.$$

Proof. It is easy to prove this by induction using the recurrence relation for the $\phi_j(z)$ and for the $\psi_j(z)$, knowing that

$$A_j = \begin{bmatrix} a_{2j-1} & b_{2j} \\ b_{2j} & a_{2j} \end{bmatrix} \text{ and } B_j = \begin{bmatrix} c_{2j+1} & b_{2j+1} \\ 0 & c_{2j+2} \end{bmatrix}.$$

□

Note that we have included "if they exist" in the definition. The value of c_{2j+1} does not play a role when computing the left column of $\psi_j(z)$. Similarly the value of c_{2j+2} does not appear when deriving the right column of $\psi_j(z)$. Therefore, the pseudomonic orthogonal polynomials $\psi_j(z)$ can be defined based on $\phi_j(z)$ taking the limit when either c_{2j+1} or c_{2j+2} is zero, i.e. when we have a rational interpolant. Using the recurrence relation for the pseudomonic block orthogonal polynomials $\psi_j(z)$, we can easily write

$$\begin{bmatrix} \psi_{\beta-1}(z) & \psi_\beta(z) \end{bmatrix} = \begin{bmatrix} \psi_{-1}(z) & \psi_0(z) \end{bmatrix} V_1 V_2 \cdots V_\beta$$

with

$$V_j = \begin{bmatrix} 0 & -F_j \\ I & zD_j - E_j \end{bmatrix}.$$

To compute the coefficients as well as to derive function values of the rational approximant, we can use a forward or a backward recurrence scheme. We shall concentrate our efforts on

the backward recurrence scheme. If we have chosen for the "left" structure, we initialize our solution σ_β as

$$\sigma_{\beta+1} = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^T,$$

and, if we have chosen for the "right" structure, we start with

$$\sigma_{\beta+1} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T.$$

With this initialization, we use the following backward recurrence scheme

$$\sigma_j = V_j \sigma_{j+1}, j = \beta, \beta - 1, \ldots, 1$$

with final step

$$\begin{bmatrix} d(z) \\ n(z) \end{bmatrix} = \begin{bmatrix} \psi_{-1} & \psi_0 \end{bmatrix} \sigma_1.$$

3 Parallel implementation

Considering the amount of computational work, we can make a distinction between the computation of the block tridiagonal matrix P from the data and the computation of the orthonormal block polynomials. Fixing the degree $\alpha \ll m$ of the rational approximant, implies that we can obtain the rational approximant from $\phi_\alpha(z)$. Hence, it is not necessary to compute the complete pentadiagonal matrix P but only the first part consisting of $A_1, A_2, \ldots, A_\alpha$ and $B_1, B_2, \ldots, B_\alpha$. Compared to the complexity m^2 of computing the complete block tridiagonal matrix P, this decreases the amount of computational work to $m\alpha$. The second part of the solution method requires the computation of $\phi_0, \phi_1, \ldots, \phi_\alpha$, which requires order α^2 floating point operations. Therefore, since $\alpha \ll m$, it is important to decrease the execution time of the first part by parallel implementation. We have tried several scenarios. For the divide and conquer approach, we could not find an efficient way to join the solutions of two subproblems. The parallel approach, that we shall implement here, is a sort of pipeline method.

Let us assume now that we have $np < m$ processors in the reduction scheme of the previous section. The nth processor will handle rows $n + 1 + j \cdot np, j = 0, 1, \ldots, [(m - n - 1)/np]$ where $[\cdot]$ denotes the integer part. These np processors are synchronized in a pipeline manner. Suppose the nth processor is executing the Givens reflection to the left and right on the lth and kth row and column by using the formulas above. As soon as c_k'' is computed the $(n + 1)$st processor could use this value for handling the $(l + 1)$st and $(k - 2)$nd row and column. Therefore, the nth processor has to compute c_k'' as soon as possible while the $(n + 1)$st processor has to delay the use of c_k'' as much as possible. We have not used this fine tuned synchronization in the implementation. Instead, we let the $(n + 1)$st processor start on row and column $k + 2$ as soon as the nth processor has computed the value c_k''. The influence on the execution time can be seen as a difference in startup time of the pipeline. Neglecting the synchronization time, we thus obtain a reduction of the computation time by a factor np when using np processors. Because we only have to know $A_1, A_2, \ldots, A_\alpha$ and $B_0, B_1, B_2, \ldots, B_\alpha$, it is not necessary to shift each row and column completely to the diagonal. From the $(\alpha + 2)$nd block row on, we have to shift only over $(\alpha + 1)$ block places. The elements $c_1, c_2, \ldots, c_{2\alpha+2}$ will determine the approximation error, i.e. the distance between the "left" and "right" structure and the data. After the execution of the first step of the algorithm, we shall print the products $\prod_{j=1}^{\beta+1} c_{2j-1}$ and $\prod_{j=1}^{\beta+1} c_{2j}$ for $\beta = 0, 1, 2, \ldots, \alpha$. Given these approximation errors, the user chooses an approximation degree $\beta \leq \alpha$ and the "left"

or "right" structure for the approximant. As we have shown in the previous section, the first zero entry in this sequence of approximation errors will indicate a rational interpolant. In practice, however, a value exactly equal to zero as in (2.2) will never be attained.

We have implemented our algorithm on the IBM3090 architecture using parallel FORTRAN, introduced in appendix A. Because a lot of mapping has to be done before the computational work is scheduled onto real processors, this machine requires coarse-grained parallelism and is therefore not suited to execute our fine-grained parallel algorithm. However, to check if the pipelined version of our algorithm works, we have implemented it on the IBM3090.

Appendix B contains the parallel FORTRAN program together with some practical information on compilation, linking and input-output redirection under the CMS operating system environment. In this FORTRAN program, we have given preference to $d(z)$ in the normalization procedure. Besides using weights, it is possible to scale the interpolation data such that $\max\{\text{abs}(f_i), \text{abs}(e_i)\} = 1$. For the examples which we tried, the scaling did not improve the results. However, it is kept in the source code for completeness.

For the sake of completeness, we give a small introduction on the use of timers in appendix C.

4 Examples

In this section, we give two examples. The first one is rather theoretical, the second one a little bit more practical. To measure the goodness of the rational approximant, we consider a relative and absolute approximation error in each point z_i. If $\text{abs}(e_i) \geq \text{abs}(f_i)$, the absolute error is defined as $abserr(z_i) = \text{abs}((n(z_i)/d(z_i)) - (f_i/e_i))$ (if $d(z_i) \neq 0$) and the relative error as $relerr(z_i) = \text{abs}(abserr(z_i)/(f_i/e_i)$ (if $f_i/e_i \neq 0$). If $\text{abs}(e_i) < \text{abs}(f_i)$, the absolute error is defined as $abserr(z_i) = \text{abs}((d(z_i)/n(z_i)) - (e_i/f_i))$ (if $n(z_i) \neq 0$) and the relative error as $relerr(z_i) = \text{abs}(abserr(z_i)/(e_i/f_i)$ (if $e_i/f_i \neq 0$). The value of the rational approximant $n(z_i)/d(z_i)$ or the inverse is computed using the Horner scheme. Of course, we could also use the backward recurrence where z is replaced by z_i. All computations were done in double precision.

Example 4.1 In the first example, we consider the 10 interpolation points $z_i = i, i = 1, 2, \ldots, 10$ with function values f_i/e_i, where $f_i = i + 1$ and $e_i = i + 2$. The weights are set equal to one. We assume all the quantities that are given as input to the parallel FORTRAN program are known within machine precision. With α chosen as big as possible, i.e. equal to 4, we get the following table of approximation errors.

j	$\prod_{k=0}^{j} c_{2k+1}$	$\prod_{k=0}^{j} c_{2k+2}$
0	0.2540D+02	0.1131D+01
1	0.2298D+02	-0.1513D-13
2	0.4326D+02	-0.2085D-13
3	-0.8152D+02	0.1284D-13
4	0.1084D+03	-0.1756D-13

From this table, it is clear that we should choose β equal to 1 and take the "right" structure for the rational approximant. The next table shows the coefficients of the numerator and denominator polynomial of the rational approximant.

degree	numerator coefficients	denominator coefficients
0	0.9999999999999806D+00	0.1999999999999972D+01
1	0.9999999999999993D+00	0.1000000000000000D+01

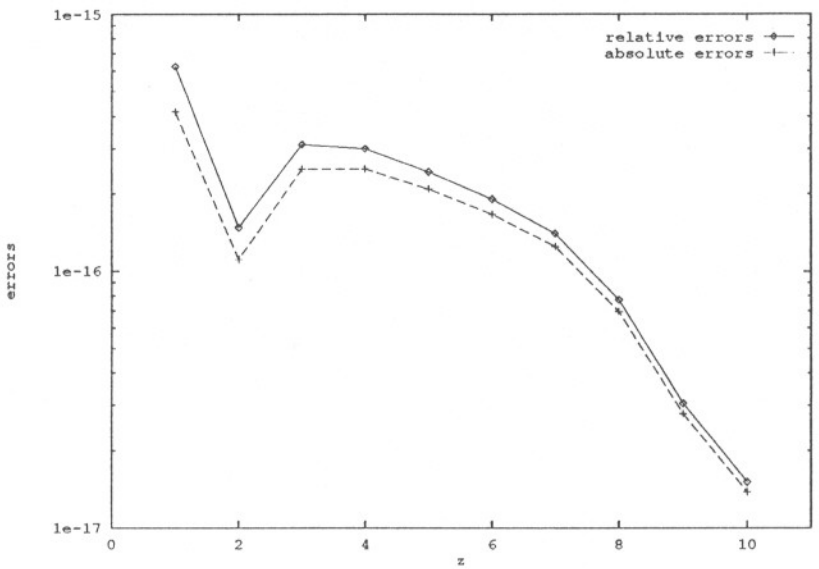


Figure 1: The absolute and relative errors of example 1

The absolute and relative error are indicated in Figure 1. We can say that the rational approximant, we obtained here, is very good, which, for this example of course will not come as a surprise.

A more realistic example is the following one.

Example 4.2 We are starting from the rational function

$$g(z) = \frac{100z^4 + 10520z^3 + 52101z^2 + 10105z + 500}{8169.13z^3 + 50664.97z^2 + 9984.32z + 500}.$$

The inverse of this function was used to demonstrate model reduction techniques [1, 7]. The data points are $z_i = i/10, i = 1, 2, \ldots, 100$. The function values are given by $f_i = g(z_i)$, computed by the algorithm of Horner, and $e_i = 1.0$. The weights are taken equal to one. For $\alpha = 10$, we get the following table of approximation errors.

j	$\prod_{k=0}^{j} c_{2k+1}$	$\prod_{k=0}^{j} c_{2k+2}$
0	0.1000D+02	0.6786D+00
1	0.3736D+01	0.3058D+00
2	0.1824D+01	0.4317D+00
3	-0.7181D-06	0.8036D+00
4	-0.1084D-09	0.1942D+01
5	-0.1142D-09	0.4571D+01
6	-0.1957D-09	0.1137D+02
7	-0.3880D-09	0.2790D+02
8	-0.9079D-09	0.6308D+02
9	-0.1937D-08	0.1600D+03
10	-0.4190D-08	0.3977D+03

From these rational approximation errors, it is clear that we should use a "left" structure. First of all, we choose β equal to 4. The coefficients of the rational approximant are given in the next table.

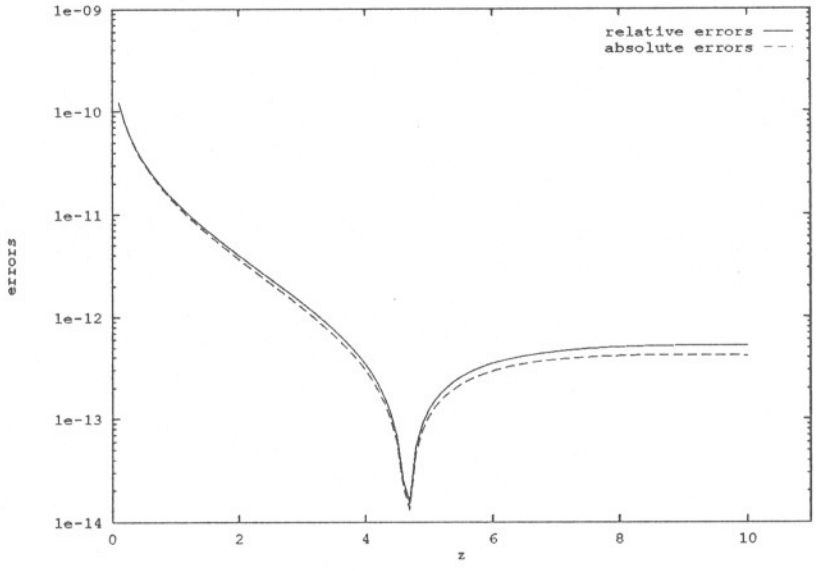


Figure 2: The absolute and relative errors of example 2 with $\beta = 4$

degree	numerator coefficients	denominator coefficients
0	0.4999659362429158D+01	0.4999659360979468D+01
1	0.1010465049127602D+03	0.9983978711954701D+02
2	0.5210092803518180D+03	0.5066491406348295D+03
3	0.1051999931526788D+03	0.8169129999998811D+02
4	0.1000000000000000D+01	

The absolute and relative approximation errors in each data point are shown in Figure 2. If we choose β equal to 3, we get the following table of coefficients.

degree	numerator coefficients	denominator coefficients
0	0.4974901332730807D+02	0.4974902323831709D+02
1	0.5104473551313152D+03	0.4984396214923242D+03
2	0.1050995010534305D+03	0.8169130146942339D+02
3	0.1000000000000000D+01	

Figure 3 indicates the absolute and relative error of this rational approximant in each data point. Considering the data points in reverse order gives the same results. As a conclusion, we can say that the original rational function $g(z)$ can be approximated very well by a rational approximant of degree 3 as was already known in the model reduction context.

5 Conclusion and future research

We have developed an elegant and efficient method to solve the discrete least squares rational approximation problem. We have shown that this method can be implemented in a pipelined way on a parallel architecture. Because of the large synchronization time, the IBM3090 parallel architecture is less suited for this purpose. However, the FORTRAN source code is included here because we also wanted to give an introduction to this architecture.

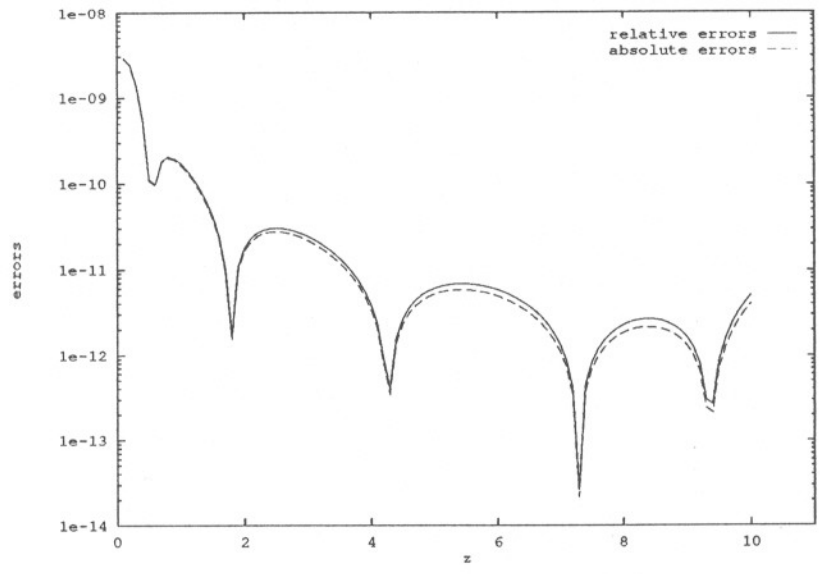


Figure 3: The absolute and relative errors of example 2 with $\beta = 3$

Because the algorithm developed here is so efficient, it could be used as the kernel of a graphical interactive program to design rational approximants. As domains of application, we mention digital filter design, modal analysis, model reduction, system identification,....

To enhance numerical stability, we have already mentioned that the backward recurrence scheme can also be used to evaluate the rational approximant. A compromise between using the coefficients for the classical basis $1, z, z^2, \ldots$ and the backward recurrence scheme, would be to compute the coefficients of the numerator and denominator polynomial of the rational approximant with respect to a more "orthogonal" basis on the approximation interval, e.g. with respect to the Chebyshev polynomials. This is easy as long as the multiplication by z (the shift operation) of such a basis polynomial can be done in an efficient way. Note that we can do this efficiently for orthogonal polynomials using the three term recurrence relation.

In future research, we shall investigate the numerical stability, e.g. the use of orthogonal polynomials. We shall see how the weights can be chosen in a good way and how the scaling of the independent variable z influences the results. We shall investigate the connection between the linearized approximation errors (the residuals) and the proper rational approximation errors and the influence of the weights hereupon. We shall extend the results to the unit circle.

To summarize, we have constructed a method, which is so efficient that it can be used interactively on large data sets.

References

[1] R. K. Appiah. Linear model reduction using Hurwitz polynomial approximation. *Int. J. Contr.*, 28(3):467–488, 1978.

[2] W. B. Gragg and W. J. Harrod. The numerically stable reconstruction of Jacobi matrices from spectral data. *Numer. Math.*, 44:317–335, 1984.

[3] W. B. Gragg, L. Reichel, and G.S. Ammar. Discrete least squares approximation by trigonometric polynomials. *Math. Comp.*, 1990. submitted.

[4] C. Pitts H. Caffey and M. Pottle. *Parallel Processing.* Cornell National Supercomputer Facility, Cornell University, Ithaca, New York, October 1988.

[5] L. Reichel. Fast QR decomposition of Vandermonde-like matrices and polynomial least squares approximation. Technical Report 89-5, Department of Mathematics, M.I.T., 1989.

[6] H. Rutishauser. On Jacobi rotation patterns. In *Proceedings of Symposia in Applied Mathematics, vol. 15, Experimental Arithmetic, High Speed Computing and Mathematics*, pages 219–239, Providence, 1963. Amer. Math. Society.

[7] Y. Shamash. Viability of methods for generating stable reduced order models. *IEEE Trans. on Automatic Control*, AC-26(6):1285–1286, December 1981.

[8] M. Van Barel and A. Bultheel. A new approach to the rational interpolation problem. *J. Comp. Appl. Math.*, September 1990.

[9] M. Van Barel and A. Bultheel. A new approach to the rational interpolation problem: the vector case. Technical Report TW130, Department of Computer Science, K.U. Leuven, May 1990.

A The IBM3090 supercomputer of the University Computing Center

The IBM3090 supercomputer which is standing in the University Computing Center of the Katholieke Universiteit Leuven is a shared memory multiple instruction multiple data (MIMD) machine with four processors of which three processors have a vector processing facility.

We have coded our program using the parallel FORTRAN based on VS FORTRAN version 2.1.1. This parallel FORTRAN has been extended in three ways:

- Compiler extensions: The compiler can automatically determine that eligible DO-loops are executed in parallel.
- Language extensions:
  - PARALLEL LOOP : To execute iterations of loops in parallel.
  - PARALLEL CASES : To execute sections of code in parallel.
  - Parallel Tasks : To execute subroutines in parallel.
- Library extensions: For synchronization, the user can apply "locks", "events", "traces".

For more information, we refer to [4]. We have implemented our algorithm using parallel tasks. These tasks are executed by FORTRAN processors, which are logical simulations of real processors that can execute parallel FORTRAN tasks. These FORTRAN processors are implemented as CMS virtual processors. These CMS virtual processors are scheduled onto real processors by the operating system. Because a lot of mapping has to be done, it is clear that the amount of computational work in each FORTRAN task has to be big

(coarse-grained parallelism). Otherwise, the synchronization, mapping and scheduling time will outweigh the computational time. The parallel tasks of our pipelined method consist in executing a Givens reflection twice on vectors of length six (fine-grained parallelism). Therefore, the synchronization, mapping and scheduling time will be much bigger than the real computational time on this architecture. Because the vectors involved have length six, also the vector processing facility can not be used efficiently due to the start-up time of the vector pipeline. We have implemented our algorithm in parallel on the IBM architecture to check if it worked in a pipelined fashion. We have come to the point where the algorithm could be implemented using dedicated hardware. We could have used an hypercube architecture with local memory connected to each processor. However, the communication of the elements of the vectors involved would outweigh the real computational work. Therefore, a shared memory architecture is advisible unless the communication time is very low compared to the computational effort. Let us focus our attention now on the FORTRAN source code.

B The FORTRAN source code

```
      PROGRAM MAIN
      PARAMETER ( N = 6, M = 1000, MALPHA = 100 )
      INTEGER ITASK(N), NP, IFREE(N), ILOW(N), IHIGH(N), MP
      INTEGER IT, IP, KTASK, COUNT, II(N), ALPHA, IALPHA, BETA
      REAL*8 A(M), B(M), C(M), T(N), U(N), V(N), W(N), R(N)
      REAL*8 Z(M), EP(M), FP(M), PBPRIM(0:MALPHA), PB(0:MALPHA)
      REAL*8 SIGMA(0:MALPHA, 4), NUM, DEN, FUNVAL, ABSERR, RELERR
      REAL*8 WEIGHT(M), AUX, F(M), E(M), HORNER
      CHARACTER*1 CH
      COMMON /ABC/ A, B, C, T, U, V, W, R, IFREE, ILOW, IHIGH
      COMMON /ZFE/ Z, FP, EP, NP, MP, IALPHA, PBPRIM, PB
C     BEGIN
C        READ THE DATA
         PRINT*, 'THE NUMBER OF INTERPOLATION DATA ?'
         READ*, MP
         PRINT*, 'THE INTERPOLATION DATA Z, F, E, WEIGHT ?'
         DO 5 I = 1, MP
            READ*, Z(I), F(I), E(I), WEIGHT(I)
            PRINT*, Z(I), F(I), E(I), WEIGHT(I)
            AUX = DSQRT( WEIGHT(I) )
            FP(I) = AUX * F(I)
            EP(I) = AUX * E(I)
 5       CONTINUE
C        POSSIBLE SCALING OF THE DATA
         PRINT*, 'SCALING ? ("Y" OR "N")'
         READ*, CH
         IF ( CH .EQ. 'Y' ) THEN
            DO 7 I = 1, MP
               AUX = MAX ( ABS(FP(I)), ABS(EP(I)) )
               FP(I) = FP(I) / AUX
               EP(I) = EP(I) / AUX
 7          CONTINUE
```

```
         ENDIF
C        COMPUTE ONLY PART OF THE PENTADIAGONAL MATRIX
C        DETERMINED BY ALPHA
         PRINT*, 'ALPHA ?'
         READ*, ALPHA
         PRINT*, 'LEFT("L") OR RIGHT("R") ?'
         READ*, CH
         IF ( CH .EQ. 'L' ) THEN
            IALPHA = 2 * ALPHA + 2
         ELSE
            IALPHA = 2 * ALPHA + 3
         ENDIF
C        DETERMINE THE NUMBER OF PROCESSORS
         NP = NPROCS()
         PRINT*, 'THE NUMBER OF PROCESSORS IS ', NP
C        ORIGINATE THE TASKS
         DO 10 I = 1, NP
            ORIGINATE ANY TASK ITASK(I)
 10      CONTINUE
C        INITIALIZE THE VARIABLES COUPLED WITH EACH TASK
         DO 15 I = 1, NP
            II(I) = I
            SCHEDULE TASK ITASK(I), SHARING(ABC,ZFE),
     *         CALLING INIT(II(I))
 15      CONTINUE
         A(1) = Z(1)
         B(1) = -FP(1)
         C(1) = EP(1)
         COUNT = 0.D0
C        MAIN LOOP
 150     IF ( COUNT .LT. NP ) THEN
            WAIT FOR ANY TASK KTASK
            DO 100 I = 1, NP
               IF ( ITASK(I) .EQ. KTASK ) IT = I
 100        CONTINUE
            II(IT) = IT
C           DETERMINE THE PREVIOUS PROCESSOR
            IP = MOD ( IT+1, NP ) + 1
            IF ( IHIGH(IT) .GT. MP ) THEN
C              PROCESSOR IT HAS COMPLETELY FINISHED
               COUNT = COUNT + 1
            ELSE
               IF ( IHIGH(IT) .LE. IHIGH(IP) ) THEN
C                 THE PREVIOUS PROCESSOR IS ALREADY WORKING ON
C                 A SUBSEQUENT ROW
                  SCHEDULE TASK KTASK, SHARING(ABC,ZFE),
     *               CALLING GIVENS( II(IT) )
               ELSE IF ( ILOW(IT) .LE. IFREE(IP) ) THEN
C                    THE PREVIOUS PROCESSOR IS WORKING ON
```

```
C                     A PREVIOUS ROW BUT HAS GIVEN ROW ILOW(IT)
C                     FREE
                      SCHEDULE TASK KTASK, SHARING(ABC,ZFE),
     *                 CALLING GIVENS( II(IT) )
               ELSE
C                 THE PROCESSOR HAS TO WAIT UNTIL ROW
C                 ILOW(IT) HAS BEEN GIVEN FREE BY THE
C                 PREVIOUS PROCESSOR
                  SCHEDULE TASK KTASK, CALLING NOWORK
               ENDIF
            ENDIF
            GOTO 150
         ENDIF
         DO 300 I = 1, NP
            TERMINATE TASK ITASK(I)
 300     CONTINUE
C        PRINT THE PENTADIAGONAL ELEMENTS
         PRINT*,'I, A, B, C'
         DO 200 I = 1, MP
            PRINT 1000, I, A(I), B(I), C(I)
 200     CONTINUE
 1000    FORMAT(I4,3D18.10)
C        COMPUTE AND PRINT THE RATIONAL APPROXIMATION ERROR
         CALL APPERR( ALPHA )
C        GIVE THE DEGREE AND TYPE OF RATIONAL APPROXIMANT
         PRINT*, 'BETA ?'
         READ*, BETA
         PRINT*, 'LEFT("L") OR RIGHT("R") ?'
         READ*, CH
C        COMPUTE THIS RATIONAL APPROXIMANT
         CALL RATAPP( BETA, CH, SIGMA )
C        PRINTING THE COEFFICIENTS OF THE RATIONAL APPROXIMANT
         PRINT*, 'THE NUMERATOR COEFFICIENTS'
         PRINT*, 'DEGREE, COEFFICIENT'
         DO 400 I = 0, BETA
            PRINT1010, I, SIGMA(I, 3)
 400     CONTINUE
 1010    FORMAT(I4,D24.16)
         PRINT*, 'THE DENOMINATOR COEFFICIENTS'
         PRINT*, 'DEGREE, COEFFICIENT'
         DO 450 I = 0, BETA
            PRINT1010, I, SIGMA(I,4)
 450     CONTINUE
C        COMPUTE AND PRINT THE ABSOLUTE AND RELATIVE ERRORS
         DO 500 I = 1, MP
            NUM = HORNER ( SIGMA(0,3), BETA, Z(I) )
            DEN = HORNER ( SIGMA(0,4), BETA, Z(I) )
            IF ( ABS(E(I)) .GE. ABS(F(I)) ) THEN
               IF ( DEN .EQ. 0.D0 ) THEN
```

```
                  PRINT1040, I, Z(I), F(I), E(I), NUM, DEN, 'INFINITY'
               ELSE
                  FUNVAL = F(I) / E(I)
                  ABSERR = ( NUM/DEN ) - FUNVAL
                  IF ( FUNVAL .NE. 0.D0 ) THEN
                     RELERR = ABSERR / FUNVAL
                     PRINT1020, I, Z(I), F(I), E(I), NUM, DEN,
     *                           ABSERR, RELERR
 1020                FORMAT(I4,7D12.4)
                  ELSE
                     PRINT1030, I, Z(I), F(I), E(I), NUM, DEN,
     *                           ABSERR, 'INF'
 1030                FORMAT(I4,6D12.4,A)
                  ENDIF
               ENDIF
            ELSE
               IF ( NUM .EQ. 0.D0 ) THEN
                  PRINT1040, I, Z(I), F(I), E(I), NUM, DEN, 'INFINITY'
 1040             FORMAT(I4,5D12.4,A)
               ELSE
                  FUNVAL = E(I) / F(I)
                  ABSERR = ( DEN/NUM ) - FUNVAL
                  IF ( FUNVAL .NE. 0.D0 ) THEN
                     RELERR = ABSERR / FUNVAL
                     PRINT1020, I, Z(I), F(I), E(I), NUM, DEN,
     *                           ABSERR, RELERR
                  ELSE
                     PRINT1030, I, Z(I), F(I), E(I), NUM, DEN,
     *                           ABSERR, 'INF'
                  ENDIF
               ENDIF
            ENDIF
 500     CONTINUE
         RETURN
      END

      SUBROUTINE INIT(I)
      PARAMETER ( N = 6, M = 1000 )
      REAL*8 A(M), B(M), C(M), T(N), U(N), V(N), W(N), R(N)
      REAL*8 Z(M), FP(M), EP(M)
      INTEGER*4 IFREE(N), ILOW(N), IHIGH(N), I, NP, MP, IALPHA
      COMMON /ABC/ A, B, C, T, U, V, W, R, IFREE, ILOW, IHIGH
      COMMON /ZFE/ Z, FP, EP, NP, MP, IALPHA
C     BEGIN
         IFREE(I) = -1
         IHIGH(I) = I + 1
         ILOW(I)  = 1
         T(I) = EP(I+1)
         U(I) = -FP(I+1)
```

```
      V(I) = 0.D0
      W(I) = 0.D0
      R(I) = Z(I+1)
      RETURN
   END

   SUBROUTINE GIVENS ( IT )
   PARAMETER ( N = 6, M = 1000 )
   REAL*8 A(M), B(M), C(M), T(N), U(N), V(N), W(N), R(N)
   REAL*8 Z(M), FP(M), EP(M)
   INTEGER IFREE(N), IT, ILOW(N), IHIGH(N), NP, MP, IALPHA
   COMMON /ABC/ A, B, C, T, U, V, W, R, IFREE, ILOW, IHIGH
   COMMON /ZFE/ Z, FP, EP, NP, MP, IALPHA
C  BEGIN
      REAL*8 CS, SN, HT, AP, UP, US, RP, H, DSQRT
      INTEGER K, L
      K = ILOW(IT)
      L = IHIGH(IT)
      IF ( T(IT) .EQ. 0.D0 ) THEN
         SN = 1.D0
         CS = 0.D0
      ELSE
         IF ( ABS(T(IT)) .GE. ABS(C(K)) ) THEN
            HT = C(K) / T(IT)
            CS = 1.D0 / DSQRT( 1.D0 + HT ** 2 )
            SN = CS * HT
         ELSE
            HT = T(IT) / C(K)
            SN = 1.D0 / DSQRT( 1.D0 + HT ** 2 )
            CS = SN * HT
         ENDIF
      ENDIF
      C(K) = SN * C(K) + CS * T(IT)
      IFREE(IT) = K-2
      IF ( K+2 .LE. IALPHA ) THEN
         H = B(K)
         B(K) = SN * H + CS * U(IT)
         T(IT) = CS * H - SN * U(IT)
         IF ( K+2 .LT. IALPHA ) THEN
            AP = SN * A(K) + CS * V(IT)
            UP = CS * A(K) - SN * V(IT)
            US = SN * V(IT) + CS * R(IT)
            RP = CS * V(IT) - SN * R(IT)
            A(K) = AP * SN + US * CS
            U(IT) = AP * CS - US * SN
C           U(IT) = UP * SN + RP * CS
            R(IT) = UP * CS - RP * SN
            IF ( K+2 .LE. L ) THEN
               H = B(K+1)
```

```
                B(K+1) = SN * H + CS * W(IT)
                V(IT) = CS * H - SN * W(IT)
                IF ( K+2 .LT. L ) THEN
                   H = C(K+2)
                   C(K+2) = SN * H
                   W(IT) = CS * H
                ENDIF
             ENDIF
          ENDIF
       ENDIF
       IF ( ILOW(IT)+1 .LT. MIN ( IHIGH(IT), IALPHA ) ) THEN
          ILOW(IT) = ILOW(IT) + 1
       ELSE
          A(IHIGH(IT)) = R(IT)
          B(IHIGH(IT)) = U(IT)
          C(IHIGH(IT)) = T(IT)
          IHIGH(IT) = IHIGH(IT) + NP
          IF ( IHIGH(IT) .LE. MP ) THEN
             ILOW(IT) = 1
             IFREE(IT) = -1
             T(IT) = EP(IHIGH(IT))
             U(IT) = -FP(IHIGH(IT))
             V(IT) = 0.D0
             W(IT) = 0.D0
             R(IT) = Z(IHIGH(IT))
          ENDIF
       ENDIF
       RETURN
    END

    SUBROUTINE NOWORK
       RETURN
    END

    SUBROUTINE APPERR( ALPHA )
    INTEGER ALPHA
C   COMPUTATION OF THE RATIONAL APPROXIMATION ERROR
    PARAMETER ( N = 6, M = 1000, MALPHA = 100 )
    INTEGER IFREE(N), ILOW(N), IHIGH(N)
    INTEGER IALPHA, NP, MP
    REAL*8 A(M), B(M), C(M), T(N), U(N), V(N), W(N), R(N)
    REAL*8 Z(M), EP(M), FP(M), PBPRIM(0:MALPHA), PB(0:MALPHA)
    COMMON /ABC/ A, B, C, T, U, V, W, R, IFREE, ILOW, IHIGH
    COMMON /ZFE/ FP, EP, NP, MP, IALPHA, PBPRIM, PB
C   BEGIN
       INTEGER I
       PBPRIM(0) = C(1)
       PB(0)     = C(2)
       PRINT1000, 0, C(1), C(2)
```

```
         DO 10 I = 1, ALPHA
            PBPRIM(I) = PBPRIM(I-1) * C(2*I+1)
            PB(I)     = PB(I-1)     * C(2*I+2)
            PRINT1000, I, PBPRIM(I), PB(I)
1000        FORMAT(I4, 2D12.4)
10       CONTINUE
         RETURN
      END

      SUBROUTINE RATAPP( BETA, CH, SIGMA )
      PARAMETER ( N = 6, M = 1000, MALPHA = 100 )
      INTEGER BETA
      CHARACTER*1 CH
      REAL*8 SIGMA(0:MALPHA, 4)
      INTEGER IFREE(N), ILOW(N), IHIGH(N)
      INTEGER IALPHA, NP, MP
      REAL*8 A(M), B(M), C(M), T(N), U(N), V(N), W(N), R(N)
      REAL*8 Z(M), EP(M), FP(M), PBPRIM(0:MALPHA), PB(0:MALPHA)
      COMMON /ABC/ A, B, C, T, U, V, W, R, IFREE, ILOW, IHIGH
      COMMON /ZFE/ FP, EP, NP, MP, IALPHA, PBPRIM, PB
C     COMPUTATION OF THE RATIONAL APPROXIMANT
C     BEGIN
         REAL*8 D12, E11, E12, E21, E22
         REAL*8 F11, F12, F21, F22, DD, EE, FF, QQ
         REAL*8 AUX, AUX1, AUX2
         INTEGER JJ, I
         IF ( BETA .EQ. 0 ) THEN
            IF ( CH .EQ. 'L' ) THEN
               SIGMA(0,3) = 1.D0
               SIGMA(0,4) = 0.D0
            ELSE
               SIGMA(0,3) = -B(1)/C(1)
               SIGMA(0,4) = 1.D0
            ENDIF
         ELSE
C           INITIALIZATION OF THE BACKWARD RECURRENCE
            JJ = BETA * 2
            EE = C(JJ-1)**2
            FF = B(JJ-1)*C(JJ)
            QQ = PB(BETA-1) / PBPRIM(BETA-1)
            IF (CH .EQ. 'L' ) THEN
               SIGMA(0,1) = -EE
               SIGMA(0,2) = -FF / QQ
               SIGMA(0,3) = -A(JJ-1)
               SIGMA(0,4) = -B(JJ) / QQ
               SIGMA(1,3) = 1.D0
               SIGMA(1,4) = 0.D0
            ELSE
               DD = -B(JJ+1) / C(JJ+1)
```

```
                GG = C(JJ)**2
                SIGMA(0,1) = -DD * EE * QQ
                SIGMA(0,2) = -( DD * FF + GG )
                SIGMA(0,3) = -( A(JJ-1) * DD + B(JJ) ) * QQ
                SIGMA(0,4) = -( B(JJ) * DD + A(JJ) )
                SIGMA(1,3) = DD * QQ
                SIGMA(1,4) = 1.D0
             ENDIF
C            BACKWARD RECURRENCE
             DO 100 J = BETA-1, 1, -1
C               COMPUTATION OF THE ELEMENTS OF THE MATRICES D,E AND F
                JJ = 2*J
                DD = -B(JJ+1) / C(JJ+1)
                EE = C(JJ-1)**2
                FF = B(JJ-1) * C(JJ)
                GG = C(JJ)**2
                QQ = PB(J-1) / PBPRIM(J-1)
                D12 = DD * QQ
                E11 = A(JJ-1)
                E12 = ( A(JJ-1) * DD + B(JJ) ) * QQ
                E21 = B(JJ) / QQ
                E22 = B(JJ) * DD + A(JJ)
                F11 = EE
                F12 = DD * EE * QQ
                F21 = FF / QQ
                F22 = DD * FF + GG
C               APPLYING THE BACKWARD RECURRENCE
                DEG = BETA - J
                SIGMA(DEG+1,3) = SIGMA(DEG,3) + D12 * SIGMA(DEG,4)
                SIGMA(DEG+1,4) =                    SIGMA(DEG,4)
                SIGMA(DEG,1)   = -F11 * SIGMA(DEG,3) - F12 * SIGMA(DEG,4)
                SIGMA(DEG,2)   = -F21 * SIGMA(DEG,3) - F22 * SIGMA(DEG,4)
                AUX = SIGMA(DEG,3)
                SIGMA(DEG,3)   = -E11 * AUX - E12 * SIGMA(DEG,4)
                SIGMA(DEG,4)   = -E21 * AUX - E22 * SIGMA(DEG,4)
                SIGMA(DEG,3)   = SIGMA(DEG,3) + SIGMA(DEG-1,3) +
     *                           D12 * SIGMA(DEG-1,4)
                SIGMA(DEG,4)   = SIGMA(DEG,4) + SIGMA(DEG-1,4)
                DO 50 I = DEG-1,0,-1
                   AUX1 = SIGMA(I,1)
                   AUX2 = SIGMA(I,2)
                   SIGMA(I,1) = -F11 * SIGMA(I,3) - F12 * SIGMA(I,4)
                   SIGMA(I,2) = -F21 * SIGMA(I,3) - F22 * SIGMA(I,4)
                   AUX = SIGMA(I,3)
                   SIGMA(I,3)   = -E11 * AUX - E12 * SIGMA(I,4)
                   SIGMA(I,4)   = -E21 * AUX - E22 * SIGMA(I,4)
                   SIGMA(I,3)   = SIGMA(I,3) + AUX1
                   SIGMA(I,4)   = SIGMA(I,4) + AUX2
                   IF ( I .GT. 0 ) THEN
```

```
                    SIGMA(I,3)   = SIGMA(I,3) + SIGMA(I-1,3) +
     *                                D12 * SIGMA(I-1,4)
                    SIGMA(I,4)   = SIGMA(I,4) + SIGMA(I-1,4)
                  ENDIF
 50             CONTINUE
 100          CONTINUE
C             MULTIPLICATION OF THE LAST TWO ROWS OF SIGMA
C             BY PSI0
              AUX = -B(1) / C(1)
              DO 200 I = 0, BETA
                 SIGMA(I,3) = SIGMA(I,3) + AUX * SIGMA(I,4)
 200          CONTINUE
           ENDIF
           RETURN
       END

       FUNCTION HORNER ( P, DEG, Z )
       REAL*8 HORNER
       REAL*8 P(0:*), Z
       INTEGER DEG
C      EVALUATION OF THE POLYNOMIAL P IN THE POINT Z
C      USING HORNER
C      BEGIN
           INTEGER I
           HORNER = P(DEG)
           DO 10 I = DEG-1, 0, -1
              HORNER = HORNER * Z + P(I)
 10        CONTINUE
           RETURN
       END
```

This parallel FORTRAN program is compiled, linked and executed on the IBM3090 machine under the CMS operating system in the following way.

```
PFPCOMP ratapp (PARALLEL(LANGUAGE REPORT(LIST))
PFPLINK ratapp
PFPEXEC ratapp (PAR(PROCS(3)))
```

Here `ratapp fortran a` is the name of the file containing the FORTRAN source code. We have executed the program with (3+1) processors. In case we need to link with a library, e.g. the ESSL library, we have to include the command

```
PFPGLOB ESVVLIB
```

before we link. To redirect the input and/or output of the "main" processor, we can use the following CMS commands.

```
FILEDEF 5 DISK input in a
FILEDEF 6 DISK output out a
```

To couple file `out3 out a` with the output of processor 3 and the input to the terminal, we can use the following commands.

```
FILEDEF S0003F05 TERMINAL
FILEDEF S0003F06 DISK out3 out a
```

Note that we have not used "locks" or "events" to synchronize the processors. We have used the array-variables `IHIGH`, `ILOW` and `IFREE` for synchronization. Each of the components of these variables is coupled with one processor. Only this processor can alter the value of the component.

C The use of timers

Because the parallel implementation of our rational approximation algorithm on the IBM3090 supercomputer is not very efficient, we have not included timing results. However, for the sake of completeness, we describe in this appendix some possibilities to time a parallel FORTRAN program on the IBM machine. We are explaining the use of `VMTIM`, `VCPUT`, `VMTIME`, `VCPUTM` and `PRTIME`. These tools are contained in the URCLIB library. Hence, before we link our parallel FORTRAN program, we have to give the following command: `PFPGLOB URCLIB`.

- `VMTIM` (virtual machine time): To start the timers, we call the subroutine as `CALL VMTIM ('start')` and to stop the timers, we use `CALL VMTIM ('stop')`. As a result, we obtain the wall clock time in seconds with an accuracy of 1 microsecond, the virtual and total CPU time in seconds with an accuracy of 1/100 of a second and the vector time in seconds with an accuracy of `NPROCS` seconds, where `NPROCS` is the number of FORTRAN processors. If we use the parallel FORTRAN, we have always at least two FORTRAN processors.

- `VCPUT` (virtual CPU time): The use is as above giving a timing for each processor. The accuracy is one microsecond for the wall clock time and one second for the other timers.

- `VMTIME` (see `VMTIM`): This subroutine is more accurate than `VMTIM` but uses the variable `TIME` of type `REAL*8 TIME(4)`, where `TIME(1)` indicates the cumulative wall clock time, `TIME(2)` the cumulative virtual CPU time, `TIME(3)` the cumulative total CPU time and `TIME(4)` the cumulative vector CPU time. The calling sequence is `CALL VMTIME (TIME)`.

- `VCPUTM` (see `VCPUT`: This subroutine is more accurate than `VCPUT` but uses the auxiliary variables, `WALL` of type `REAL*8` and `VTIME, TTIME, VECTM`, each of type `REAL*8 (NPROCS+1)`. The function of these variables is clear from their name. We call the subroutine as `CALL VCPUTM (WALL, VMTIME, TTIME, VECTM)`.

- `PRTIME`: To measure the cumulative wall clock time, the virtual CPU time, the total CPU time and the virtual vector CPU time for a virtual processor in use. The identification number of the processor in use can be obtained by `ID = IPROC()` or `CALL IPROC(ID)`. The calling sequence for the subroutine is `CALL PRTIME (TIME)` where `TIME` is of type `REAL*8 TIME(4)` where the components have a similar meaning as for the subroutine `VMTIME`. All measurements are given in seconds and are accurate within one microsecond.