An Efficient Implementation for SSOR and Incomplete Factorization Preconditionings

Randolph E. Bank[1] and Craig C. Douglas[2]

Research Report YALEU/DCS/RR-323

[1] Department of Mathematics, University of California at San Diego
[2] Department of Computer Science, Duke University
This research was supported in part by ONR Grant N00014-82-K-0197[1], ONR Grant N00014-82-K-0184[2], FSU-ONR Grant F1.N00014-80-C-0076[2], and NSF MCS-8106181[2]. This work was done while the second author was at the Yale University Department of Computer Science.

Summary: We investigate methods for efficiently implementing a class of incomplete factorization preconditioners which includes Symmetric Gauss Seidel [9], SSOR [9], generalized SSOR [1], Dupont Kendall Rachford [4], ICCG(0) [7], and MICCG(0) [6]. Our techniques can be extended to similar methods for nonsymmetric matrices.

1 Symmetric Matrices

We consider the solution of the linear system

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

where A is an NxN symmetric, positive definite matrix and $A = D - L - L^T$, where D is diagonal and L is strictly lower triangular. Such linear systems are often solved by iterative methods, for example, Symmetric Gauss Seidel [9], SSOR [9], generalized SSOR [1], Dupont Kendall Rachford [4], ICCG(0) [7], and MICCG(0) [6].

A single step of a basic (unaccelerated) iterative method, starting from an initial guess $\hat{x}$ can be written as

$$\begin{aligned} &\text{(a) Solve } B\delta = r \equiv b - A\hat{x} \\ &\text{(b) Set } \hat{x} = \hat{x} + \delta \end{aligned} \tag{2}$$

For the iterative methods cited before, B is symmetric, positive definite and can be written as

$$B = (\tilde{D} - L)\tilde{D}^{-1}(\tilde{D} - L^T) \tag{3}$$

Since A and B are symmetric and positive definite, the underlying iterative scheme (2) can be accelerated by standard techniques such as Chebyshev, conjugate gradients, and conjugate residuals.

Let $\Delta = D - \tilde{D}$ be a diagonal matrix and let M denote the computational cost (in floating point multiplies) of forming the matrix-vector product Ax. The obvious approach to implementing the basic iterative step (2)(a) apparently requires $2M + O(N)$ multiplies. Our goal is to reduce this to $M + O(N)$. See Eisenstat [5] for a different solution to the same problem.

The basic idea for accomplishing this reduction in cost is embodied in the following procedure for solving

$$Bz = \alpha(r + Lv), \tag{4}$$

where r and v are input vectors and α is a scalar. This is solved using the process

$$\begin{aligned}
&\text{(a) } \tilde{D}w = \alpha r + L(\alpha v + w) \equiv q \\
&\text{(b) } (\tilde{D} - L^T)z = q. \qquad (5) \\
&\text{(c) } r - Az = r - q + \Delta z + Lz.
\end{aligned}$$

Despite the apparently implicit nature of (5)(a), it can be solved easily for w. In fact, w itself need not be saved in any form since q is the important vector computed in this equation. Computing q and z, given r and v, requires M + 3N multiplies (multiplies and divides). Computing $r-Az$ requires N multiplications if we represent the vector implicitly in terms of $r-q+\Delta z$ and z.

The basic algorithm, using fixed acceleration parameters τ_i, $1 \leq i \leq m$, is given by

Algorithm 1: (Fixed Acceleration Parameters - Preliminary)

(1) $r_0 = b - Ax_0$

(2) For i = 1 to m

(a) $Bz_i = \tau_i^{-1} r_i$

(b) $x_i = x_{i-1} + z_i$

(c) $r_i = r_{i-1} + Az_i$

Straightforward implementation of Algorithm 1 requires 2M + 2N multiplies. Using the process in (5) we can reformulate this algorithm as

Algorithm 2: (Fixed Acceleration Parameters - Final)

(1) $r_0 = b - Dx_0 + L^T x_0$

(2) For i = 1 to m

(a) $\tilde{D}w_i = \tau_i^{-1} r_i + L(\tau_i^{-1} x_{i-1} + w_i) \equiv q_i$

(b) $(\tilde{D} - L^T)z_i = q_i$

(c) $r_i = r_{i-1} - q_i + \Delta z$

(d) $x_i = x_{i-1} + z_i$

(3) $\hat{r}_m = r_m + Lx_m \equiv b - Ax_m$

The computational cost of the inner loops of Algorithm 2 is at most M + 4N multiplies. If we do not accelerate at all ($\tau_i = 1$), the cost is reduced to at most M + 2N multiplies. Algorithm 2 requires one additional N-vector for storing q_i and z_i (which may share the same space). The vector r_i can be stored over the original right hand side b.

This technique is not limited to fixed acceleration parameters. For instance, the preconditioned conjugate gradient algorithm is given by

Algorithm 3: (PCG - Preliminary)

(1) $r_0 = b - Ax_0$

(2) $p_0 = 0$

(3) For $i = 1$ to m

(a) $Bz_i = r_{i-1}$

(b) $\gamma_i = z_i^T r_{i-1}$; $\beta_i = \gamma_i \,/\, \gamma_{i-1}$; $\beta_1 = 0$

(c) $p_i = z_i + \beta_i p_{i-1}$

(d) $\alpha_i = \gamma_i \,/\, p_i^T A p_i$

(e) $x_i = x_{i-1} + \alpha_i p_i$

(f) $r_i = r_{i-1} - \alpha_i A p_i$

In order to reduce the number of matrix multiplies to one, we implicitly represent Ap_i as well as the residual. Thus, we set $Ap_i = v_i - Lp_i$. Then we can reformulate this algorithm as

Algorithm 4: (PCG - Final)

(1) $r_0 = b - Dx_0 + L^T x_0$

(2) $p_0 = v_0 = 0$

(3) For $i = 1$ to m

(a) $\tilde{D} w_i = r_{i-1} + L(x_{i-1} + w_i) \equiv q_i$

(b) $\gamma_i = q_i^T w_i$; $\beta_i = \gamma_i \,/\, \gamma_{i-1}$; $\beta_1 = 0$

(c) $(\tilde{D} - L^T) z_i = q_i$

(d) $v_i = q_i + \beta_i v_{i-1} + \Delta z_i$

(e) $p_i = z_i + \beta_i p_{i-1}$

(f) $\alpha_i = \gamma_i \,/\, (p_i^T(v_i + v_i - Dp_i))$

(g) $r_i = r_{i-1} - \alpha_i v_i$

(h) $x_i = x_{i-1} + \alpha_i p_i$

(4) $\hat{r}_m = r_m + Lx_m \equiv b - Ax_m$

To implement Algorithm 4, we need three temporary vectors of length N, one each for v_i, p_i, and q_i. The vector z_i can share the space of q_i. As before, r_i can be stored over the right hand side b. The inner loops of Algorithm 4 requires at most M + 8N multiplies per iteration.

2 Nonsymmetric Matrices

Assume A is an NxN nonsymmetric stiffness matrix and $A = D-L-U$, where D is diagonal, L is strictly lower triangular, and U is strictly upper triangular. Then the matrix B corresponding to the incomplete LDU factorization class of smoothers is

$$B = (\tilde{T}-L)\tilde{S}^{-1}(\tilde{D}-U) \qquad (6)$$

where $\tilde{D}$, $\tilde{S}$, and $\tilde{T}$ are diagonal.

The algorithms of the last section can be extended to handle B of the form (6). Given the linear system (4), we replace (5) by

(a) $\tilde{T}w = \alpha r + L(\alpha v+w)$

(b) $q = \tilde{S}w$

(c) $(\tilde{D}-U)z = q$.

(d) $r - Az = r-q + \Delta z + Lz$.

The generalization of Algorithm 2 requires M + O(N) multiplies. Unfortunately, some adaptive schemes, like Orthomin(1) [8]) or Orthodir(1) [10], appear to require 1.5M + O(N) multiplies (assuming the cost of multiplying by L and U are the same). This is because the identity

$$x^TLx = x^TL^Tx,$$

which is implicitly used in Algorithm 4, line 3f, does not necessarily hold when U replaces L^T. Thus, it appears we need an extra half matrix multiply to form the equivalent of Ap for purposes of computing inner products.

3 Final Remarks

Table 1 contains a summary of the cost of each algorithm. The column in Table 1 corresponding to the special case of $\Delta = 0{\cdot}I$ is important since it corresponds to the Symmetric Gauss Seidel preconditioner. In practice, variants of the Gauss Seidel iteration are among the most popular smoothing iterations used in multigrid codes [2, 3]. Since the cost of smoothing is usually a major expense in a multigrid code, reducing the number of matrix multiplies can significantly reduce the overall computational cost.

Although the cost of the adaptive acceleration in Algorithm 4 is somewhat higher than the cost for the fixed acceleration in Algorithm 2 in terms of multiplications, the actual cost may not be that much greater. In particular, if A is stored in a general sparse format, then the effective cost of floating point operations of a matrix multiply is normally somewhat higher than those for inner products or scalar vector multiplies, because operations corresponding to matrix multiplication are usually done in N short loops and accessing each nonzero of A involves some sort of indirect addressing.

Table 1: Inner Loop Operation Counts for the Preconditionings

Algorithm / Form:	Preliminary	Final	Final with $\Delta = 0 \cdot I$
Unaccelerated	2M + N	M + 2N	M + N
Accelerated/Fixed	2M + 2N	M + 4N	M + 3N
PCG	2M + 5N	M + 8N	M + 7N

Bibliography

[1] O. Axelsson, A generalized SSOR, *BIT*, 13 (1972), pp. 443-467.

[2] R. E. Bank, *PLTMG User's Guide*, Technical Report, Univeristy of California at San Diego, 1981.

[3] C. Douglas, A multigrid optimal order solver for elliptic boundary value problems: the finite difference case, R. Vichnevetsky and R. S. Stepleman, eds., *Advances in Computer Methods for Partial Differential Equations - V*, IMACS, New Brunswick, NJ, 1984, pp. 369-374.

[4] T. Dupont, R. P. Kendall, and H. H. Rachford, Jr., An approximate factorization procedure for solving self-adjoint elliptic difference equations, *SIAM J. Numer. Anal.*, 5 (1968), pp. 559-573.

[5] S. C. Eisenstat, Efficient implementation of a class of conjugate gradient methods, *SIAM J. Sci. Stat. Comp.*, 2 (1981), pp. 1-4.

[6] I. Gustafsson, A class of first order factorization methods, *BIT*, 18 (1978), pp. 142-156.

[7] J. A. Meijerink and H. A. van der Vorst, An iterative solution method for linear systems of which the the coefficient matrix is a symmetric M-matrix, *Math. Comp.*, 31 (1977), pp. 148-162.

[8] P. K. W. Vinsome, Orthomin, an iterative method for solving sparse sets of simultaneous linear equations, Society of Petroleum Engineers of AIME, *Proceedings of the Fourth Symposium on Reservoir Simulation*, 1976, pp. 149-159.

[9] D. M. Young, *Iterative Soultion of Large Linear Systems*, Academic Press, New York, 1971.

[10] D. M. Young and K. C. Jea, Generalized conjugate gradient acceleration of nonsymmetrizable iterative methods, *Linear Algebra and Its Applications*, 24 (1980), pp. 159-194.