SIAM J. SCI. STAT. COMPUT.
Vol. 6, No. 4, October 1985

PRACTICAL USE OF POLYNOMIAL PRECONDITIONINGS FOR THE CONJUGATE GRADIENT METHOD*

YOUCEF SAAD†

Abstract. This paper presents some practical ways of using polynomial preconditions for solving large sparse linear systems of equations issued from discretizations of partial differential equations. For a symmetric positive definite matrix A these techniques are based on least squares polynomials on the interval $[0, b]$ where b is the Gershgorin estimate of the largest eigenvalue. Therefore, as opposed to previous work in the field, there is no need for computing eigenvalues of A. We formulate a version of the conjugate gradient algorithm that is more suitable for parallel architectures and discuss the advantages of polynomial preconditioning in the context of these architectures.

Key words. conjugate gradient method, polynomial preconditionings, parallel algorithms, vectorization

1. Introduction. When combined with a suitable preconditioning, the conjugate gradient method constitutes one of the most powerful techniques for solving large sparse symmetric positive definite linear systems of equations. However, most of the preconditionings have originally been designed for scalar computers and much work must be done to reformulate them or develop new ones that are more suitable for the new generation of computers. One attractive possibility considered by several authors [1], [3], [5], [8], [10], [19] is the use of polynomial preconditionings. Given a symmetric system $Ax = f$, the principle of polynomial preconditioning, which goes back to Rutishauser [16], consists in solving the (preconditioned) linear system $s(A)Ax = s(A)f$, where s is some polynomial, usually of low degree. The polynomial s is chosen so that the matrix $s(A)A$ has an eigenvalue distribution that is favorable to the conjugate gradient method, i.e. so that the conjugate gradient method applied to the preconditioned system converges rapidly. In the classical context of scalar computers, there is little reason for using polynomial preconditionings because the conjugate gradient method is an optimal process and the total number of matrix by vector multiplications required by the conjugate gradient method applied to the preconditioned system $s(A)Ax = s(A)f$ will be higher than that of the nonpreconditioned system $Ax = f$. The loss incurred by a larger number of matrix-vector multiplications may in some cases be offset by the smaller number of inner products required when polynomial preconditioning is used but the overall performance is not likely to be much different in those cases. Moreover, incomplete factorization based preconditionings are then quite effective, and are usually preferred to polynomial preconditionings.

Jordan [10] reports some experiments on the CRAY-1 showing that polynomial preconditionings are competitive on vector computers. He concludes that more work is needed to simplify the derivation of "good" polynomials. One of our main goals is to propose a class of effective polynomials that are easy to derive and that do not require eigenvalue estimates.

In addition to their importance for vector computers polynomial preconditionings are attractive for parallel architectures. As will be seen, when A is block tridiagonal, a great deal of parallelism can be achieved when computing $s(A)Av$. A few ideas describing how to take advantage of polynomial preconditionings in this particular context will be seen in § 3.

* Received by the editors December 28, 1983, and in final revised form August 1, 1984. This work was supported by the Office of Naval Research under grant N000014-82-K-0184 and by the National Science Foundation under grant MCS-81-06181.

† Computer Science Department, Yale University, New Haven, Connecticut 06520.

In the first part of this paper we focus on the problem of choosing a good polynomial s. The classical choice is to take $s(\lambda)$ so that the residual polynomial $R(\lambda) \equiv 1 - \lambda s(\lambda)$ minimizes $\|R(\lambda)\|_\infty$ over all polynomials R of degree not exceeding k so that $R(0) = 1$, where $\|\cdot\|_\infty$ is the infinity norm on some interval $[a, b]$ containing the spectrum of A, with $0 < a < b$ [16]. This leads to the well-known Chebyshev iteration. In this paper we suggest using a polynomial s that minimizes the L_2-norm $\|R(\lambda)\|_w$ with respect to some weight function w defined on an interval $[a, b]$ that contains the spectrum of A. The idea of using the L_2-norm instead of the infinity norm goes back to Stiefel [20] and was recently suggested for polynomial preconditioning by Johnson, Micchelli and Paul [8]. An important observation made by Stiefel is that a and b need not be accurate estimates of the smallest and the largest eigenvalues of A as long as $[a, b]$ contains the spectrum of A. Indeed, while it is necessary for Chebyshev iteration that $0 < a < b$, for the L_2-norm there is no such restriction and one can simply use the interval which is provided by Gershgorin's theorem. For example, we may use an interval of the form $[0, b]$ when it is known that A is positive definite and that b is an upper bound for the largest eigenvalue provided e.g. by Gershgorin's theorem. The Gershgorin bounds can be obtained as the matrix A is built and therefore there is no need for an adaptive scheme which may considerably slow down the flow of computations in vector or parallel machines.

One might ask whether we lose efficiency when using least squares polynomials instead of Chebyshev polynomials since we require less information, i.e. since we require Gershgorin values instead of eigenvalues. In fact, an interesting revelation from the numerical experiments is that the usual optimum parameters $a = \lambda_1 \equiv$ the smallest eigenvalue of A and $b = \lambda_N \equiv$ the largest eigenvalue of A, used in the Chebyshev iteration, *do not in general minimize the total number of conjugate gradient iterations required for convergence.* If these values were used, then our experiments show that the least squares polynomial approach performs better, not worse as might have been expected, than the more complicated Chebyshev polynomial approach. The above optimal parameters are known to minimize the condition number of the iteration matrix $As(A)$, as was proven by Johnson, Micchelli and Paul [8]. However, the condition number does not matter as much as the overall distribution of eigenvalues. For this reason it is clear that it will be difficult to compute the best parameters, i.e. the parameters a and b that maximize the rate of convergence. This constitutes the main drawback of Chebyshev polynomial preconditioning. The numerical experiments indicate that the least squares polynomials perform quite well in spite of the fact that they do not require eigenvalue estimates.

The second part of this paper will discuss the practical implementation of polynomial preconditionings in parallel computation. Although there are many possible implementations, our idea rests on the simple principle, which is not completely new, that when A is block tridiagonal then one can perform the products $Av, A^2v, \cdots, A^kv$ concurrently.

2. Polynomial iteration and polynomial preconditioning.

2.1. Basic theory. Consider the linear system

$$Ax = f, \tag{1}$$

where A is symmetric positive definite. An efficient technique for solving (1) is the conjugate gradient method applied to the preconditioned system

$$Q^{-1}Ax = Q^{-1}f, \tag{2}$$

where Q^{-1} is some approximate inverse of A, for which systems of the form $Qy = z$ are easy to solve.

The matrix Q is referred to as the preconditioning matrix and a classical example is the incomplete Choleski factorization of A [12]. Although the preconditioned system (2) is no longer symmetric, symmetry can be recovered by using the inner product $(x, y)_Q = (Qx, y)$ instead of the Euclidean inner product in the conjugate gradient method [12]. It is assumed that $Q^{-1}A$ is self-adjoint and positive definite for this inner product.

Incomplete factorization preconditionings are very powerful techniques in scalar machines but do not vectorize well and may not be the best choice for supercomputers. Some newly developed preconditionings do, however, vectorize fairly well. Among them let us mention the vectorizable version of the incomplete Choleski factorization proposed by Van der Vorst [21]. In [10] a few ways of adapting classical preconditioners to vector and parallel computers are compared.

Several authors have suggested using polynomial preconditionings, that is taking $Q^{-1} \equiv s(A)$, where s is some polynomial; see [5], [8], [10]. The simplest such polynomial suggested by Dubois et al. [5], is given by the Neuman series

$$s(A) = I + N + N^2 + \cdots + N^k$$

where $N = I - A$ is assumed to be such that $\|N\| \leqq 1$, which is often verified.

More efficient polynomials can be obtained if one knows good estimates a and b of the smallest and the largest eigenvalues of A. Indeed, let $\sigma(A) = \{\lambda_i\}_{i=1,\cdots,N}$ be the spectrum of A with $\lambda_1 \leqq \lambda_2 \leqq \cdots \leqq \lambda_N$. Let $s(\lambda)$ be any polynomial of degree not exceeding $k-1$ and consider the matrix $Q^{-1}A = s(A)A$ of the preconditioned system (2). The purpose of the preconditioning is to transform the eigenvalue distribution into one that is more favorable to the conjugate gradient method. For example, we could choose the polynomial s so that $\|I - s(A)A\|$ is minimized, where $\|\cdot\|$ represents the 2-norm. Noticing that

$$\|I - s(A)A\| = \max_{\lambda_i \in \sigma(A)} |1 - \lambda_i s(\lambda_i)|, \tag{3}$$

it is clear that a good polynomial $s(\lambda)$ would be one for which

$$\max_{\lambda \in [a,b]} |1 - \lambda s(\lambda)|, \tag{4}$$

is minimal over all polynomials of degree $\leqq k-1$, where $[a, b]$ is an interval that contains $\sigma(A)$ with $0 < a < b$. It is then well known that the best such polynomial is such that $1 - \lambda s(\lambda)$ is an appropriately scaled and shifted Chebyshev polynomial of degree k of the first kind, see [2]. This constitutes the foundation of Chebyshev iteration [6] and was also considered for polynomial preconditioning [8], [10]. In fact it can be shown [8] that when $a = \lambda_1$ and $b = \lambda_N$, the resulting preconditioned matrix minimizes the condition number of the preconditioned matrices of the form $As(A)$ over all polynomials s of degree $\leqq k-1$. However, an interesting numerical observation is that when used in conjunction with the conjugate gradient method the best polynomial, i.e. the one which minimizes the total number of conjugate gradient iterations *is far from being the one that minimizes the condition number.* The behaviour of the polynomial preconditioned conjugate gradient is not simple to analyse. Thus, if instead of taking $a = \lambda_1$ and $b = \lambda_N$ we took $[a, b]$ to be slightly inside the interval $[\lambda_1, \lambda_N]$, we would achieve faster convergence in general. Unfortunately, the true optimal parameters, i.e. those that minimize the number of iterations of the polynomial preconditioned conjugate gradient method, are not known and we do not know of any way to obtain them.

Several authors have also considered instead of (4) a L_2-norm over the interval $[a, b]$ with $a = \lambda_1$, $b = \lambda_N$ with respect to some weight function w [8], [17], [18], [20]. Johnson, Micchelli and Paul [8] have experimentally shown that the resulting preconditionings may lead to faster convergence than the Chebyshev based ones.

A further disadvantage of the approaches described above is that the parameters a and b which approximate the smallest and largest eigenvalues of A are usually not available beforehand and must be obtained in some dynamic way. This slows down the process and makes it unattractive for supercomputers where one should avoid halting the flow of computation.

In order to overcome the difficulty of computing the parameters a and b, Stiefel suggested using for a and b the values provided by an application of Gershgorin's theorem. Thus, the parameter a which *estimates* the smallest eigenvalue of A may be nonpositive even when A is a positive definite matrix. However, when $a \leqq 0$ the problem of minimizing (4) is not well defined, i.e. it does not have a unique solution. This is due to the nonstrict-convexity of the uniform norm. Stiefel then suggests using the L_2-norm on $[a, b]$ with respect to some weight function $w(\lambda)$.

Consider the inner product on the space P_k of polynomials of degree not exceeding k:

$$\langle p, q \rangle = \int_a^k p(\lambda) q(\lambda) w(\lambda)\, d\lambda \tag{5}$$

where $w(\lambda)$ is some nonnegative weight function on (a, b). We will denote by $\|p\|_w$ and call w-norm the 2-norm induced by this inner product.

From the above discussion, we seek the polynomial $s_{k-1}(\lambda)$ which minimizes

$$\|1 - s(\lambda)\|_w \tag{6}$$

over all polynomials s of degree $\leqq k-1$. We will call s_{k-1} the least squares iteration polynomial, or simply the least squares polynomial and will refer to $R_k(\lambda) \equiv 1 - \lambda s_{k-1}(\lambda)$ as the least squares residual polynomial. A crucial observation made by Stiefel is that, as opposed to the classical approach using the infinity norm, the least squares polynomial is now well defined for arbitrary values of a and b. As will be shown, $s_{k-1}(A)$ will constitute a good approximation to A^{-1} as the degree $k-1$ increases. Computing the polynomial $s_{k-1}(\lambda)$ is not a difficult task when the weight function w is suitably chosen. This will constitute the object of the next section.

2.2. Computation of the least squares polynomials. There are at least three ways of computing the least squares polynomial defined in the previous section:

1. By using the kernel polynomials formula [20]:

$$R_k(\lambda) = \left[\sum_{i=0}^{k} q_i(0) q_i(\lambda) \right] \Big/ \left[\sum_{i=0}^{k} q_i(0)^2 \right] \tag{7}$$

in which the q_i's represent a sequence of polynomials orthogonal with respect to the weight function $w(\lambda)$.

2. By generating a three term recurrence satisfied by the residual polynomials $R_k(\lambda)$ [20]. Indeed, it is known that the residual polynomials are orthogonal with respect to the new weight function $\lambda w(\lambda)$. For more details see Stiefel [20].

3. By solving the corresponding normal equations [17]:

$$\langle 1 - \lambda s_{k-1}(\lambda), \lambda Q_j(\lambda) \rangle = 0, \qquad j = 0, 1, 2, \cdots k-1 \tag{8}$$

where Q_j, $j = 1, \cdots k-1$ is any basis of the space P_{k-1} of polynomials of degree $\leqq k-1$. See [17] for the treatment of a similar problem in a slightly more general context.

Each of these three approaches is useful in a different context. Approach 1 is general and is useful for computing explicitly least squares polynomials of low degree. For high degree polynomials the last two approaches are to be preferred for numerical stability. Approach number 2 is restricted to the case where $a \geqq 0$, while approach number 3 is more general. We should point out that the degrees of polynomial preconditioners are often low, e.g. not exceeding 5 or 10 so we will describe the first formulation in more detail.

Let $q_i(\lambda)$, $i = 0, 1, \cdots, n, \cdots$ be the *orthogonal* polynomials with respect to $w(\lambda)$. It is known that the least squares residual polynomial $R_k(\lambda)$ of degree k is determined by the kernel polynomials formula (7). To get $s_{k-1}(\lambda)$ simply notice that

$$s_{k-1}(\lambda) = (1 - R_k(\lambda))/\lambda = \left[\sum_{i=0}^{k} q_i(0) t_i(\lambda) \right] \Big/ \left[\sum_{i=0}^{k} q_i(0)^2 \right]$$

with

$$t_i(\lambda) = (q_i(0) - q_i(\lambda))/\lambda$$

which allows one to compute s_{k-1} as a linear combination of the polynomials $t_i(\lambda)$. Thus from the orthogonal polynomials q_i one can compute the desired least squares polynomials. The polynomials q_i satisfy a three term recurrence of the form

$$\beta_{i+1} q_{i+1}(\lambda) = (\lambda - \alpha_i) q_i(\lambda) - \beta_i q_{i-1}(\lambda), \qquad i = 1, 2, \cdots,$$

from which we derive the following recurrence for the t_i's

$$\beta_{i+1} t_{i+1}(\lambda) = (\lambda - \alpha_i) t_i(\lambda) - \beta_i t_{i-1}(\lambda) + q_i(0), \qquad i = 1, 2, \cdots.$$

The weight function w will be chosen so that the three term recurrence of the orthogonal polynomials q_i is explicitly known and/or is easy to generate. One interesting class of weight functions that satisfy this requirement is considered next.

2.3. Choice of the weight functions. In this section we assume that $a = 0$ and $b = 1$. Consider the Jacobi weights

$$w(\lambda) = \lambda^{\alpha - 1}(1 - \lambda)^{\beta} \quad \text{where } \alpha > 0 \quad \text{and} \quad \beta \geqq -\tfrac{1}{2}. \tag{9}$$

For these weight functions, the recurrence relations are explicitly known for the polynomials that are orthogonal with respect to $w(\lambda)$, $\lambda w(\lambda)$ or $\lambda^2 w(\lambda)$, thus allowing the use of any of the three methods described in the previous section for computing $s_{k-1}(\lambda)$. Moreover, from a result of [8], the matrix $As_k(A)$ is known to be positive definite when A is positive definite and $\alpha - 1 \geqq \beta \geqq -\frac{1}{2}$.

The following explicit formula for $R_k(\lambda)$ can easily be derived from the explicit expression of the Jacobi polynomials [4] and the fact that $\{R_k\}$ is orthogonal with respect to the weight $\lambda w(\lambda)$:

$$R_k(\lambda) = \sum_{j=0}^{k} \kappa_j^{(k)} (1 - \lambda)^{k-j} (-\lambda)^j \quad \text{where} \quad \kappa_j^{(k)} = \binom{k}{j} \prod_{i=0}^{j-1} \frac{k - i + \beta}{i + 1 + \alpha}. \tag{10}$$

From (1) it is easy to derive the polynomial $s_{k-1}(\lambda) = (1 - R_k(\lambda))/\lambda$ "by hand" for small degrees.

As an example, when $\alpha = \frac{1}{2}$ and $\beta = -\frac{1}{2}$ we get the following first four polynomials:

$$s_0(\lambda) = \tfrac{4}{3},$$

$$s_1(\lambda) = 4 - (16/5)\lambda,$$

$$s_2(\lambda) = \tfrac{2}{7}[28 - 56\lambda + 32\lambda^2],$$

$$s_3(\lambda) = \tfrac{2}{9}[60 - 216\lambda + 288\lambda^2 - 128\lambda^3].$$

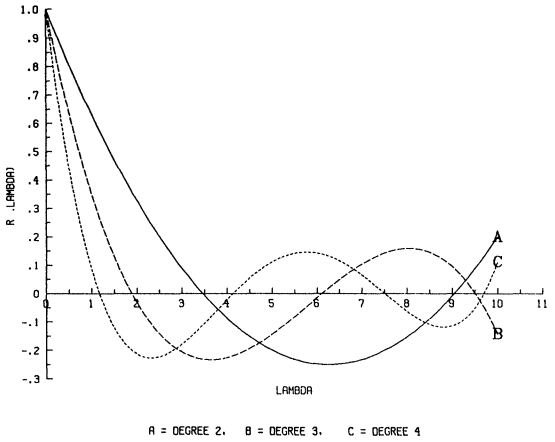

FIG. 2.1. *Residual polynomials R_k for $k = 2, 3$ and 4.*

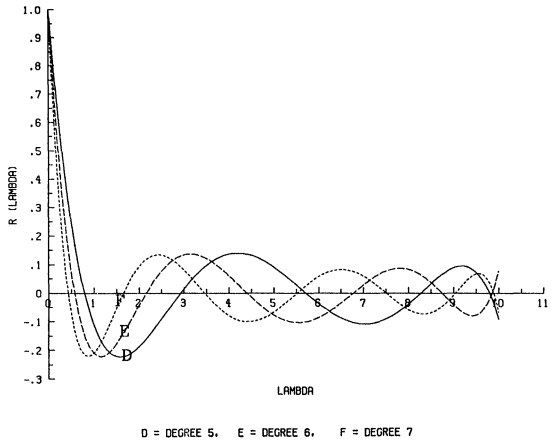

FIG. 2.2. *Residual polynomials R_k for $k = 5, 6$ and 7.*

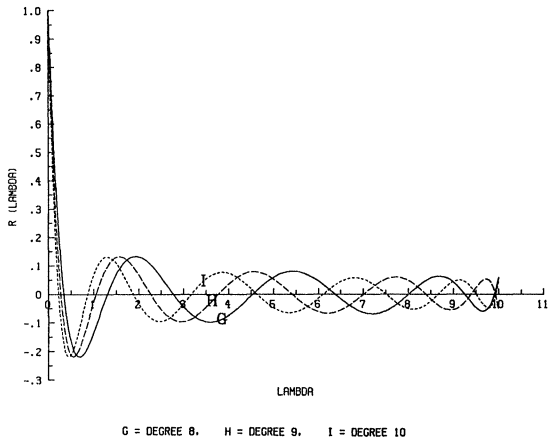

FIG. 2.3. *Residual polynomials R_k for $k = 8, 9$ and 10.*

Recall that the above polynomials correspond to the interval $[0, 1]$. For a more general interval of the form $[0, b]$, a change of variable must be applied to map the variable in $[0, b]$ into $[0, 1]$, i.e. the kth best polynomial for the interval $[0, b]$ is $(1/b)s_{k-1}(\lambda/b)$. The plots of the residual polynomials $R_k(\lambda)$, $k = 2, 3, \cdots, 10$, for the above weight functions are presented in Figs. 2.1, 2.2, 2.3. The interval considered is $[a, b] = [0, 10]$. The polynomials $s_k(\lambda) k = 1, 2, \cdots, 10$ have been computed by the method of the previous section and are listed in the appendix.

Note that we have taken $\alpha = \frac{1}{2}$ and $\beta = -\frac{1}{2}$ as an example only because this choice leads to a very simple recurrence for the polynomials q_i, which are the Chebyshev polynomials of the first kind. We will also use this selection for the numerical experiments in § 4. Johnson, Micchelli and Paul [8] have taken as an example $\alpha = 1$, and $\beta = 0$ which corresponds to the Legendre weight. An interesting problem from the practical viewpoint is to determine what is a "good" choice for α and β.

2.4. Theoretical considerations. An interesting theoretical question is whether the least squares residual polynomial will become small in some sense as its degree increases. Consider first the case $0 < a < b$. Since the residual polynomial R_k minimizes the norm $\|R\|_w$ associated with the weight w, over all polynomials R of degree $\leqq k$ such that $R(0) = 1$, the polynomial $(1 - \lambda/c)^k$ where $c = (a + b)/2$ satisfies

$$\|R_k\|_w \leqq \|(1 - \lambda/c)^k\|_w \leqq \|[(b - a)/(b + a)]^k\|_w = \kappa[(b - a)/b + a)]^k,$$

where κ is the w-norm of the function unity on the interval $[a, b]$. Hence, the (known) result that the norm of R_k will tend to zero geometrically as k tends to infinity.

Consider now the case $a = 0$, $b = 1$ and the Jacobi weight (9). Then for this choice of the weight function, the least squares residual polynomial is known to be $p_k(\lambda)/p_k(0)$ where p_k is the kth degree Jacobi polynomial associated with the weight function $w'(\lambda) = \lambda^\alpha (1 - \lambda)^\beta$, [20]. Consider the 2-norm of such a residual polynomial with respect to this weight. From [4] and after a change of variable that maps the interval $[-1, 1]$ into $[0, 1]$ we obtain

$$p_k(0) = \frac{\Gamma(k + \alpha + 1)}{\Gamma(k + 1)\Gamma(\alpha + 1)}$$

where Γ represents the Γ function, and

$$\|p_k\|_{w'}^2 = \frac{1}{2k + \alpha + \beta + 1} \frac{\Gamma(k + \alpha + 1)\Gamma(k + \beta + 1)}{\Gamma(k + 1)\Gamma(k + \beta + \alpha + 1)}.$$

Hence,

$$\|p_k/p_k(0)\|_{w'}^2 = \frac{\Gamma^2(\alpha + 1)\Gamma(k + \beta + 1)}{(2k + \alpha + \beta + 1)\Gamma(k + \alpha + \beta + 1)} \frac{\Gamma(k + 1)}{\Gamma(k + \alpha + 1)}.$$

For the case $\alpha = \frac{1}{2}$ and $\beta = -\frac{1}{2}$ this becomes

$$\frac{[\Gamma(3/2)]^2}{(2k + 1)(k + 1/2)} = \frac{\pi}{2(2k + 1)^2}.$$

Therefore, the w'-norm of the least squares residual polynomial will converge to zero like $1/k$ as the degree k increases. This is not as fast as when $a > 0$ but we must remember that the condition $p(0) = 1$, implies that the polynomial is large in some interval around the origin.

The case $a<0<b$ is more difficult to study. Clearly, the problem is to analyse the numbers

$$\min_{p\in P_k, p(0)=1} \int_a^b [p(\lambda)]^2 w_0(\lambda)\, d\lambda$$

as k increases to infinity, where w_0 is some weight function. By the change of variable $\mu=(\lambda-a)(b-a)$, these numbers are transformed into

$$\kappa_k(\gamma)=\min_{p\in P_k, p(\gamma)=1} \int_0^1 |p(\mu)|^2 w(\mu)\, d\mu,$$

where $w(\mu)=w_0(\lambda)$ and $\gamma=-a/(b-a)$ is inside [0, 1]. A detailed analysis by Nevai [14] has shown that this function of γ, called the Christoffel function, decreases to zero like $1/k$, when w is a Jacobi weight.

3. Polynomial preconditionings and parallel processing.

3.1. Parallel computation of $p(A)v$. Consider a linear system $Ax=f$ issued from the discretization of a partial differential equation in two or three dimensions. With a suitable ordering of the nodes, the matrix A is block-tridiagonal and often of very large size. We would like to show how to exploit the polynomial preconditioning discussed in the preceding sections to solve these linear systems. A critical part in the realization of the conjugate gradient method lies in the computation of $s(A)Av$ for any given vector v, where $p(A)\equiv As(A)=s(A)A$ is the polynomial preconditioned matrix. Because of the particular structure of A, a great deal of parallelism can be achieved in the computation of $p(A)v$. In the following description we will assume that the degree of p is 3. We will call ν the block dimension of A and m the dimension of each block, i.e. we have $N=\nu m$.

Let v be any vector and let $w=Av$, $z=A^2v=Aw$, $t=A^3v=Az$. We partition all the vectors according to the block structure of A and denote by v_i, w_i, z_i, t_i, $i=1,\cdots,\nu$, the block entries of v, w, z, and t respectively. Notice that the computation of any block entry w_i of $w=Av$, requires only the knowledge of the block entries v_{i-1}, v_i and v_{i+1}. Then a key observation is the following: while we compute w_i from v_{i-1}, v_i and v_{i+1} we can at the same time compute z_{i-2} from w_{i-3}, w_{i-2}, w_{i-1} and t_{i-4} from z_{i-5}, z_{i-4}, z_{i-3}. This is illustrated in Fig. 3.1 where we assume that we perform the computations

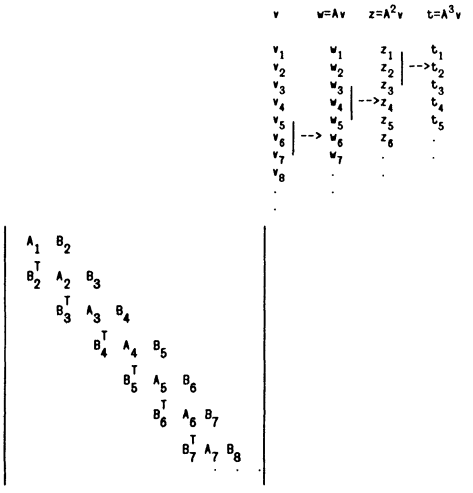

FIG. 3.1. *Simultaneous computation of Av, A^2v and A^3v.*

from top to bottom. These three computations can obviously be performed independently from each other by three different processors. This simple idea is not completely new as a similar principle was already used in a different context in 1963 by Pfeifer [15] who suggested performing simultaneously several steps of the three line cyclically reduced SOR method.

Concerning the entries of the matrix A, note that in order to compute w_i we only need the blocks A_i, B_i and B_{i+1}. Likewise, we need B_{i-2}, A_{i-2} and B_{i-1} to compute z_{i-2} and we need B_{i-4}, A_{i-4} and B_{i-3}, to compute t_{i-4}. Hence, the computation can be pipelined by holding 4 blocks in each processor and moving everything to the left by two blocks (A_i and B_{i+1}) at each time step as is shown in Fig. 3.2. Although only three of the four blocks in each processor are used, this organization simplifies the data flow of the matrix entries. Meanwhile, the vectors move only by one block at a time. Figure 3.2 illustrates two successive steps of the process. Note that the computations in each processor are identical but use three different sets of data. Processor P_1 has the task of computing $w = Av$, processor P_2 computes $z = Aw$, and processor P_3 computes $t = Az$. As soon as a component is computed, it is sent to the processor on its left. For example, as the figure shows, when the computation of w_6 is completed, w_6 is moved to processor P_2, where it will be used in the next step together with w_4 and w_5 to produce z_5. Simultaneously, z_4 is moved to P_2 and t_2, the final result, is moved to some host processor that will complete the conjugate gradient step as will be discussed shortly.

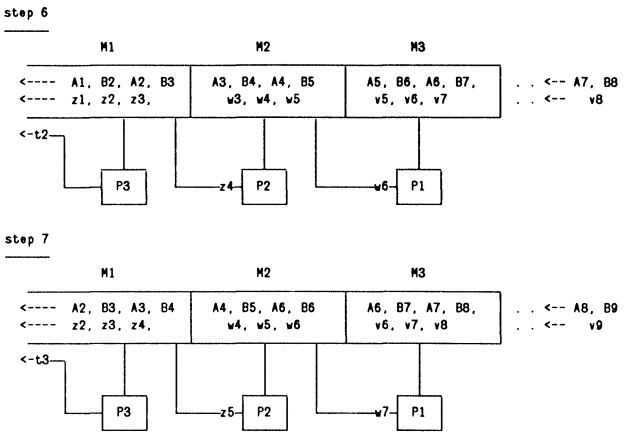

FIG. 3.2. *Parallel processing for computing* $p(A)v$.

In the network described by the figure, the processors P_1, P_2, P_3 may represent vector or array processors with their own memories M_1, M_2, M_3 sufficiently large to hold four blocks of A. In [11] Jordan describes a technique for computing $w = Av$ that only requires two blocks of A at a time, namely A_i and B_{i+1} to get one corresponding bloc of w. This amounts to separating the computation of $w_i = B_i^T v_{i-1} + A_i v_i + B_{i+1} v_{i+1}$ $i = 1, 2, \cdots$, into

$$w_i = \tilde{w}_i + A_i v_i + B_{i+1} v_{i+1}, \tag{11}$$

$$\tilde{w}_{i+1} = B_{i+1}^T v_i, \tag{12}$$

with $\tilde{w}_1 = 0$. With such a technique we would only need two blocks of A in each processor. The vector $\tilde{w}_{i+1}$ is not moved to the left but will remain in the same processor to compute w_{i+1} by (11) in the next time step.

The advantages of the approach outlined in this section are the following:

1. A high degree of parallelism is reached. The processors are idle only during a very small portion of the total time spent to compute $p(A)v$, namely at the beginning and at the end of the computation.

2. The computation involved in each of the processors is a highly vectorizable process. Thus, each processor can represent a vector or array processor and this can be quite important for very large problems, e.g. those issued from finite difference discretizations of three-dimensional PDE's.

3. The data communication required is regular. Moreover, a large part of the data transfer can easily be overlapped with computation.

We have just described how to compute the vectors Av, A^2v, A^3v but in reality we need to compute the vector $t = p(A)v$ where p is some polynomial of the form $p(\lambda) = \alpha_3[\lambda^3 + \alpha_2\lambda^2 + \alpha_1\lambda + \alpha_0]$. Therefore, we will not compute the vectors $w = Av$, $z = Aw$, $t = Az$, but rather the vectors:

$$w := \alpha_2 v + Av, \tag{13}$$

$$z := \alpha_1 v + Aw, \tag{14}$$

$$t := \alpha_0 v + Az, \tag{15}$$

$$t := \alpha_3 t, \tag{16}$$

which result from the application of Horner's scheme for evaluating a polynomial. The resulting modification to the above scheme requires the transfer of the additional vector v through the three processors, and is straightforward.

3.2. Application to the conjugate gradient method. We now turn to the implementation of the polynomial preconditioned conjugate gradient method a classical version of which can be described as follows.

ALGORITHM 1.

1. *Start*: Choose $x^{(0)}$ and compute $r^{(0)} := s(A)(f - Ax^{(0)})$. Set $v^{(0)} := r^{(0)}$, $\rho^{(0)} := (r^{(0)}, r^{(0)})$.

2. *Iterate*: For $j = 1, 2, \cdots$ until convergence *do*

$$t^{(j)} := s(A)Av^{(j)}; \tag{17}$$

$$\alpha^{(j)} := \rho^{(j)}/(t^{(j)}, v^{(j)}); \tag{18}$$

$$x^{(j+1)} := x^{(j)} + \alpha^{(j)}v^{(j)} \tag{19}$$

$$r^{(j+1)} := r^{(j)} - \alpha^{(j)}t^{(j)}; \tag{20}$$

$$\rho^{(j+1)} := (r^{(j+1)}, r^{(j+1)}); \tag{21}$$

$$\beta^{(j)} := \rho^{(j+1)}/\rho^{(j)}; \tag{22}$$

$$v^{(j+1)} := r^{(j+1)} + \beta^{(j)}v^{(j)}. \tag{23}$$

The inner products in (18) and (22) constitute two bottlenecks in the above algorithm. Indeed, as the algorithm is presented, we must proceed as follows. As the blocks of the vectors v and t emerge from the pipeline of Fig. 3.2 we must compute the partial inner product in (18) corresponding to these blocks. The blocks are then stored back into memory awaiting for the completion of the partial inner products. After the ν partial inner products have been computed and added up, the vectors t

and v are then recalled one block at a time and the scalar by vector products (19), (20) will be performed.

A similar remark holds for the computation of the inner product (22). However, an interesting observation is that there is an alternative way of computing $\beta^{(j)}$ which avoids this second bottleneck. Indeed, let $r^{(j+1)}$ and $r^{(j)}$ be two successive residual vectors, and $t^{(j)} = p(A)v^{(j)}$, $\alpha^{(j)} = (r^{(j)}, r^{(j)})/(t^{(j)}, v^{(j)})$ where $v^{(j)}$ is the conjugate direction at step j. Since the two vectors $r^{(j)}$ and $r^{(j+1)}$ are known to be orthogonal, we have from (20)

$$(r^{(j+1)}, r^{(j+1)}) + (r^{(j)}, r^{(j)}) = [\alpha^{(j)}]^2 (t^{(j)}, t^{(j)}).$$

Hence, another way of obtaining the inner product $(r^{(j+1)}, r^{(j+1)})$ is through the formula

$$(r^{(j+1)}, r^{(j+1)}) = [\alpha^{(j)}]^2 (t^{(j)}, t^{(j)}) - (r^{(j)}, r^{(j)}). \tag{24}$$

As a consequence $(r^{(j+1)}, r^{(j+1)})$ is available from $(r^{(j)}, r^{(j)})$ and $(t^{(j)}, t^{(j)})$. The fundamental difference with the usual way of computing $(r^{(j+1)}, r^{(j+1)})$ is that $(t^{(j)}, t^{(j)})$ can be computed simultaneously with $(t^{(j)}, v^{(j)})$, i.e. it can be accumulated as the blocks of the vectors $t^{(j)}$ come out of the network of Fig. 3.1 one by one. A similar observation was made by Johnsson [9] while Van Rosendale [22] studied the inner product data dependencies in the conjugate gradient method in a more general and theoretical way.

The following algorithm implements the above approach.

ALGORITHM 2.

1. *Start*: Choose $x^{(0)}$ and compute $r^{(0)} := s(A)(f - Ax^{(0)})$. Set $v^{(0)} := r^{(0)}$, $\rho^{(0)} := (r^{(0)}, r^{(0)})$.

2. *Iterate*: For $j = 1, 2, \cdots$ until convergence *do*

a) Compute in parallel:

$$t^{(j)} := s(A)Av^{(j)} \tag{25}$$

b) Compute in parallel:

$$(t^{(j)}, v^{(j)}), \tag{26}$$

$$(t^{(j)}, t^{(j)}), \tag{27}$$

then compute

$$\alpha^{(j)} := \rho^{(j)}/(t^{(j)}, v^{(j)}), \tag{28}$$

$$\beta^{(j)} := \rho^{(j)} \frac{(t^{(j)}, t^{(j)})}{(t^{(j)}, v^{(j)})^2} - 1,$$

$$\rho^{(j+1)} := \rho^{(j)} \beta^{(j)}.$$

c) Compute in parallel:

$$x^{(j+1)} := x^{(j)} + \alpha^{(j)} v^{(j)} \tag{29}$$

$$r^{(j+1)} := r^{(j)} - \alpha^{(j)} t^{(j)}, \tag{30}$$

$$v^{(j+1)} := r^{(j+1)} + \beta^{(j)} v^{(j)}. \tag{31}$$

Note that because of roundoff, the equation (28) could lead to a negative value for $\beta^{(j)}$. The effect of roundoff can be reduced by a periodic application of the less effective formulas of Algorithm 1. In any case the sign of $\beta^{(j)}$ should be checked before

applying the result to the next equations and if this sign is negative then $\beta^{(j)}$ should be recomputed by the classical formulas of Algorithm 1.

This second form of the conjugate gradient algorithm is also to be preferred on vector computers such as the CRAY-1. Indeed, in that situation the expressions "in parallel" in the above algorithm should be interpreted in the sense that the presence of the operands in the vector registers must be exploited to perform the desired computations at once thus economizing vector "load" and "store" operations. Practically, this can be achieved in FORTRAN by having a single DO loop for (26) and (27) and another single DO loop for (29), (30) and (31).

Fig. 3.3 shows a network of processors for realizing the above conjugate gradient algorithm. The first part represents the pipelined module described in Fig. 3.2 consisting of k linearly connected processors $P_1, \cdots, P_k$. The second part is a host processor, denoted by H in the figure, which realizes steps 2b and 2c of Algorithm 2. Processor H need not hold the matrix A but only the vectors x, r, v and t. It may itself consist of one or more array processors.

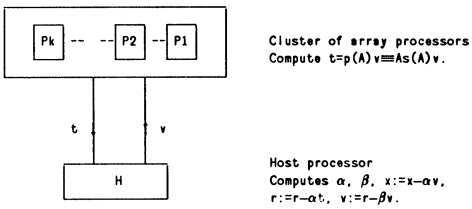

FIG. 3.3. *A parallel architecture for the polynomial preconditioned conjugate gradient method.*

An interesting question is how does this compare with the conjugate gradient algorithm without preconditioning. The following is a rough estimate of the time spent to execute one step of the polynomial preconditioned conjugate gradient method. Assume that H and each of the P_i's are similar and that all can perform the product of two vectors of length m, or the product of a vector by a scalar, in an average time of $\tau_1 m$, and an inner product in an average time of $\tau_2 m$. These times are assumed to include data transfers. Furthermore, assume that A is a 5-point discretization matrix. Taking into account the combinations (13)-(15), the computation of each block of $p(A)v$ can be achieved in time $6m\tau_1$ in each processor. Since the time to make the first block available, i.e. the start up time, is $k(6m\tau_1)$ and since we have ν blocks to treat, $p(A)v$ can be computed in time $6\tau_1 mk + 6\tau_1 m\nu = [6\tau_1 + k/\nu]N$.

After computation of each block of the vector $t = p(A)v$ is completed, this block is sent to H to begin computing the inner products (t, v) and (t, t) of (26) and (27). Each partial inner product costs $m\tau_2$. However, assuming that this time is not larger than $6m\tau_1$, we see that the computation of the inner products can be overlapped with the computation of t. The only time that is really spent is the start up time in H, which is $m\tau_2$ for each of the inner products (26) and (27), i.e. $2m\tau_2$ altogether. The rest of the conjugate gradient calculations (29), (30), (31) consumes a time of $3\rho_1 N$. Therefore, an estimate of the time spent to perform one iteration of the polynomial preconditioned conjugate gradient algorithm is

$$\tau = N[(9 + k/\nu)\tau_1 + (2/\nu)\tau_2].$$

With a single processor, a conjugate gradient step would require $5\tau_1 N$ (for the matrix by vector product) $+ 2\tau_2 N$ (for the inner products (18), (22)) $+ 3\tau_1 N$ (for (19), (20), (23)) which sums up to $8\tau_1 N + 2\tau_2 N$. This is slightly larger than the previous

case when $k \ll \nu$ and τ_1 and τ_2 are of the same order. Hence the speed up is of the same order as the ratio of the total number of preconditioned conjugate gradient steps over that of the nonpreconditioned conjugate gradient steps. Although it is difficult to a priori estimate this ratio, we can say that it is of the form $\gamma(k+1)$ where γ is a scalar larger than one. Often γ will be between 1 and 2, sometimes close to one. In an example shown in § 4, γ is around 1.2.

The above comparison uses a very simple model. An advantage of the multiprocessor described above and not stressed in the comparison is that the matrix is accessed only once to perform k matrix by vector products, and then moved in a uniform way. With a single processor we would have to access the matrix k times to perform k similar matrix by vector products. This will be reflected by smaller times τ_1 and τ_2 in a multiprocessor environment than in a single processor environment, and this was not taken into account in the above simplistic comparison.

4. Numerical experiments. In this section we describe a few numerical experiments in order to point out some additional facts about polynomial preconditionings. All tests have been performed in double precision on a VAX-11-700 computer for which the double precision mantissa is 56. The least squares polynomials referred to in this section are those associated with the Jacobi weight with $\alpha = \frac{1}{2}$ and $\beta = -\frac{1}{2}$ given as an example in § 2.

4.1. Varying the degree k. The purpose of the first experiment is to illustrate the behaviour of the polynomial preconditioned conjugate gradient method as the degree of s_{k-1} varies. Consider the 1200×1200 block tridiagonal matrix:

$$A = \text{Block-Tridiag}\,[B_i^T, A_i, B_{i+1}], \qquad i = 1, \cdots \nu$$

$$\text{with } A_i = \text{Tridiag}\,[-1, 4, -1] \text{ and } B_i = \text{Diag}\,(-1),$$

resulting from the 5-point discretization of the Laplacian operator on a rectangle. The dimension of each of the blocks is $m = 40$, i.e. the block-dimension of A is $\nu = 30$. We tested the polynomial preconditioned conjugate gradient method on the system $Ax = f$ and have used for a and b the Gershgorin values $a = 0$, $b = 8$. The right-hand side is chosen to be $f = Ae$, where $e = (1, 1, 1, \cdots 1)^T$. The initial vector is a random vector. The algorithm is stopped as soon as the approximate solution $x^{(j)}$ satisfies

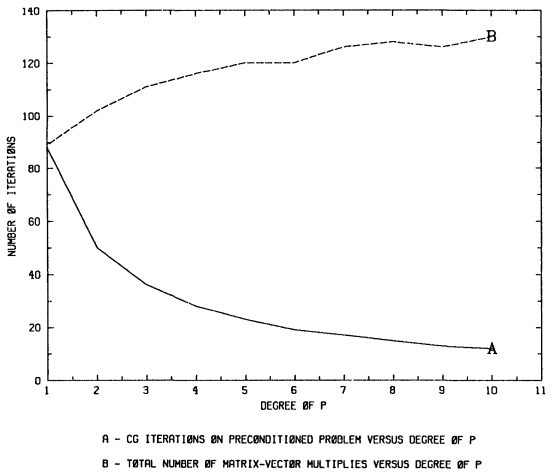

FIG. 4.1. *Iterations versus degree of preconditioning.*

$\|f - Ax^{(j)}\| / \|f - Ax^{(0)}\| \leqq \varepsilon$, where $\varepsilon = 10^{-5}$. These residual norms are actually computed at each (preconditioned) conjugate gradient step. Note that in a realistic implementation we need not compute these actual residual vectors because we can either use the generalized residual vectors $r^{(j)} = s_k(A)(f - Ax^{(j)})$ or use a classical implementation of the preconditioned conjugate gradient algorithm in which both the actual residual and the generalized residual are available, at the expense of a little more storage, e.g. see [12]. The plot of Fig. 4.1, shows the number of iterations for all values of the degree k of $p_k(A) \equiv As_{k-1}(A)$ between 1 and 10. Note that $k = 1$ means that s_{k-1} is a constant and corresponds to the nonpreconditioned conjugate gradient algorithm.

On the same figure we have plotted the total number of matrix by vector multiplications needed for convergence, i.e. the numbers $k(IT+1)$ where IT is the number of iterations. Note that except for the first 3 degrees, the total number of matrix by vector multiplications changes relatively little which is why the number of iterations drops like $1/k$ as k increases. The polynomials s_{k-1} have been determined by the kernel polynomial formulation described in § 2.2.

4.2. Varying the parameter *a*. In the next experiment we illustrate the behaviour of the polynomial preconditioned conjugate gradient method as the parameter a varies. It is known that the performance of the Chebyshev iteration algorithm [6], [7] depends critically on a being an accurate approximation of the smallest eigenvalue λ_1. We would like to show that the number of steps of the polynomial preconditioned conjugate gradient method depends very little on the accuracy of a as an approximation to the eigenvalue λ_1. The linear system tested is the same as above, and so are the tolerance ε and the initial vector x_0. The value of the parameter b is fixed to 8 and a is varied from 0.0 to 1.3. The larger values of a are more widely spaced than the first ones. We have tried three different degrees: $k = 3$, $k = \beta$ and $k = 10$. The results are shown in Fig. 4.2 in which we plot the total number of CG iterations to achieve convergence. Notice that around the smallest eigenvalue which is $\lambda_1 \approx 1.613 \times 10^{-2}$, there is little change in using different values of a. Amazingly, even for quite large values of a, the performance is little affected. For $k = 3$, for example, there is hardly any difference in performance for $a = 0.0$ through $a = 1.3$. We attribute this phenomenon to the fact that

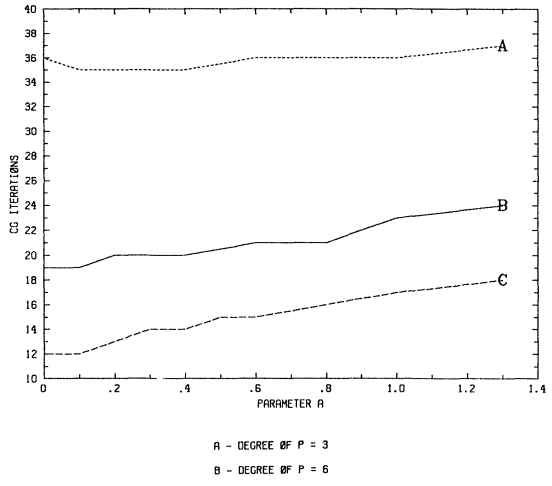

FIG. 4.2. *Behavior of the least squares polynomial preconditioning as the parameter a varies.*

the polynomials change very little when the parameter a moves around the origin to its right. We believe that this phenomenon will also be verified for other choices of the Jacobi weights such as the Legendre weight although this remains to be verified. This would mean that we expect the preconditionings suggested here to perform as well as those of [8] in spite of the fact that they require no eigenvalue computation.

4.3. Comparison with Chebyshev polynomial preconditionings. The next experiment compares Chebyshev polynomial preconditionings with the least squares polynomial preconditionings and reveals an interesting phenomenon which does not seem to have been pointed out in the literature. Let us consider the problem tested in § 4.1 with the same dimension $N = 1{,}200$, the same right-hand side f and initial vector x_0 and the same stopping criterion. If we try the Chebyshev polynomial preconditioning with $a = 0.016 \approx \lambda_1$ and $b = 7.984 \approx \lambda_N$, we find that for a polynomial $\lambda s(\lambda)$ of degree 5, the method converges in a total of 165 matrix by vector multiplications versus 120 for the least squares polynomial preconditioning using $a = 0$, $b = 8$ as is reported in § 4.1.

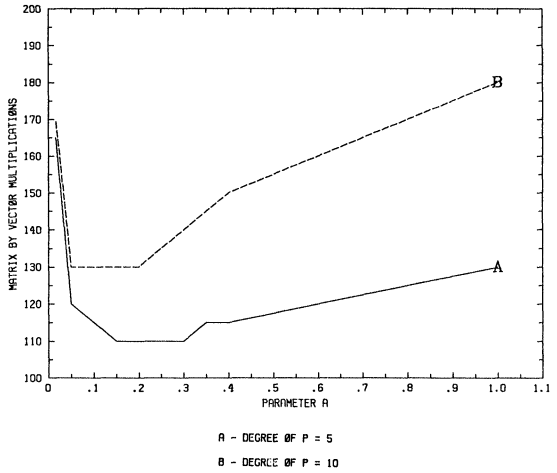

FIG. 4.3. *Behavior of Chebyshev polynomial preconditioning as the parameter a varies.*

Redoing the experiment with several different values for the parameter a, b being fixed, we find, to our surprise, that $a = \lambda_1$ is not the best possible value for a. Indeed, Fig. 4.3 shows that as a increases the performance improves drastically in the beginning and then starts deteriorating slowly as a becomes too large. In the plot the value of the parameter b is fixed to be 7.984 throughout and a is varied from 0 to 1. The best value for a is around $a \approx 0.2$ which yields convergence in a total of 110 matrix by vector multiplications. This phenomenon can be explained as follows. By taking a slightly larger λ_1, we obtain a matrix $As(A)$ that has a few "large" eigenvalues around one but the remainder of the eigenvalues are smaller than with $a = \lambda_1$. This situation is more favourable for the conjugate gradient method than when there are no "large" eigenvalues but the whole spectrum is wider; this occurs when a moves towards the origin. The same experiment is repeated for the degree $k = 10$. The conclusions to be drawn from these experiments are the following:

1. The usual optimal parameters $a = \lambda_1$ and $b = \lambda_N$ used in Chebyshev iteration are no longer optimal in Chebyshev polynomial preconditioned conjugate gradient method. When b is fixed, the true optimal parameter a is likely to be larger than λ_1. We do not know of any way of determining the actual optimal parameters a and b.

2. The least squares polynomial preconditioning with the simple parameters $a=0$, $b=8$ given by a Gershgorin argument performs nearly as well as the best performing Chebyshev polynomial preconditionings using the parameters $a=0.2$ for $k=5$ and $a=0.1$ for $k=10$.

Note that these observations are not isolated. Therefore, not only is the Chebyshev polynomial approach difficult to optimize but even when optimized, the result is not significantly better than the simpler least squares polynomial approach. Using the eigenvalues λ_1 and λ_N as optimal parameters for Chebyshev preconditioning may yield poor convergence.

Finally we would like to close this section by mentioning that G. Meurant [13] recently performed some interesting experiments on the CRAY-1 and CYBER 205 vector machines which show that polynomial preconditionings of the type proposed in this paper compare quite well with other vector machine oriented preconditionings. His final conclusion, however, is that there is no winner as performances depend heavily on the architecture as well as the data. An important advantage of polynomial preconditionings is that whereas classical preconditionings must start by computing an approximate factorization, polynomial preconditionings do not require any sort of preprocessing. It should be pointed out that in most cases the computation of an incomplete factorization is *not* a vectorizable process, and can take a substantial portion of the overall computing time. The timings reported in [13] do not account for preprocessings. Two other advantages of the least squares polynomial preconditionings described in this paper are their simplicity and their generality.

5. Appendix. This appendix gives the least squares polynomials up to the degree 10, when we use the Jacobi weights with $\alpha=\frac{1}{2}$ and $\beta=-\frac{1}{2}$. We chose to develop the formulas for the interval $[0, 4]$ as they are simpler to write down. Moreover, we rescaled the polynomials for simplicity. More precisely the kth degree polynomial in the list below must be multiplied by the factor $4/(3+2k)$ to obtain the least squares polynomials s_k as defined in § 2. Note, however that a scaling factor is unimportant if one wants to use these polynomials for preconditioning. The polynomials have been obtained by a simple FORTRAN program based on the kernel polynomial approach described in § 2.2. Finally, recall that a change of variable is necessary if $b\neq 4$ to map the interval $[0, b]$ into $[0,4]$, i.e. for a general interval one must take the polynomial $s_k(4\lambda/b)$.

$$s_1(\lambda)=5-\lambda,$$

$$s_2(\lambda)=14-7\lambda+\lambda^2,$$

$$s_3(\lambda)=30-27\lambda+9\lambda^2-\lambda^3,$$

$$s_4(\lambda)=55-77\lambda+44\lambda^2-11\lambda^3+\lambda^4,$$

$$s_5(\lambda)=91-182\lambda+156\lambda^2-65\lambda^3+13\lambda^4-\lambda^5,$$

$$s_6(\lambda)=140-378\lambda+450\lambda^2-275\lambda^3+90\lambda^4-15\lambda^5+\lambda^6,$$

$$s_7(\lambda)=204-714\lambda+1122\lambda^2-935\lambda^3+442\lambda^4-119\lambda^5+17\lambda^6-\lambda^7,$$

$$s_8(\lambda)=285-1{,}254\lambda+2{,}508\lambda^2-2{,}717\lambda^3+1{,}729\lambda^4-665\lambda^5+152\lambda^6-19\lambda^7+\lambda^8,$$

$$s_9(\lambda)=385-2{,}079\lambda+5{,}148\lambda^2-7{,}007\lambda^3+5{,}733\lambda^4-2{,}940\lambda^5+952\lambda^6-189\lambda^7+21\lambda^8-\lambda^9,$$

$$s_{10}(\lambda)=506-3{,}289\lambda+9{,}867\lambda^2-16{,}445\lambda^3+16{,}744\lambda^4-10{,}948\lambda^5+4{,}692\lambda^6-1{,}311\lambda^7+230\lambda^8-23\lambda^9+\lambda^{10}.$$

Acknowledgments. The author has benefited from valuable discussions with Prof. Martin Schulz, Prof. Stan Eisenstat and Dr. Gerard Meurant.

REFERENCES

[1] L. M. ADAMS, *Iterative algorithms for large sparse linear systems on parallel computers*, Ph.D. thesis, Applied Mathematics, Univ. of Virginia, Blacksburg, 1982. Also available as NASA Contractor Report #166027.

[2] C. C. CHENEY, *Introduction to Approximation Theory*, McGraw-Hill, New York, 1966.

[3] P. CONCUS, G. H. GOLUB AND G. MEURANT, *Block preconditioning for the conjugate gradient method*, Technical Report LBL-14856, Lawrence Berkeley Lab., 1982; this Journal, 6 (1985), pp. 220–252.

[4] P. H. DAVIS, *Interpolation and Approximation*, Blaisdell, Waltham, MA, 1963.

[5] P. F. DUBOIS, A. GREENBAUM AND G. H. RODRIGUE, *Approximating the inverse of a matrix for use on iterative algorithms on vectors processors*, Computing, 22 (1979), pp. 257–268.

[6] G. H. GOLUB AND R. S. VARGA, *Chebyshev semi-iterative methods, successive overrelaxation iterative methods and second order Richardson iterative methods*, Numer. Math., 3 (1961), pp. 147–168.

[7] A. L. HAGEMAN AND D. M. YOUNG, *Applied Iterative Methods*, Academic Press, New York, 1981.

[8] O. G. JOHNSON, C. A. MICCHELLI AND G. PAUL, *Polynomial preconditions for conjugate gradient calculations*, SIAM J. Numer. Anal., 20 (1983), pp. 362–376.

[9] L. JOHNSSON, *Highly concurrent algorithms for solving linear systems of equations*, in Elliptic Problem Solvers II, Proceedings of the Elliptic Problem Solvers Conference, Monterey CA, Jan. 10–12, 1983, G. N. Birkhoff and A. Schoenstadt, eds., Academic Press, New York, 1983, pp. 105–126.

[10] T. L. JORDAN, *Conjugate gradient preconditioners for vector and parallel processors*, in Elliptic Problem Solvers II, Proceedings of the Elliptic Problem Solvers Conference, Monterey CA, Jan. 10–12, 1983, G. N. Birkhoff and A. Schoenstadt, eds., Academic Press, New York, 1983, pp. 127–139.

[11] ———, *A guide to parallel computation and some Cray-1 experiences*, in Parallel Computations, Garry Rodrigue, ed., Academic Press, New York, 1982, pp. 1–50.

[12] J. A. MEIJERINK AND H. A. VAN DER VORST, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

[13] G. MEURANT, *Vector preconditioning for the conjugate gradient on the CRAY-1 and CDC Cyber 205*, in Proceedings of the 6th International Colloquium on Scientific and Technical Computation, Versailles, France, 1984, INRIA ed., North-Holland, Amsterdam, 1984.

[14] P. G. NEVAI, *Distribution of zeros of orthogonal polynomials*, Trans. Amer. Math. Soc., 249, 2 (1979), pp. 241–261.

[15] C. J. PFEIFER, *Data flow and storage allocation for the PDQ-5 program on the Philco-2000*, Comm. ACM, 6 (1963), pp. 365–366.

[16] H. RUTISHAUSER, *Theory of gradient methods*, in Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems, Institute of Applied Mathematics, Zurich, Basel-Stuttgart, 1959, pp. 24–49.

[17] Y. SAAD, *Iterative solution of indefinite symmetric systems by methods using orthogonal polynomials over two disjoint intervals*, SIAM J. Numer. Anal., 20 (1983), pp. 784–811.

[18] D. C. SMOLARSKI, *Optimum semi-iterative methods for the solution of any linear algebraic system with a square matrix*, Ph.D. thesis, Technical Report UIUCDCS-R-81-1077, Univ. Illinois, Urbana-Champaign, 1981.

[19] SO CHENG CHEN, *Polynomial scaling in the conjugate gradient method and related topics in matrix scaling*, Ph.D. thesis, Technical Report CS-82-23, Pennsylvania State Univ., University Park, 1982.

[20] E. L. STIEFEL, *Kernel polynomials in linear algebra and their applications*, U.S. NBS Applied Math. Series, 49 (1958), pp. 1–24.

[21] H. A. VAN DER VORST, *A vectorizable version of some ICCG methods*, this Journal, 3 (1982), pp. 350–356.

[22] J. VAN ROSENDALE, *Minimizing inner product data dependencies in conjugate gradient iteration*, Technical Report 172178, ICASE-NASA, 1983.