1992

P.M. de Zeeuw

Incomplete line LU for discretized coupled PDEs as preconditioner in Bi-CGSTAB

Department of Numerical Mathematics Report NM-R9213 July

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Incomplete Line LU for Discretized Coupled PDEs as Preconditioner in Bi-CGSTAB

P.M. de Zeeuw
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

A variant of the recently developed Bi-CGSTAB method is applied for solving the linear systems that typically occur when applying the Newton method to a discretized set of coupled elliptic partial differential equations in two dimensions. The Incomplete Line LU relaxation is generalized for the case of coupled equations and applied as preconditioner. Both an eligible stopping and restart criterion are developed for the Bi-CGSTAB method. Numerical experiments are performed for problems stemming from the area of semiconductor modeling and an aquifer problem. For the latter problem a comparison is made with an existing multigrid algorithm in which Incomplete Line LU relaxation is used as smoothing procedure.

1980 Mathematics Subject Classification: 65F10, 65N20, 65H10.
Key Words and Phrases: Bi-CGSTAB, coupled PDEs, ILLU, ILU, incomplete decompositions, iterative solver, multigrid method, nonsymmetric linear systems, preconditioner, smoother, semiconductor equations, sparse linear systems.

1 Introduction

Recently Van der Vorst introduced the Bi-CGSTAB method [22, 9] which is a reformulation of the method of Induced Dimension Reduction (IDR) as developed by Sonneveld [23, 19]. It is indicated that Bi-CGSTAB cures the irregular convergence behaviour of the Conjugate Gradient-Squared (CG-S) method and, in many cases, it converges considerably faster. In Section 2 we introduce a generalisation of Incomplete Line LU (ILLU) as a preconditioner within Bi-CGSTAB, suitable for the non-scalar case. With this preconditioner we applied Bi-CGSTAB to real-life problems in process and device modeling of semiconductors. In this context of hard testproblems seeming technical questions about stopping and restarting criterions turn out to be important. Section 3 explains why a particular variant of Bi-CGSTAB is favourable from this viewpoint and how to embed this variant within the Newton method. The algorithm has been used to improve the efficiency of TRENDY [17, 24], an integrated programme for IC process and device simulation, developed by the Integrated Circuits and Electronics group of the University of Twente, The Netherlands. Numerical results for problems from this application appear in Section 4.

For scalar problems, the ILLU-relaxation is also used as an efficient and robust smoothing procedure within multigrid methods. In Section 5 a comparison is made between our version of Bi-CGSTAB and the multigrid program MGD9V [25]. In Section 6 conclusions are summarized.

Report NM-R9213
ISSN 0169-0388
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

2 ILLU for discretized coupled PDEs

The ILLU decomposition has been originated by Underwood [21], and has also been proposed and elaborated upon by Concus, Golub and Meurant [7], Axelsson [2, 3], Meijerink [15] and others. This ILLU-method may serve both as a smoother [14, 13, 20, 25] in a multigrid context and as a preconditioner for conjugate gradient type methods e.g. [20]. In [13, 20] an extensive description of the method can be found. The method is of interest because it is shown to be more robust than incomplete point LU (ILU) and often also more efficient (see [12, 20]) though more complicated to implement. In the context of multigrid methods excellent convergence rates are reported for anisotropic diffusion and convection-diffusion equations. In this section we show how to generalize ILLU for a scalar PDE to a set of coupled PDEs.

We assume to have a rectangular computational grid, n_x denotes the number of volumes in the x-direction in the case of a cell-centered discretization or the number of vertical lines in the case of a vertex-centered discretization. Likewise we define n_y, corresponding with the y-direction. Further we assume the common five point coupling (as with central differences). With these assumptions, after discretization (and linearization) a set of n coupled PDEs is represented by a block tridiagonal linear system of the form

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

where

$$A = \begin{pmatrix} D_1 & U_1 & & & & \\ L_2 & D_2 & U_2 & & & \\ & L_3 & D_3 & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & & \cdot & D_{n_y} \end{pmatrix}. \tag{2}$$

The block D_j has the tridiagonal form:

$$D_j = \begin{pmatrix} d_{1j} & u_{1j} & & & & \\ l_{2j} & d_{2j} & u_{2j} & & & \\ & l_{3j} & d_{3j} & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & & \cdot & d_{n_x j} \end{pmatrix}. \tag{3}$$

The blocks L_j and U_j are diagonal-matrices and of the same dimension as D_j. The entries l_{ij}, d_{ij} and u_{ij} of D_j are just scalars if $n = 1$, but for $n > 1$ they are $n \times n$-matrices. In this way we recognize a three-level blockstructure: matrix A has a dimension of n_y blocks each with a dimension of n_x blocks of dimension n. Block matrices combine just as matrices with scalar entries [10, § 1.3]. Because of this we can apply all algebra for the scalar case also for the case $n > 1$. The complete and exact factorization of the matrix A can be written in the form (see [6, § 8.5])

$$\begin{pmatrix} I & & & & \\ L_2\overline{D}_1^{-1} & I & & & \\ & L_3\overline{D}_2^{-1} & I & & \\ & & \cdot & \cdot & \\ & & & \cdot & I \end{pmatrix} \begin{pmatrix} \overline{D}_1 & U_1 & & & \\ & \overline{D}_2 & U_2 & & \\ & & \overline{D}_3 & U_3 & \\ & & & \cdot & \cdot \\ & & & & \overline{D}_{n_y} \end{pmatrix} \tag{4}$$

where

$$\overline{D}_1 = D_1, \tag{5}$$

$$\overline{D}_j = D_j - L_j \overline{D}_{j-1}^{-1} U_{j-1}, \quad j = 2(1)n_y. \tag{6}$$

The point of the ILLU-method is to make an incomplete factorization of A simply by substituting formulae (5) and (6) by

$$\overline{D}_1 = D_1, \tag{7}$$

$$\overline{D}_j = D_j - \mathbf{tridiag}(L_j \overline{D}_{j-1}^{-1} U_{j-1}), \quad j = 2(1)n_y. \tag{8}$$

The operator **tridiag** restricts a block (by clipping) to the sparsity pattern of the D_j. The entries of the matrices in formulae (8) can be blocks of dimension n instead of scalars. At the actual working-out of these formulae, as known and described (e.g. [13]) for the scalar case, we have to replace operations on scalars x and y by operations on matrices X and Y of dimension n as follows [11]:

$$\begin{aligned} x \pm y &\longrightarrow X \pm Y \\ xy &\longrightarrow XY \\ x/y &\longrightarrow XY^{-1} \end{aligned}$$

However, an important difference is that multiplication is no longer commutative and therefore the working-out of formulae (8) needs careful overhauling.

Performing one ILLU-relaxation sweep for the approximate solution of (1) is denoted by $RELAX(A, x, b)$. It requires the following steps:

ILLU-sweep:

$$\begin{aligned} &r = b - Ax; \\ &\quad z_1 = r_1; \\ &\quad z_j = r_j - L_j \overline{D}_{j-1}^{-1} z_{j-1}, \quad j = 2(1)n_y; \\ &\quad c_{n_y} = \overline{D}_{n_y}^{-1} z_{n_y}; \\ &\quad c_j = \overline{D}_j^{-1}(z_j - U_j c_{j+1}), \quad j = n_y - 1(-1)1; \\ &x = x + c; \end{aligned}$$

Every matrix $\overline{D}_j$ is stored by means of its exact decomposition (block bidiagonal matrices) so multiplication by $\overline{D}_j^{-1}$ requires just a forward and backward substitution. In Figure 1 degenerated forms of matrix A are shown, symbolically, for which ILLU turns into an exact linear solver. When a $\circ$ occurs within the stencil notation this means that all matrices of dimension n on the corresponding diagonal have to be zero, when a $\bullet$ occurs those matrices are allowed to be unequal to zero matrices. The statement follows immediately from comparison of (5,6) and (7,8). The complexity of the ILLU-decomposition amounts to a total of

$$13 n_y n_x n^3 \text{flops}$$

where a flop is the amount of work associated with a multiplication joined with an addition. The storage requirements are at least

$$3 n_y n_x n^2 \text{reals}.$$

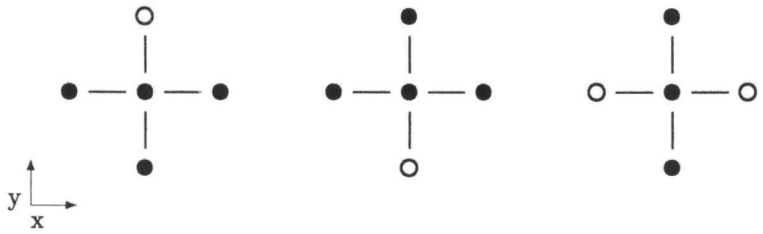

Figure 1: Degenerated forms of A for which ILLU is an exact linear solver.

For comparison, the complete factorization according to (5) and (6) takes

$$n_y n_x^3 n^3 \text{flops}$$

and

$$n_y n_x^2 n^2 \text{reals}$$

respectively.

3 A particular variant of Bi-CGSTAB

3.1 Description

We consider the linear system (1). Let r denote a residual and $\underline{0}$ the zero vector. A simplified version of the Bi-CGSTAB algorithm for solving this system reads [22]:

Bi-CGSTAB:

x_0 is an initial guess;
$r_0 = b - Ax_0$;
$\rho_0 = \alpha = \omega_0 = 1$;
$v_0 = p_0 = \underline{0}$;
for $i = 1, 2, 3, ...$
  $\rho_i = (r_0, r_{i-1})$;
  $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$;
  $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$;
  $v_i = Ap_i$;
  $\alpha = \rho_i/(r_0, v_i)$;
  $s = r_{i-1} - \alpha v_i$;
  $t = As$;
  $\omega_i = (t, s)/(t, t)$;
  $x_i = x_{i-1} + \alpha p_i + \omega_i s$;
  $r_i = s - \omega_i t$;
end

At the i-th sweep this scheme delivers some approximation x_i of the solution x of (1) and the corresponding residual r_i.

The advantages of preconditioning are well-known and preconditioning from both sides by incomplete decompositions [16] is widely in use. For reasons to be explained in Section 3.2, 3.3 we decide in favour of preconditioning from the left. In order to obtain a thus preconditioned version of Bi-CGSTAB we apply the method to another equation (equivalent to (1)):

$$\tilde{A}x = \tilde{b} \tag{9}$$

where

$$\begin{aligned} \tilde{A} &= K^{-1}A \\ \tilde{b} &= K^{-1}b \end{aligned} \tag{10}$$

(K^{-1} is an approximate inverse of A to be defined later on). Further we write

$$\tilde{r}_i = K^{-1}r_i. \tag{11}$$

Direct application of Bi-CGSTAB to (9) leads to the following method (Bi-CGSTAB preconditioned from the left, first version)

left-Bi-CGSTAB-v1:

x_0 is an initial guess;
$r_0 = b - Ax_0$;
$\tilde{r}_0 = K^{-1}r_0$;
$\rho_0 = \alpha = \omega_0 = 1$;
$v_0 = p_0 = \underline{0}$;
for $i = 1, 2, 3, ...$
  $\rho_i = (\tilde{r}_0, \tilde{r}_{i-1})$;
  $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$;
  $p_i = \tilde{r}_{i-1} + \beta(p_{i-1} - \omega_{i-1}v_{i-1})$;
  $v_i = K^{-1}Ap_i$;
  $\alpha = \rho_i/(\tilde{r}_0, v_i)$;
  $s = \tilde{r}_{i-1} - \alpha v_i$;
  $t = K^{-1}As$;
  $\omega_i = (t, s)/(t, t)$;
  $x_i = x_{i-1} + \alpha p_i + \omega_i s$;
  $\tilde{r}_i = s - \omega_i t$;
end

Note that this method delivers the variable x_i corresponding to system (9) and therefore corresponding also to the original system (1). For K one may take the incomplete decomposition as developed in Section 2. Of course the actual computation is done implicitly by performing an ILLU-relaxation sweep. We generalise this feature by performing σ sweeps; if the ILLU-relaxation is convergent then an increasing σ corresponds to a better approximation of A^{-1} for K^{-1}.

At the first line of **left-Bi-CGSTAB-v1** we need an initial guess for x_0. From experience we know that starting with the zero solution the first ILLU-sweeps are usually efficient by strongly reducing high frequent components in the error and residual. Therefore we make a guess for x_0 by applying σ ILLU-sweeps to the zero solution. Thus we arrive at the following scheme (Bi-CGSTAB preconditioned from the left, second version)

left-Bi-CGSTAB-v2:

$x_0 = \underline{0}$;
to σ do $RELAX(A, x_0, b)$;
$r_0 = b - Ax_0$;
to σ do $RELAX(A, \tilde{r}_0, r_0)$;
$\rho_0 = \alpha = \omega_0 = 1$;
$v_0 = p_0 = \underline{0}$;
for $i = 1, 2, 3, ...$
$\rho_i = (\tilde{r}_0, \tilde{r}_{i-1})$;
$\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$;
$p_i = \tilde{r}_{i-1} + \beta(p_{i-1} - \omega_{i-1}v_{i-1})$;
$v_i = \underline{0}$; to σ do $RELAX(A, v_i, Ap_i)$
$\alpha = \rho_i/(\tilde{r}_0, v_i)$;
$s = \tilde{r}_{i-1} - \alpha v_i$;
$t = \underline{0}$; to σ do $RELAX(A, t, As)$
$\omega_i = (t, s)/(t, t)$;
$x_i = x_{i-1} + \alpha p_i + \omega_i s$;
$\tilde{r}_i = s - \omega_i t$;
end

Compared with preconditioning from both sides we gain a degree of freedom, for we can choose $\sigma > 1$. We have to compute $(2+2(\sigma-1))$ matrix-vector multiplications for each i. Therefore, choosing $\sigma = 2$ instead of $\sigma = 1$ roughly doubles the amount of work per sweep. Yet, various numerical experiments have indicated that $\sigma = 2$ provides a more efficient choice for this parameter because of faster convergence. Still higher values of σ decrease the efficiency. The numerical results reported in this paper, are obtained by the last scheme provided with a proper stopping and restarting criterion. These criterions will be the subject of the next sections.

3.2 Bi-CGSTAB and Newton

Commonly, seeking a solution of (1) is just one step in Newton's method for solving a system of nonlinear equations. It was already shown by Brussino and Sonnad [4] that preconditioned iterative methods have the potential for reducing dramatically the storage and CPU time required by direct methods, a potential which is growing with the size of the problem.

It is often argued (e.g. [8]) that, when a Newton-iterate is still far from the solution, e.g. at the first Newton-sweeps, it is justified to solve system (1) up to a limited accuracy. Of course, when the Newton-iterates get closer to the solution, one should increase this accuracy.

However, devising such an inexact Newton method in practice, may cause troubles. The obvious aim of an inexact method is to reduce the costs of an individual Newton-sweep, yet avoiding that the required number of Newton-sweeps increases (setting up a new Jacobian is expensive). Now let us consider the practice of semiconductor-modeling. The equations, encountered in this field, are highly nonlinear and therefore assumptions about smoothness of the nonlinear operator are hard to make. This raises a question about the applicability of the theory on inexact Newton methods. Parameters that have to be tuned, cast another doubt on inexact Newton methods. Such a set of tuned parameters may work fine for a limited set of problems, but there exists the danger of having to retune again and again for other problems. The

penalty consists of a growth of the number of Newtonsweeps compared with the exact Newton method. In practice we encountered examples of this phenomenon. Because, in practice, robustness of the method is paramount, we put severe demands to the desired tolerance. We simply emulate an exact linear solver by using an iterative solver performing an adequate number of iterations.

3.3 Stopping and restarting

We study the question how to tune our stopping and restart criterion within the Bi-CGSTAB algorithm. This question is not at all trivial. E.g. within the context of semiconductor modelling we know that the entries of A may differ by orders of magnitude. Generally speaking, this makes it hard to decide whether a residual is small or not. With a too pessimistic view of the residual we might iterate within Bi-CGSTAB without further convergence, with a too optimistic view of the residual we may be punished by an increasement of the number of Newtonsteps. In the context of semiconductor problems these seemingly technical considerations turned out to be crucial for the performance of the nonlinear solution process as a whole. In this respect, an important advantage of **left-Bi-CGSTAB-v2** is that $\tilde{r}_i$ is a properly *scaled* residual. In fact, applying ILLU we find $\tilde{r}_i$ to be a close approximation of the *error* rather than the residual. Suppose we are satisfied with a relative error δ in solution x_i:

$$\|x - x_i\| < \delta \|x\| \tag{12}$$

with respect to the 2-norm (x denotes the exact solution). If δ equals the *unit round-off* [10, § 2.4] of the computer, we emulate a direct linear solver. For $\sigma \geq 1$, K is an approximation of A and therefore

$$K^{-1} r_i \approx x - x_i \tag{13}$$

Because of definition (11) we find

$$\tilde{r}_i \approx x - x_i. \tag{14}$$

Further we note that because the cg-iterations in Bi-CGSTAB are preceded by σ ILLU-sweeps, the norm $\|x\|$ is represented well enough by $\|x_i\|$, even for low i. So, in practice, we can approximate the evaluation of inequality (12) by checking the inequality

$$\|\tilde{r}_i\| < \delta \|x_i\| \tag{15}$$

at the end of every iteration-sweep.

It might happen that $\rho_i \approx 0$ for some i, which makes the evaluation of β for $i+1$ unreliable or even incalculable. Therefore a restart criterion is needed: we make a restart, setting $x_0 = x_i$ at the very first line of the algorithm as soon as the inequality

$$|\rho_i| < \|\delta \tilde{r}_0\|^2 \tag{16}$$

is satisfied.

In addition to the above we have to reckon with the event of a 'lucky breakdown' when $s \approx \underline{0}$. We therefore check the inequality

$$\|s\| < \delta \|x_{i-1}\|. \tag{17}$$

In this particular case we set ω_i to zero. After this event stopcriterion (15) will be satisfied.

4 Numerical results

4.1 Problem 1

We consider an example of device self-heating, caused by currents through interconnect material like aluminium and tungsten [24]. Figure 2 shows the geometry of the device, the various dimensions have been indicated in μm. The following equations are solved:

$$\nabla \cdot \mathbf{J} = 0 \tag{18}$$

$$\mathbf{J} = -\sigma \nabla \psi \tag{19}$$

in conductors and

$$-\nabla \cdot \mathcal{K} \nabla T + \mathcal{C} \frac{\partial T}{\partial t} = H \tag{20}$$

in all materials. The heat generation term H equals $\mathbf{J} \cdot \mathbf{J}/\sigma$ in conductors and zero

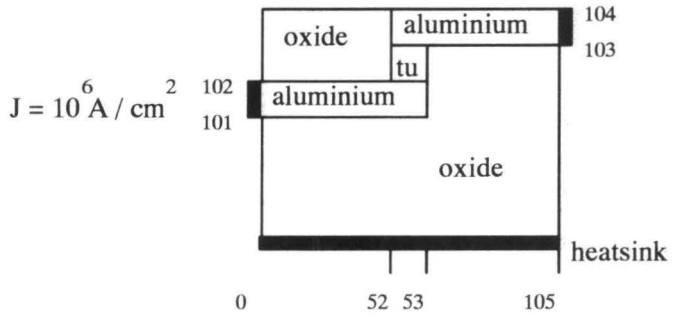

Figure 2: A Tungsten filled via-structure.

otherwise. The $\mathcal{C}$ denotes the heat capacity constant, which depends on the material. The $\mathcal{K}$ denotes the thermal conductivity of the material. For aluminium and tungsten the $\mathcal{K}$ is constant (see table 1), for oxide it depends on the local temperature:

$$\mathcal{K}(T) = 28.1\ T^{-\frac{4}{3}} [W cm^{-1} K^{-1}].$$

A current-boundary contact is connected to one side of the structure, while a heatsink

material	$\mathcal{C}$[JK^{-1}cm^{-3}]	$\mathcal{K}$[Wcm^{-1}K^{-1}]
aluminium	2.429	2.37
tungsten	2.606	1.73

Table 1: Heat capacity and thermal conductivity.

with a temperature $T = 300$ Kelvin is connected over the full bottom. The current source delivers a current of 10^6 A/cm^2. The electrical conductivity of the aluminium and tungsten are taken to be $1/\sigma_a = 2.66\mu$ ohm·cm and $1/\sigma_t = 10.0\mu$ ohm·cm respectively. Here, our main concern is the efficiency of iterative solvers and not so much the physical modeling of devices, therefore we refer to [24] for the procedures as

followed. The problem is discretized on a rectangular 74×54-grid with not constant spacing. At the first Newton-iteration we solve for the potential ψ, the second Newton-iteration we solve for the temperature T, the third iteration for the potential again. In Figure 3 we see the 10-logarithm of the Euclidean norm of the iteration vectors r_i versus consumed workunits, one Bi-CGSTAB sweep ($\sigma = 2$) takes 2 workunits. As

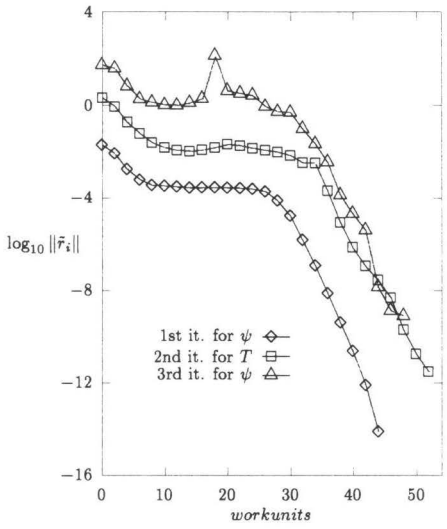

Figure 3: Convergence history of problem 1.

for **left-Bi-CGSTAB** the r_i are *scaled* residuals. Typically, the norm of the residual is stuck on a plateau for several iterations upon which suddenly the norm drops very fast. There is no real economizing in interrupting Bi-CGSTAB during such a drop. These results supply an additional argument to be not too much concerned at devising inexact Newton methods. The convergence history of Bi-CGSTAB as iterative linear solver is, in a practical sense, unpredictable. Generally, the unpredictable convergence behaviour of CG-methods seems to be in conflict with cleverly economizing strategies. Linear multigrid methods usually show a regular convergence behaviour, therefore they may fit better within such strategies (see Section 5).

4.2 Problem 2

The next example problem models a NPN-transistor. We want to solve the Poisson and continuity equations simultaneously and obtain the electrostatic potential and the concentration of holes and electrons. For a comprehensive description of these equations the reader is referred to [18]. Figure 4 shows the geometry of the device and its dimensions. The added impurities and corresponding concentrations have been indicated too. We have prescribed voltages at the emitter (E), the base (B) and the collector (C). Here is an example where we have a system of three coupled PDEs. The problem is discretized on a rectangular 26×59-grid with not constant spacing. Again we use **left-Bi-CGSTAB-v2**, but now we use the form of ILLU which is generalized for the case of discretized coupled PDEs. The convergence history is shown in Figure 5, where the same conventions hold as for Figure 3. We observe a satisfactory convergence behaviour for the subsequent Newtonsweeps.

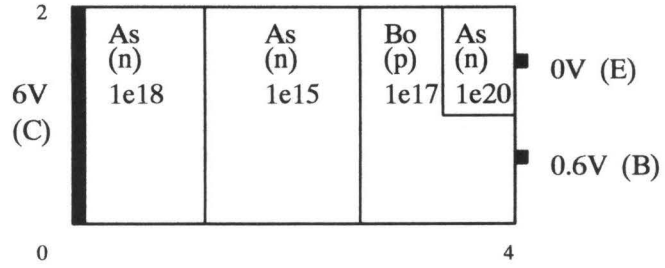

Figure 4: Geometry of the NPN-transistor of problem 2.

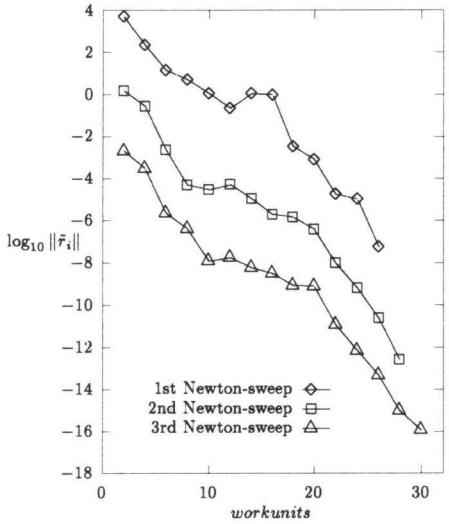

Figure 5: Convergence history of problem 2.

5 A comparison with multigrid

The rate of convergence of standard multigrid methods often deteriorates when the coefficients in the differential equation are discontinuous [1], or when dominating first-order terms are present [26]. The shortly described testproblem covers both difficulties. By means of an extension and further analysis of techniques as described in [1, 5] the blackbox multigrid solver MGD9V was developed [25]. By the choice of particular matrix-dependent gridtransfer-operators within MGD9V, the deterioration of convergence has been overcome. Intended to tackle hard problems, this solver has been equipped with ILLU as smoother. This facilitates a comparison with our version of Bi-CGSTAB where we use ILLU as preconditioner.

5.1 Problem 3

The following testproblem has been proposed by Van der Vorst [22]. It is a simplified

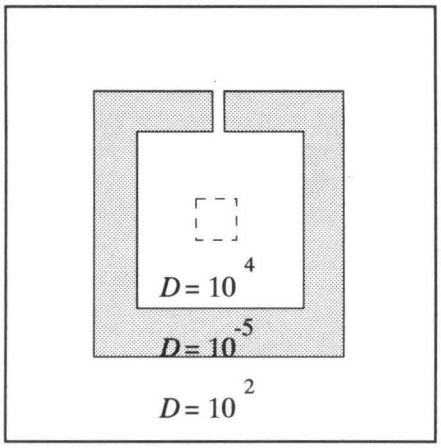

Figure 6: Geometry of Van der Vorst's aquifer-problem.

aquiferproblem, the convection-diffusion equation reads

$$-\nabla \cdot (D\nabla u) + b(x,y)\frac{\partial u}{\partial x} = f(x,y)$$
$$\Omega = (0,1) \times (0,1)$$

where the diffusion coefficient function D can be read from Figure 6, and

$$b(x,y) = 2\exp(2(x^2+y^2)).$$

We have Dirichlet boundary conditions: $u = 1$ on $\partial\Omega$ except for $y = 1$ where $u = 0$. The function $f(x,y)$ vanishes everywhere, except for the small (dashed) square in the centre where $f(x,y) = 100$. We use meshsize $h = 1/130$, leading to a system with 129^2 unknowns. Here we use the same discretization as chosen by Van der Vorst, i.e. central differences. That's why the resulting linear system is badly conditioned: at the shell where $D = 10^{-5}$ we observe that $h\|b\| > \|D\|$. For completeness we also report the results for **right-Bi-CGSTAB** which is the analogue of **left-Bi-CGSTAB-v2** but now with preconditioning from the right. In Figure 7 we see the 10-logarithm of the Euclidean norm of the iteration vectors r_i versus consumed workunits. One Bi-CGSTAB sweep ($\sigma = 2$) takes 2 workunits, one MGD9V cycle takes 1 workunit. For MGD9V and **right-Bi-CGSTAB** the r_i are residuals, as for **left-Bi-CGSTAB** the r_i are *scaled* residuals. Figure 7 suggests that from the viewpoint of efficiency, MGD9V has a clear advantage over the Bi-CGSTAB algorithms. Along with ILLU as relaxation method this is due to advanced features like matrix-dependent gridtransfers and an automatic Galerkin approximation of coarse grid matrices. Considerable research and programming effort has been put in this multigrid-program which is nevertheless only suited for a scalar equation. It is no trivial matter to generalize the advanced features in MGD9V to a set of coupled equations. Here we observe an advantage of Bi-CGSTAB over MGD9V because the generalization was rather straightforward.

There appears to be no substantial difference in overall convergence rate between **left-** and **right-Bi-CGSTAB**. Yet, for the latter variant it is less easy to devise a stopping and restart criterion which holds under the circumstances mentioned in Section 3.3.

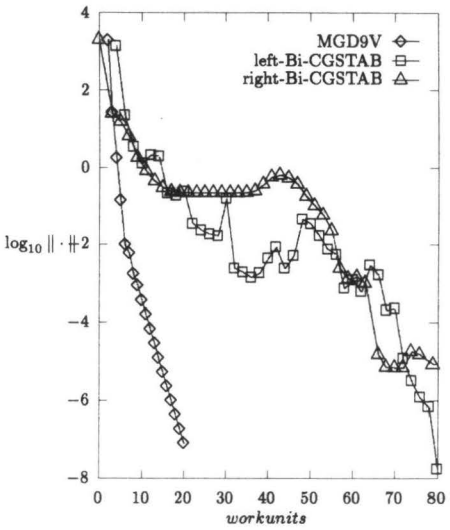

Figure 7: Convergence history of MGD9V and Bi-CGSTAB for the aquifer problem.

6 Concluding remarks

Variants of the Bi-CGSTAB algorithm of Van der Vorst, furnished with ILLU as preconditioner, prove to be robust and efficient iterative solution methods. Of course one might consider other preconditioners with higher efficiency in the sense of high floprates on vector and parallel computers. However, in the practice of semiconductor modelling for instance, robustness is paramount. The presented algorithm appears to equal Gaussian elimination in robustness, but is of course far more efficient in CPU and memory requirements. This makes the algorithm eligible for practical purposes. An existing multigrid-algorithm, MGD9V, with ILLU as smoothing procedure shows considerably faster convergence. However, this algorithm is applicable only to the scalar case and contains features which are hard to generalize to the case of coupled equations.

Acknowledgements

The author wishes to thank Prof. Dr. Henk A. Van der Vorst for his comments and for providing his subroutine that discretizes the aquifer problem of Section 5. The members of the Integrated Circuits and Electronics group (ICE) of the University of Twente, The Netherlands, are gratefully acknowledged for providing two testproblems.

References

[1] R.E. Alcouffe, A. Brandt, J.E. Dendy Jr. and J.W. Painter, *The multi-grid method for the diffusion equation with strongly discontinuous coefficients*, SIAM J.Sci.Statist.Comput. **2**(4), 430-454, 1981.

[2] O. Axelsson, *A General Incomplete Block-Matrix Factorization Method*, Linear Algebra and its Applications **74**, pp. 179-190, 1986.

[3] O. Axelsson, *A survey of preconditioned iterative methods for linear systems of algebraic equations*, BIT **25**, pp. 166-187, 1985.

[4] G. Brussino and V. Sonnad, *A comparison of direct and preconditioned iterative techniques for sparse unsymmetric systems of linear equations*, Int. J. for Num. methods in Eng., 28, pp. 801-815, 1989.

[5] J.E. Dendy Jr., *Blackbox multigrid for nonsymmetric problems*, Appl. Math. Comput. **13**, pp. 261-283, 1983.

[6] I.S. Duff, A.M. Erisman, J.K. Reid, *Direct Methods for Sparse Matrices*, Monographs on Numerical Analysis, Clarendon Press, Oxford, 1986.

[7] P. Concus, G.H. Golub and G. Meurant, *Block preconditioning for the conjugate gradient method*, SIAM J.Sci.Statist.Comput., **6** (1), pp. 220-252, 1985.

[8] Ron S. Dembo, Stanley C. Eisenstat and Trond Steihaug, *Inexact Newton Methods*, SIAM J.Numer.Anal., **19** (2), pp. 400-408, 1982.

[9] M. Driessen and H.A. Van der Vorst, *Bi-CGSTAB in semiconductor modelling*, Philips Nat. Lab. Unclassified Report 011/91, 1991.

[10] G.H. Golub, C.F. Van Loan, *Matrix Computations 2nd edition*, The Johns Hopkins University Press, Baltimore and London, 1989.

[11] P.W. Hemker, private communication.

[12] P.W. Hemker, R. Kettler, P. Wesseling and P.M. de Zeeuw, *Multigrid methods: development of fast solvers*, Applied Mathematics and Computations **13**, pp. 311-326, 1983.

[13] P.W. Hemker and P.M. de Zeeuw, *Some implementations of multigrid linear system solvers* in: D.J. Paddon and H. Holstein (Eds.) *Multigrid methods for integral and differential equations*, Inst. Math. Appl. Conf. Ser. New Ser. (Oxford Univ. Press, New York), 85-116, 1985.

[14] R. Kettler, *Analysis and comparison of relaxation schemes in robust multigrid and preconditioned conjugate gradient methods* in: W. Hackbusch and U. Trottenberg (Eds.), Lecture Notes in Math. **960** (Springer, Berlin), 502-534, 1981.

[15] R. Kettler and J.A. Meijerink, *A multigrid method and a combined multigrid-conjugate gradient method for elliptic problems with strongly discontinuous coefficients in general domains* Shell Publ. 604, KSEPL, Rijswijk, The Netherlands, 1981.

[16] 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.of Comp., **31**, pp. 148-162, 1977.

[17] E. van Schie, *TRENDY: an integrated program for IC process and device simulation*, Ph.D. Thesis, Integrated Circuits and Electronics group, University of Twente, The Netherlands, ISBN 90-9003357-2, 1990.

[18] S. Selberherr, *Analysis and simulation of semiconductors devices*, (Springer-Verlag, Berlin, New York), 1984.

[19] P. Sonneveld, *CGS: a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J.Sci.Statist.Comput., **10**, pp. 36-52, 1989.

[20] P. Sonneveld, P. Wesseling and P.M. de Zeeuw, *Multigrid and conjugate gradient methods as convergence acceleration techniques*, in: D.J. Paddon and H. Holstein (Eds.) *Multigrid methods for integral and differential equations*, Inst. Math. Appl. Conf. Ser. New Ser. (Oxford Univ. Press, New York), pp. 117-167, 1985.

[21] R.R. Underwood, *An approximate factorization procedure based on the block Cholesky decomposition and its use with the conjugate gradient method*, Report NEDO-11386, General Electric Co., Nuclear Energy Div., San Jose, California, 1976.

[22] H.A. Van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J.Sci.Statist.Comput. **13**(2), pp. 631-644, 1992.

[23] P. Wesseling, P. Sonneveld, *Numerical experiments with a multiple grid and a preconditioned Lanczos type method*, in: R. Rautmann (ed.), *Approximation methods for Navier-Stokes problems*, Proceedings, Paderborn, Germany 1979, Lecture Notes in Mathematics, Springer, Berlin-etc., 1980.

[24] P. Wolbert, *Modeling and simulation of semiconductor devices in TRENDY*, Ph.D. Thesis, Integrated Circuits and Electronics group, University of Twente, The Netherlands, ISBN 90-9004446-9, 1991.

[25] P.M. de Zeeuw, *Matrix-dependent prolongations and restrictions in a blackbox multigrid solver* Journal of Computational and Applied Mathematics, **33**, pp. 1-27, 1990.

[26] P.M. de Zeeuw and E.J. van Asselt, *The convergence rate of multi-level algorithms applied to the convection-diffusion equation* SIAM J.Sci.Statist.Comput. **6**(2), pp. 492-503, 1985.

A ILLU and left-Bi-CGSTAB implemented in C

```
#include <stdio.h>
#include <math.h>
#include "typdefs.h"
#include "define.h"

main()
{
 int n, nx, ny, moni = 1;
 REAL **jacobrhs;
 FILE *historyf;

 n  =   3;
 nx =  26;
 ny =  59;

 /*
    now allocate sufficient space to jacobrhs
    (i.e. jacobian & righthandside)
 */
 allocmat(n*nx*ny, 5*n+3+3*n+3*n+10, &jacobrhs);

 /* now store matrix and righthandside */
 filldatastruct(jacobrhs,n,nx,ny,moni);

 if ( (historyf = fopen("../historyf","w+")) == NULL )
  {
    printf("/n history file is not opened! ");
    exit(68);
  }

 /* now run left-Bi-CGSTAB-v2          */
 bi_cgstab_prec(historyf,jacobrhs,n*nx*ny,n,nx,ny,moni);

 fclose(historyf);
}

/* Fill the datastructure              */

int filldatastruct(matrixrhs, n,nx,ny,moni)
int n,nx,ny,moni;
REAL **matrixrhs;
{
  int i, j, kh, kv;

/*
   Computationally we have a rectangular grid, which
   is curvilinear in the geometrical sense.

   nx is the number of gridpoints in the x-direction,
   ny is the number of gridpoints in the y-direction,
   n  is the dimension of the coupled system of PDEs
```

```
   and the number of solutioncomponents at the same time
   (typical values for n are 1 or 3, but any positive
    integer value is permitted).

For numbering and subscripting we use i, j, kh, kv:

  i  = 1, 2, .... nx,
  j  = 1, 2, .... ny,
  kh = 1, 2, .... n,
  kv = 1, 2, .... n.

Hence, the pair (i,j) corresponds to the position of a
node in the x, y-plane.
At each node (i,j) we have n solutioncomponents subscripted
by kh = 1, 2, .... n.
At each node (i,j) we have n linear equations subscripted
by kv = 1, 2, .... n.

As for the discretization we assume to have the standard
central five point coupling:

     y

     ^
     |
     |             (i, j+1)
     |
     |   (i-1, j)  (i, j  )  (i+1, j)
     |
     |             (i, j-1)
     |
      ------------------------------>  x

The matrix and righthandside of the sparse linear system
are stored into an array as follows:
  the length of the array amounts to ny*nx*n,
  the width of this array is 5n + 3 + 3n + 3n +10.
Hence we have 11n+13 columns with length ny*nx*n.
The user needs to initialize the first 5n+3 columns, the
other columns are in use with the preconditioner and the
bi-cgstab algorithm.
The matrix has to be stored in the first 5n columns,
the righthandside in column number 5n+1,
the initial residual in column number 5n+2,
the initial solution in column number 5n+3.
The ordering of the last three vectors is as follows:
the kh-th solutioncomponent at the node (i,j) is to be
found at element-number
          ( (j-1) * nx + i-1 ) * n + kh
of the column.
The linear equations with the system correspond to the rows
          ( (j-1) * nx + i-1 ) * n + kv
of the array.
```

```
   The first 5n columns of the array contain the matrix in the
   following way (consider the five point stencil depicted above)

   node (i  , j-1)  <--> columns     1, ... ,  0+kh, ... , n
   node (i-1, j  )  <--> columns   n+1, ... ,  n+kh, ... , 2n
   node (i  , j  )  <--> columns  2n+1, ... , 2n+kh, ... , 3n
   node (i+1, j  )  <--> columns  3n+1, ... , 3n+kh, ... , 4n
   node (i  , j+1)  <--> columns  4n+1, ... , 4n+kh, ... , 5n

*/

  return;
}

/*
   Elementary computations on the level of
   matrices and vectors of dimension n
*/

int matxvec(n,ja_pt,ia,da,jb_pt,ib,db,jt_pt,it,dt)
int n,ia,da,ib,db,it,dt;
REAL **ja_pt,**jb_pt,**jt_pt;
/* vec(t) = mat(a) * vec(b)        */
{
  REAL **jia_pntr, **jib_pntr, **jit_pntr;
  REAL *a_pntr, *b_pntr, *t_pntr;
  int kv, kk;

  jia_pntr = ja_pt + ia*n;
  jib_pntr = jb_pt + ib*n;
  jit_pntr = jt_pt + it*n;

  for (kv=0; kv<n; kv++)
    {
     t_pntr = *(jit_pntr++)+dt;
     *(t_pntr) = 0.0;
     a_pntr = *(jia_pntr++)+da;
     for (kk=0; kk<n; kk++)
       {
         b_pntr = *(jib_pntr+kk)+db;
         *(t_pntr) += ( *(a_pntr++) ) * ( *(b_pntr) );
       }
    }
  return;
}

int substrmatxvec(n,ja_pt,ia,da,jb_pt,ib,db,jt_pt,it,dt)
int n,ia,da,ib,db,it,dt;
REAL **ja_pt, **jb_pt, **jt_pt;
/* vec(t) -= mat(a) * vec(b)       */
{
  REAL **jia_pntr, **jib_pntr, **jit_pntr;
  REAL *a_pntr, *b_pntr, *t_pntr;
  int kv, kk;
```

```
  jia_pntr = ja_pt + ia*n;
  jib_pntr = jb_pt + ib*n;
  jit_pntr = jt_pt + it*n;

  for (kv=0; kv<n; kv++)
    {
     t_pntr = *(jit_pntr++)+dt;
     a_pntr = *(jia_pntr++)+da;
     for (kk=0; kk<n; kk++)
       {
         b_pntr = *(jib_pntr+kk)+db;
         *(t_pntr) -= ( *(a_pntr++) ) * ( *(b_pntr) );
       }
    }
  return;
}

int copyvec(n,ja_pt,ia,da,jb_pt,ib,db)
int n,ia,da,ib,db;
REAL **ja_pt, **jb_pt;
/* vec(b) = vec(a)                  */
{
  REAL **jia_pntr, **jib_pntr;
  REAL *a_pntr, *b_pntr;
  int kv;

  jia_pntr = ja_pt + ia*n;
  jib_pntr = jb_pt + ib*n;

  for (kv=0; kv<n; kv++)
    {
     a_pntr = *(jia_pntr++)+da;
     b_pntr = *(jib_pntr++)+db;
     *(b_pntr) = *(a_pntr);
    }
  return;
}

int zerovec(n,ja_pt,ia,da)
int n,ia,da;
REAL **ja_pt;
/*  set vec(a) to zero             */
{
  REAL **jia_pntr;
  REAL *a_pntr;
  int kv;

  jia_pntr = ja_pt + ia*n;

  for (kv=0; kv<n; kv++)
    {
     a_pntr = *(jia_pntr++)+da;
     *(a_pntr) = 0.0;
```

```
    }
  return;
}

int matxmat(n,ja_pt,ia,da,jb_pt,ib,db,jt_pt,it,dt)
int n,ia,da,ib,db,it,dt;
REAL **ja_pt, **jb_pt, **jt_pt;
/* mat(t) = mat(a) * mat(b)        */
{
  REAL **jia_pntr, **jib_pntr, **jit_pntr;
  REAL *a_pntr, *wa_pntr, *b_pntr, *t_pntr;
  int wdb, kv, kh, kk;

  jia_pntr = ja_pt + ia*n;
  jib_pntr = jb_pt + ib*n;
  jit_pntr = jt_pt + it*n;

  for (kv=0; kv<n; kv++)
    {
     t_pntr = *(jit_pntr++)+dt;
     a_pntr = *(jia_pntr++)+da;
     wdb = db;
     for (kh=0; kh<n; kh++)
       {
/**      compute entry at kv-th row and kh-th column    **/
         wa_pntr = a_pntr;
         *(t_pntr) = 0.0;
         for (kk=0; kk<n; kk++)
           {
             b_pntr = *(jib_pntr+kk)+wdb;
             *(t_pntr) += ( *(wa_pntr++) ) * ( *(b_pntr) );
           }
         t_pntr ++;
         wdb ++;
       }
    }

  return;
}

int minmatxmat(n,ja_pt,ia,da,jb_pt,ib,db,jt_pt,it,dt)
int n,ia,da,ib,db,it,dt;
REAL **ja_pt, **jb_pt, **jt_pt;
/* mat(t) = - mat(a) * mat(b)       */
{
  REAL **jia_pntr, **jib_pntr, **jit_pntr;
  REAL *a_pntr, *wa_pntr, *b_pntr, *t_pntr;
  int wdb, kv, kh, kk;

  jia_pntr = ja_pt + ia*n;
  jib_pntr = jb_pt + ib*n;
  jit_pntr = jt_pt + it*n;

  for (kv=0; kv<n; kv++)
```

```
    {
     t_pntr = *(jit_pntr++)+dt;
     a_pntr = *(jia_pntr++)+da;
     wdb = db;
     for (kh=0; kh<n; kh++)
       {
/**      compute entry at kv-th row and kh-th column   **/
         wa_pntr = a_pntr;
         *(t_pntr) = 0.0;
         for (kk=0; kk<n; kk++)
           {
             b_pntr = *(jib_pntr+kk)+wdb;
             *(t_pntr) -= ( *(wa_pntr++) ) * ( *(b_pntr) );
           }
         t_pntr ++;
         wdb ++;
       }
    }
  return;
}

int substrmatxmat(n,ja_pt,ia,da,jb_pt,ib,db,jt_pt,it,dt)
int n,ia,da,ib,db,it,dt;
REAL **ja_pt, **jb_pt, **jt_pt;
/* mat(t) -= mat(a) * mat(b)       */
{
  REAL **jia_pntr, **jib_pntr, **jit_pntr;
  REAL *a_pntr, *wa_pntr, *b_pntr, *t_pntr;
  int wdb, kv, kh, kk;

  jia_pntr = ja_pt + ia*n;
  jib_pntr = jb_pt + ib*n;
  jit_pntr = jt_pt + it*n;

  for (kv=0; kv<n; kv++)
    {
     t_pntr = *(jit_pntr++)+dt;
     a_pntr = *(jia_pntr++)+da;
     wdb = db;
     for (kh=0; kh<n; kh++)
       {
/**      compute entry at kv-th row and kh-th column   **/
         wa_pntr = a_pntr;
         for (kk=0; kk<n; kk++)
           {
             b_pntr = *(jib_pntr+kk)+wdb;
             *(t_pntr) -= ( *(wa_pntr++) ) * ( *(b_pntr) );
           }
         t_pntr ++;
         wdb ++;
       }
    }
  return;
}
```

```
int allocmat(n,m,mat)
int n, m;
REAL ***mat;
/*  allocates space for a m times n matrix */
{
  int j;
  REAL *preal;

  *mat = (REAL **) calloc(n,ADRES_SIZE);
  preal = (REAL *) malloc(n*m*sizeof(REAL));
  if(preal == NILL)
   {
     printf("no space allocated for n * m mat\n");
     exit(1);
   }
  for (j=0; j<n; j++)
   {
     *(*mat+j) = preal+j*m;
   }
  return;
}

int freemat(mat)
REAL ***mat;
/* dismiss allocated space         */
{
   free(**mat);
   free( *mat);
   *mat=NILL;

  return;
}

int invertmat(n,ja_pt,ia,da)
int n,ia,da;
REAL **ja_pt;
/* invert mat(a)                   */
{
  REAL **jia_pntr, **mat;
  REAL *mat_pntr, *r1_pntr, *r2_pntr, *r3_pntr;
  REAL a11, a12, a13, a21, a22, a23, a31, a32, a33, odet;
  REAL ari, max, pivot, vp, vd;
  int i, j, diagonal, jp;

  jia_pntr =  ja_pt + ia*n;
  mat_pntr = *jia_pntr+da;

  switch (n)
  {
      case  1:
          odet = *mat_pntr;
          if( odet > -1.0e-19 && odet < 1.0e-19 )
            {
```

```
        printf("\n WARNING determinant %8.2e at i %d ",odet,ia);
        if ( odet == 0.0 )
         {
           printf("\n -Error- singular main diagonal matrix");
           exit(2);
         }
       }
     *mat_pntr = 1.0/odet;
     break;
case  2:
     r1_pntr = mat_pntr;
     a11 = *r1_pntr; a12 = *(++r1_pntr);
     r2_pntr = *(jia_pntr+1)+da;
     a21 = *r2_pntr; a22 = *(++r2_pntr);

     odet =  a11 * a22 - a21 * a12;
     if( odet > -1.0e-19 && odet < 1.0e-19 )
       {
         printf("\n WARNING determinant %8.2e at i %d ",odet,ia);
         if ( odet == 0.0 )
           {
            printf("\n -Error- singular main diagonal matrix");
            exit(2);
           }
       }
     odet = 1.0/odet;

     *(  r1_pntr) = - a12 * odet;
     *(--r1_pntr) =   a22 * odet;
     *(  r2_pntr) =   a11 * odet;
     *(--r2_pntr) = - a21 * odet;

     break;

default:
     if( n <= 0 )
      {
        printf("\n - Error - not inverting for n <= 0 %d\n", n);
        exit(2);
      }
     allocmat(n,2*n,&mat);
     for ( j=0; j<n; j++ )
       {
         /* make copy    */
         r1_pntr = *(jia_pntr+j);
         r2_pntr = *(mat     +j);
         for ( i=0; i<n;     i++)
           {
             *(r2_pntr+i) = *(r1_pntr+da+i);
           }
         /* set righthandside to zero */
         for ( i=n; i<(2*n); i++)
           {
             *(r2_pntr+i) = 0.0;
```

```
      }
    /* apply rowscaling */
    max = 0.0;
    for ( i=0; i<n;     i++)
      {
        ari = fabs( *(r2_pntr+i) );
        if ( ari > max )
         {
           max = ari;
         }
      }
    if ( max == 0.0 )
      {
       printf("\n -Error- singular main diagonal matrix");
       exit(2);
      }
    for ( i=0; i<n; i++)
      {
        *(r2_pntr+i) /= max;
      }
    *(r2_pntr+n+j) = 1.0/max;
  }

for ( diagonal=0; diagonal<n; diagonal++ )
  {
    /* pivot search */
    pivot = 0.0; jp = -1;
    for ( j=diagonal; j<n; j++ )
      {
        ari = *(*(mat+j)+diagonal);
        if ( fabs(ari) > fabs(pivot) )
         {
           pivot = ari;
           jp = j;
         }
      }
    if( pivot > -1.0e-19 && pivot < 1.0e-19 )
     {
       printf("\n WARNING pivot  %8.2e at i %d ",pivot,ia);
       if ( pivot == 0.0 )
        {
         printf("\n -Error- singular main diagonal matrix");
         exit(2);
        }
     }
    /* divide pivotrow by pivot */
    for ( i=diagonal+1; i<2*n; i++ )
      {
        *(*(mat+jp)+i) /= pivot;
      }
    /* exchange pivotrow with current diagonalrow */
    if( jp != diagonal )
     {
       for ( i=diagonal;   i<2*n; i++ )
```

```
                {
                  vp = *(*(mat+jp      )+i);
                  vd = *(*(mat+diagonal)+i);
                  *(*(mat+jp      )+i) = vd;
                  *(*(mat+diagonal)+i) = vp;
                }
            }
            /* eliminate column_entries except for diagonal-entry */
            for ( j=0; j<n; j++ )
              {
                if ( j!=diagonal )
                 {
                   ari = *(*(mat+j)+diagonal);
                   for ( i=diagonal+1; i<2*n; i++ )
                     {
                       *(*(mat+j)+i) -= ari * *(*(mat+diagonal)+i);
                     }
                 }
              }
          }

        /* copy back to jacobian */
        for ( j=0; j<n; j++ )
          {
            r1_pntr = *(jia_pntr+j);
            r2_pntr = *(mat     +j);
            for ( i=0; i<n;     i++)
              {
                *(r1_pntr+da+i) = *(r2_pntr+n+i);
              }
          }
        freemat(&mat);
        break;
  }

  return;
}

int zeromat(n,ja_pt,ia,da)
int n,ia,da;
REAL **ja_pt;
/*  set mat(a) to zero              */
{
  REAL **jia_pntr;
  REAL *a_pntr;
  int kv, kh;

  jia_pntr = ja_pt + ia*n;

  for (kv=0; kv<n; kv++)
    {
     a_pntr = *(jia_pntr+kv)+da;
     for (kh=0; kh<n; kh++)
       {
```

```
/**       entry at kv-th row and kh-th column is set to zero   **/
          *(a_pntr ++) = 0.0;
        }
    }
  return;
}

int copymat(n,ja_pt,ia,da,jb_pt,ib,db)
int n,ia,da,ib,db;
REAL **ja_pt, **jb_pt;
/* mat(b) = mat(a)                  */
{
  REAL **jia_pntr, **jib_pntr;
  REAL *a_pntr, *b_pntr;
  int kv, kh;

  jia_pntr = ja_pt + ia*n;
  jib_pntr = jb_pt + ib*n;

  for (kv=0; kv<n; kv++)
    {
     a_pntr = *(jia_pntr++)+da;
     b_pntr = *(jib_pntr++)+db;
     for (kh=0; kh<n; kh++)
       {
/**       entry at kv-th row and kh-th column at b is copied from a  **/
         *(b_pntr ++) = *(a_pntr ++);
       }
    }
  return;
}

int compresidu(rhscol,matrix,solcol,rescol,n,nx,ny)
int rhscol,solcol,rescol,n,nx,ny;
REAL **matrix;
/*
  Performs: vec(rescol) = vec(rhscol) - [matrix] * vec(solcol)
*/
{
    int i, j, k, m, nxn;
    REAL **j_pt, **jm_pt, **jp_pt;

    nxn = nx*n;

    for (j=ny-1; j>=0; j--)
      { j_pt = matrix+j*nxn;
        if ( j!=0    ) { jm_pt = matrix+(j-1)*nxn; }
        if ( j!=ny-1 ) { jp_pt = matrix+(j+1)*nxn; }
        for (i=0; i<nx; i++)
          {
            copyvec(n, j_pt,i,rhscol, j_pt,i,rescol);
            substrmatxvec(n, j_pt,i  ,2*n,
                             j_pt,i  ,solcol,
                             j_pt,i  ,rescol);
```

```
            if (j != 0 )
             {
               substrmatxvec(n, j_pt,i  ,0,
                              jm_pt,i  ,solcol,
                               j_pt,i  ,rescol);
             }
            if (j != ny-1 )
             {
               substrmatxvec(n, j_pt,i  ,4*n,
                              jp_pt,i  ,solcol,
                               j_pt,i  ,rescol);
             }
            if (i != 0 )
             {
               substrmatxvec(n, j_pt,i  ,n,
                               j_pt,i-1,solcol,
                               j_pt,i  ,rescol);
             }
            if (i != nx-1 )
             {
               substrmatxvec(n, j_pt,i  ,3*n,
                               j_pt,i+1,solcol,
                               j_pt,i  ,rescol);
             }
          }
       }
  return;
 }

/* Now follow expedients for incomplete_line_lu  */
/* (see  ksolve( ))                              */

int tridiaginv(n,nx,j_pt,beginl,begind,beginu,
               beginq1,beginq2,beginq3)
int n,nx,beginl,begind,beginu,beginq1,beginq2,beginq3;
REAL **j_pt;
/*
                                -1
        Computes Q  = tridiag( D   ) ,
                  j             j
        D  is given by its decomposition L , D , U .
         j                                 j   j   j
        Q  is stored in q1 q2 q3
         j
*/
{
  int k;

  copymat(n, j_pt,nx-1,begind,
             j_pt,nx-1,beginq2);
  minmatxmat(n, j_pt,nx-1,beginq2,
                j_pt,nx-1,beginl,
                j_pt,nx-1,beginq1);
```

```
  for (k=nx-2; k>=0; k--)
    {
      copymat(n, j_pt,k,begind,
                 j_pt,k,beginq2);
      substrmatxmat(n, j_pt,k  ,beginu,
                       j_pt,k+1,beginq1,
                       j_pt,k  ,beginq2);
      if ( k>0 )
       {
         minmatxmat(n, j_pt,k,beginq2,
                       j_pt,k,beginl,
                       j_pt,k,beginq1);
       }
      minmatxmat(n, j_pt,k  ,beginu,
                    j_pt,k+1,beginq2,
                    j_pt,k  ,beginq3);
    }
  return;
}

int tridiblockcrout(n,nx,j_pt)
int n,nx;
REAL **j_pt;
/*
   Computes the Crout-decomposition of the tridiagonal block D
   which is stored in d1, d2, d3.
   The storage of D  is overwritten by the decomposition.
                   j
*/
{
  int k, begind1, begind2, begind3, beginw1;

  begind1  = 5*n+3;
  begind2  = begind1+n;
  begind3  = begind2+n;
  beginw1  = begind3+n;

  invertmat(n,j_pt,0,begind2);
  for (k=1; k<nx; k++)
    {
      matxmat(n,       j_pt,k-1,begind2,
                       j_pt,k-1,begind3,
                       j_pt,k-1,beginw1);
      copymat(n,       j_pt,k-1,beginw1,
                       j_pt,k-1,begind3);

      substrmatxmat(n, j_pt,k  ,begind1,
                       j_pt,k-1,begind3,
                       j_pt,k  ,begind2);
      invertmat(n,     j_pt,k  ,begind2);

      matxmat(n,       j_pt,k  ,begind1,
                       j_pt,k-1,begind2,
                       j_pt,k  ,beginw1);
```

```
      copymat(n,        j_pt,k   ,beginw1,
                        j_pt,k   ,begind1);
    }

  return;
}

int djsolve(n,nx,jd_pt,pd,jc_pt,pc,jz_pt,pz)
int n,nx,pd,pc,pz;
REAL **jd_pt, **jc_pt, **jz_pt;
/**
   Solution of D z = c, ( D  is the j_th diagonal block).
                 j          j
                                                          **/
{
  int i;

/** firstly forward substitution                          **/
/** note: c is overwritten !!!                            **/

  for (i=1; i<nx; i++)
    {
      substrmatxvec(n, jd_pt, i  , pd   ,
                       jc_pt, i-1, pc   ,
                       jc_pt, i  , pc   );
    }

/** secondly solve diagonal system                        **/

  for (i=0; i<nx; i++)
    {
      matxvec(n, jd_pt, i, pd + n,
                 jc_pt, i, pc    ,
                 jz_pt, i, pz    );
    }


/** thirdly backward substitution                         **/

  for (i=nx-2; i>=0; i--)
    {
      substrmatxvec(n, jd_pt, i  , pd + 2 * n,
                       jz_pt, i+1, pz        ,
                       jz_pt, i  , pz        );
    }

  return;
}

/*
   More expedients for incomplete_line_lu( )
   Now follow operations on the level of matrices and vectors
   of dimension ny * nx * n
*/
```

```
int incompletelineludec(matrix,n,nx,ny,moni)
int n,nx,ny,moni;
REAL **matrix;
/* Performs the Incomplete Line LU decomposition.  */
{
    int begind1, begind2, begind3, beginw1, beginw2, beginw3;
    int i, j, co, m;
    REAL **j_pt, **jm_pt;

    if (moni>0) { printf("\n incompletelineludec \n"); }

    begind1  = 5*n+3;
    begind2  = begind1+n;
    begind3  = begind2+n;
    beginw1  = begind3+n;
    beginw2  = beginw1+n;
    beginw3  = beginw2+n;

    for (j=0; j<ny; j++)
      {
        j_pt = matrix+j*nx*n;

/**     first we assign D  = A                                  **/
/**                     j    jj                                 **/
        copymat(n, j_pt,0,2*n, j_pt,0,begind2);
        copymat(n, j_pt,0,3*n, j_pt,0,begind3);
        for (i=1; i<nx-1; i++)
          {
            copymat(n, j_pt,i,1*n, j_pt,i,begind1);
            copymat(n, j_pt,i,2*n, j_pt,i,begind2);
            copymat(n, j_pt,i,3*n, j_pt,i,begind3);
          }
        copymat(n, j_pt,nx-1,1*n, j_pt,nx-1,begind1);
        copymat(n, j_pt,nx-1,2*n, j_pt,nx-1,begind2);

        if (j>0)
         {
           jm_pt = matrix+(j-1)*nx*n;

/**                                    -1                      **/
/**        we compute Q    = tridiag( D   )                    **/
/**                    j-1             j-1                     **/
/**        Q    is stored in w1 w2 w3 at j-1                   **/
/**         j-1                                                **/

           tridiaginv(n,nx,jm_pt,begind1,begind2,begind3,
                                 beginw1,beginw2,beginw3);

/**        we compute A    Q   and store in w1 w2 w3 at j      **/
/**                    jj-1 j-1                                **/
/**        subsequently                                        **/
/**        we compute D = A  - ( A    Q   ) A                 **/
/**                    j   jj     jj-1 j-1   j-1j              **/
```

```
            for ( m=1; m<nx; m++)
              {
                matxmat(n, j_pt,m,      0,
                          jm_pt,m,beginw1,
                           j_pt,m,beginw1);
                substrmatxmat(n, j_pt,m  ,beginw1,
                                jm_pt,m-1,    4*n,
                                 j_pt,m  ,begind1);
              }
            for ( m=0; m<nx; m++)
              {
                matxmat(n, j_pt,m,      0,
                          jm_pt,m,beginw2,
                           j_pt,m,beginw2);
                substrmatxmat(n, j_pt,m,beginw2,
                                jm_pt,m,    4*n,
                                 j_pt,m,begind2);
              }
            for ( m=0; m<nx-1; m++)
              {
                matxmat(n, j_pt,m,      0,
                          jm_pt,m,beginw3,
                           j_pt,m,beginw3);
                substrmatxmat(n, j_pt,m  ,beginw3,
                                jm_pt,m+1,    4*n,
                                 j_pt,m  ,begind3);
              }

          } /** end of if  (j>0)   **/

/**     block Crout-decomposition of diagonal block D     **/
/**                                                   j    **/
        tridiblockcrout(n,nx,j_pt);

      }     /** end of for(j=...   **/

  return;
}

int blockforw_backw_substit(matrix,n,nx,ny,moni)
int n,nx,ny,moni;
REAL **matrix;
/** approximates solution of:  [matrix] correction = res              **/
/** subsequently:              solution = solution + correction       **/
{
  int i, j, m, beginres, beginsol, begind, beginw1, beginw2, nxn;
  REAL **j_pt, **jm_pt, **jp_pt, *row_pntr;

  nxn = nx * n;
  beginres = 5*n+1;
  beginsol = beginres+1;
  begind   = beginsol+1;
  beginw1  = begind+3*n;
  beginw2  = beginw1+n;
```

```
  if (moni>1) { printf("\n blockforw_backw_substit \n"); }

/** firstly forward substitution                                        **/
/** solution thereof is put into res                                    **/

/** the case j=0 already completed but for copying to workspace w2      **/
    j_pt = matrix;
    for(i=0; i<nx; i++)
      {
        copyvec(n, j_pt,i,beginres,
                   j_pt,i,beginw2 );
      }

/** the case j>0                                                        **/
  for (j=1; j<ny; j++)
    {
       j_pt = matrix+ j   *nxn;
      jm_pt = matrix+(j-1)*nxn;

      djsolve(n,nx,jm_pt,begind,
                   jm_pt,beginw2,
                    j_pt,beginw1);
      for (i=0; i<nx; i++)
        {
          substrmatxvec(n, j_pt,i,      0,
                           j_pt,i,beginw1,
                           j_pt,i,beginres);
          copyvec(n, j_pt,i,beginres,
                     j_pt,i,beginw2 );
        }
    }

/** secondly backward substitution                                      **/
/** solution thereof is put into w1                                     **/

  j_pt = matrix+(ny-1)*nxn;
  djsolve(n,nx,j_pt,begind,
               j_pt,beginres,
               j_pt,beginw1);
  for (m=0; m<nxn; m++)
    {
      row_pntr = *(j_pt+m);
      *(row_pntr+beginsol) += *(row_pntr+beginw1);
    }

  for (j=ny-2; j>=0; j--)
    {
       j_pt = matrix+ j   *nxn;
      jp_pt = matrix+(j+1)*nxn;
      for(i=0; i<nx; i++)
        {
          substrmatxvec(n, j_pt,i,    4*n,
                           jp_pt,i,beginw1,
```

```
                        j_pt,i,beginres);
        }
      djsolve(n,nx,j_pt,begind,
                  j_pt,beginres,
                  j_pt,beginw1);
      for (m=0; m<nxn; m++)
        {
          row_pntr = *(j_pt+m);
          *(row_pntr+beginsol) += *(row_pntr+beginw1);
        }

    }

  return;
}


/*
   Now follow expedients for bi_cgstab_prec( ).
   The operations are on the level of matrices and vectors of
   dimension ny * nx * n
*/

int xxz(matrix,clength,colx,colz)
/**                          x = x * z                                    **/
int clength, colx, colz;
REAL **matrix;
{
  register int jik;
  register REAL **jik_pntr;
  register REAL *row_pntr;

  jik_pntr = matrix;
  for (jik=0; jik<clength; jik++)
    {
      row_pntr = *(jik_pntr++);
      *(row_pntr+colx) *=  *(row_pntr+colz);
    }

  return;
}

int xxpalfz(matrix,clength,colx,alfa,colz)
/**                          x += alfa * z                                 **/
int clength, colx, colz;
REAL **matrix, alfa;
{
  register int jik;
  register REAL **jik_pntr;
  register REAL *row_pntr;

  jik_pntr = matrix;
  for (jik=0; jik<clength; jik++)
    {
```

```
      row_pntr = *(jik_pntr++);
      *(row_pntr+colx) += alfa *  *(row_pntr+colz);
    }

  return;
}

int xymalfz(matrix,clength,colx,coly,alfa,colz)
/**                          x = y - alfa * z                              **/
int clength, colx, coly, colz;
REAL **matrix, alfa;
{
  register int jik;
  register REAL **jik_pntr;
  register REAL *row_pntr;

  jik_pntr = matrix;
  for (jik=0; jik<clength; jik++)
    {
      row_pntr = *(jik_pntr++);
      *(row_pntr+colx) = *(row_pntr+coly) - alfa *  *(row_pntr+colz);
    }

  return;
}

REAL innerxy(matrix,clength,colx,coly)
/**          innerproduct (x,y)                                            **/
int clength, colx, coly;
REAL **matrix;
{
  register int jik;
  register REAL **jik_pntr;
  register REAL *row_pntr;
  register REAL d = 0.0;

  jik_pntr = matrix;
  for (jik=0; jik<clength; jik++)
    {
      row_pntr = *(jik_pntr++);
      d += *(row_pntr+colx)  *  *(row_pntr+coly);
    }

  return (d);
}

REAL innerxx(matrix,clength,colx)
/**          innerproduct (x,x)                                            **/
int clength, colx;
REAL **matrix;
{
  register int jik;
  register REAL **jik_pntr;
  register REAL *row_pntr;
```

```
  register REAL d = 0.0, s;

  jik_pntr = matrix;
  for (jik=0; jik<clength; jik++)
    {
      s = *(*(jik_pntr++)+colx); d += s * s;
    }

  return (d);
}

int minmatrixxcol(matrix,solcol,rescol,n,nx,ny)
int solcol,rescol,n,nx,ny;
REAL **matrix;
/*
   Performs: vec(rescol) = -[matrix] * vec(solcol)
*/
{
    int i, j, k, m, nxn;
    REAL **j_pt, **jm_pt, **jp_pt;

    nxn = nx*n;

    for (j=0; j<ny; j++)
      { j_pt = matrix+j*nxn;
        if ( j!=0    ) { jm_pt = matrix+(j-1)*nxn; }
        if ( j!=ny-1 ) { jp_pt = matrix+(j+1)*nxn; }
        for (i=0; i<nx; i++)
          {
            zerovec(n, j_pt,i,rescol);
            substrmatxvec(n, j_pt,i  ,2*n,
                             j_pt,i  ,solcol,
                             j_pt,i  ,rescol);
            if (j != 0 )
             {
               substrmatxvec(n, j_pt,i  ,0,
                              jm_pt,i  ,solcol,
                               j_pt,i  ,rescol);
             }
            if (j != ny-1 )
             {
               substrmatxvec(n, j_pt,i  ,4*n,
                              jp_pt,i  ,solcol,
                               j_pt,i  ,rescol);
             }
            if (i != 0 )
             {
               substrmatxvec(n, j_pt,i  ,n,
                               j_pt,i-1,solcol,
                               j_pt,i  ,rescol);
             }
            if (i != nx-1 )
             {
               substrmatxvec(n, j_pt,i  ,3*n,
```

```
                                j_pt,i+1,solcol,
                                j_pt,i   ,rescol);
            }
          }
        }
  return;
 }

int copycol(matrix,clength,cola,colb)
int clength, cola, colb;
REAL **matrix;
/* Performs: vec(colb) = vec(cola) */
{
  int jik;
  REAL **jik_pntr;
  REAL *row_pntr;

  jik_pntr = matrix;
  for (jik=0; jik<clength; jik++)
    {
      row_pntr = *(jik_pntr++);
      *(row_pntr+colb) = *(row_pntr+cola);
    }

  return;
}

int zerocol(matrix,clength,col)
int clength, col;
REAL **matrix;
/* Set vec(col) to zero            */
{
  int jik;
  REAL **jik_pntr;
  REAL *row_pntr;

  jik_pntr = matrix;
  for (jik=0; jik<clength; jik++)
    {
      row_pntr = *(jik_pntr++);
      *(row_pntr+col) = 0.0;
    }

  return;
}

int ksolve(matrix,colrhs,colsol,sweeps,start,n,nx,ny,moni)
int colrhs,colsol,sweeps,start,n,nx,ny,moni;
REAL **matrix;
/*
   Solves: matrix(K) * vec(colsol) = vec(colrhs) (see bi_cgstab_prec)
*/
{
  int beginrhs, beginres, beginsol, beginw1, sweep,
```

```
    jmax, imax, kmax;
REAL max;

beginrhs= 5*n;
beginres= beginrhs+1;
beginsol= beginres+1;
beginw1= (beginsol+1)+3*n;

if (moni>1)
 { printf("\n number of ILLU-sweeps will be %d ",sweeps); }
if ( start==0 )
 {
   if (moni>1) { printf("\n starting with zero solution! "); };
   zerocol(matrix,ny*nx*n,beginsol);
   copycol(matrix,ny*nx*n,colrhs,beginres);
 }
else
 {
   if (moni>1)
    { printf("\n starting with nonzero solution and residual");
    }
   if ( colsol != beginsol )
    {
      copycol(matrix,ny*nx*n,colsol,beginsol);
    }
 }

if ( colrhs != beginrhs )
 {
   copycol(matrix,ny*nx*n,colrhs,beginrhs);
 }

if (moni>1)
 {
   max = innerxx(matrix,ny*nx*n, beginres);
   printf("\n innerproduct residual %8.2e ",max);
   printf("\n norm         residual %8.2e ",sqrt(max));
 }

blockforw_backw_substit(matrix,n,nx,ny,moni);

for (sweep=2; sweep<=sweeps; sweep++)
  {
    compresidu(beginrhs,matrix,beginsol,beginres,n,nx,ny);
    if (moni>1)
     {
       max = innerxx(matrix,ny*nx*n, beginres);
       printf("\n innerproduct residual %8.2e ",max);
       printf("\n norm         residual %8.2e ",sqrt(max));
     }
    blockforw_backw_substit(matrix,n,nx,ny,moni);

  }
```

```
  if (moni>1)
   {
     compresidu(beginrhs,matrix,beginsol,beginres,n,nx,ny);
     max = innerxx(matrix,ny*nx*n, beginres);
     printf("\n innerproduct residual %8.2e ",max);
     printf("\n norm          residual %8.2e ",sqrt(max));
   }

/** take care that the solution is stored where it is expected          **/

  if ( colsol != beginsol )
   {
     copycol(matrix,ny*nx*n,beginsol,colsol);
   }

  return;
}

int bi_cgstab_prec(histf,matrix,n,nx,ny,moni)
int n,nx,ny,moni;
REAL **matrix;
FILE *histf;
/*************************************************************************

    Implementation of: left-Bi-CGSTAB-Prec-v2 (see this report).

    Solution of a linear system using a pre-conditioned version
    of the Bi-CGSTAB algorithm of Henk van der Vorst (1990),
    preconditioning is done by Incomplete Line LU decomposition.

    input  :  **matrix = the band matrix and righthanside
              n        = number of collective continuous equations.
              nx       = number of points in x-direction.
              ny       = number of points in y-direction.

*************************************************************************/
{
  int beginrhs, beginres, beginsol, begind, beginw1, beginw2, endillu,
       sweep, sweeps, jmax, imax, kmax, nynxn;
  int cgfree, cgsweep, cgsweeps,
      cgrhs, cgxi, cgr0, cgri, cgminvi, cgpi, cgy, cgs, cgz, cgmint,
      restart = 0, go_on = 1;

  REAL unit_roundoff = 1.0e-13;
  /* see Matrix Computations, 2nd Ed., 1989, p.62, Golub & van Loan */

  REAL wucount = 0.0;
  REAL maxsc, [illegible], scwork, stopcrit, critrestart, ints, intt, sins, inrr;
  REAL alfa = 1.0, beta, [illegible] 1 = 1.0, omei, omei_1 = 1.0;

  nynxn = ny*nx*n;
  beginrhs= 5*n;
  beginres= beginrhs+1;
  beginsol= beginres+1;
```

```
  begind  = beginsol+1;
  beginw1 = begind+3*n;
  beginw2 = beginw1+n;
  endillu = beginw2+n+n;
  cgfree  = beginw2+1;

  printf("\n iterative solver bi-cgstab-prec \n");

  incompletelineludec(matrix,n,nx,ny,moni);

  sweeps = 2;
  if (moni>1) { printf("\n Initially some ILLU-sweeps"); }
  if (moni>0)
   {
     inrr = innerxx(matrix,nynxn, beginrhs);
     printf("\n CGSTABP innerproduct residual %8.2e ",inrr);
     printf("\n CGSTABP norm         residual %8.2e ",sqrt(inrr));
   }
  ksolve(matrix,beginrhs,beginsol,sweeps,0,n,nx,ny,moni);
  wucount += 1.0;

/** define the columns where the various gridfunctions are stored **/

  cgrhs   = cgfree;
  cgxi    = cgrhs    +1;
  cgr0    = cgxi     +1;
  cgri    = cgr0     +1; /**  cgri_1    = cgri;    **/
  cgminvi = cgri     +1; /**  cgminvi_1 = cgminvi; **/
  cgy     = cgminvi  +1;
  cgz     = cgy      +1;
  cgs     = cgz      +1;
  cgmint  = cgs      +1;
  cgpi    = cgmint   +1; /**  cgpi_1    = cgpi;    **/


  compresidu(beginrhs,matrix,beginsol,cgr0,n,nx,ny);
  copycol(matrix,nynxn,beginrhs,cgrhs);
  copycol(matrix,nynxn,beginsol,cgxi);

  ksolve(matrix,cgr0,beginsol,sweeps,0,n,nx,ny,moni);
  copycol(matrix,nynxn,beginsol,cgr0);
  inrr = innerxx(matrix,nynxn, cgr0);
  wucount += 1.0;

  stopcrit = inrr * unit_roundoff * unit_roundoff;
  critrestart = stopcrit;
  if (moni>0)
   {
     printf("\n stopcriterion %8.2e \n restart criterion %8.2e ",
                stopcrit,critrestart);
   }
  if (moni>0)
   {
     printf("\n                          -1 ");
```

```
   printf("\n CGSTABP innerproduct K  ri %8.2e ",inrr);
   printf("\n                        -1 ");
   printf("\n CGSTABP norm         K  ri %8.2e ",sqrt(inrr));
 }
 fprintf(histf,"\n %17.9e %17.9e ",wucount,log10(sqrt(inrr)));

fract = 1.0;
if (moni>0)
 { printf("\n When fraction of work for gauss_elim eqs %8.2e ",fract); }
scwork = fract*nx*nx*n*n-13*n*n-8*n-2*(sweeps-1)*(5*n+1)-2*sweeps*(8*n+1);
cgsweeps=scwork/(10+10*n+2*(sweeps-1)*(5*n+1)+2*sweeps*(8*n+1));
if (moni>0)
 { printf("\n then max number Bi-CGSTAB sweeps should be %d ",cgsweeps);}
if ( cgsweeps < 40 ) { cgsweeps = 40; }

if (moni>0)
 { printf("\n max number Bi-CGSTAB sweeps will be   %d ",cgsweeps);}

  go_on = (inrr > stopcrit);
  for(cgsweep=1; ( cgsweep<=cgsweeps && go_on ); cgsweep++ )
  {
    if ( cgsweep==1 || restart )
     {
       copycol(matrix,nynxn,cgr0,cgri); /** do not forget this line! **/
       rhoi = innerxx(matrix,nynxn, cgr0);
     }
    else
     {
       rhoi = innerxy(matrix,nynxn, cgr0, cgri);
     }
    if (moni>1) { printf("\n cgstabp rhoi      %8.2e ",rhoi); }

    if ( rhoi_1 > -critrestart && rhoi_1 < critrestart &&
         cgsweep > 1 && restart < 1 )
     {
       if (moni>0) { printf("\n RESTART because of rhoi_1 "); }
       if ( cgri != beginres )
        {
          copycol(matrix,ny*nx*n,cgri,beginres);
        }
       ksolve(matrix,cgrhs,cgxi,sweeps,1,n,nx,ny,moni);
       compresidu(cgrhs,matrix,cgxi,cgr0,n,nx,ny);
       wucount += 1.0;

       ksolve(matrix,cgr0,beginsol,sweeps,0,n,nx,ny,moni);
       copycol(matrix,nynxn,beginsol,cgr0);

       stopcrit = innerxx(matrix,nynxn,cgxi);
       stopcrit *= unit_roundoff * unit_roundoff;
       inrr = innerxx(matrix,nynxn, cgr0);
       critrestart = inrr * unit_roundoff * unit_roundoff;
       if (moni>0)
        {
         printf("\n new stopcriterion     %8.2e ",stopcrit);
```

```
     printf("\n new restart criterion %8.2e ",critrestart);
    }
   if (moni>0)
    {
      printf("\n                          -1 ");
      printf("\n CGSTABP innerproduct K  ri %8.2e ",inrr);
      printf("\n                          -1 ");
      printf("\n CGSTABP norm         K  ri %8.2e ",sqrt(inrr));
    }
   alfa = 1.0; rhoi_1 = 1.0; omei_1 = 1.0; restart = 1;
 }
else
 {
   if( omei_1 == 0.0 )
     { beta = 0.0; }
   else
     { beta = (rhoi/rhoi_1) * (alfa/omei_1); }
   if (moni>1) { printf("\n cgstabp beta      %8.2e ",beta); }

   if ( cgsweep==1 || restart )
    {
      copycol(matrix,nynxn,cgr0,cgpi);
    }
   else
    {
      xymalfz(matrix,nynxn, cgz , cgpi, -omei_1, cgminvi);
      xymalfz(matrix,nynxn, cgpi, cgri,   -beta, cgz    );
    }
   restart = 0;

   minmatrixxcol(matrix, cgpi, cgy, n,nx,ny);
   if (moni>1) { printf("\n Solve -vi from K(-vi) = y = -Api"); }
   ksolve(matrix,cgy,cgminvi,sweeps,0,n,nx,ny,moni);

   alfa =  -rhoi/innerxy(matrix,nynxn, cgr0, cgminvi);
   if (moni>1) { printf("\n cgstabp alfa      %8.2e ",alfa); }

   xymalfz(matrix,nynxn, cgs, cgri, -alfa, cgminvi);

   minmatrixxcol(matrix, cgs, cgz, n,nx,ny);
   if (moni>1) { printf("\n Solve -t from K(-t) = z = -As"); }
   ksolve(matrix,cgz,cgmint,sweeps,0,n,nx,ny,moni);

   sins =  innerxx(matrix,nynxn, cgs);
   if (moni>1) { printf("\n cgstabp s inner s   %8.2e ",sins); }
   ints = -innerxy(matrix,nynxn, cgmint, cgs);
   if (moni>1) { printf("\n cgstabp s inner t   %8.2e ",ints); }
   intt =  innerxx(matrix,nynxn, cgmint);
   if (moni>1) { printf("\n cgstabp t inner t   %8.2e ",intt); }
   omei = ints/intt;

   xxpalfz(matrix,nynxn, cgxi, alfa, cgpi);
   if( sins > stopcrit )
    { xxpalfz(matrix,nynxn, cgxi, omei, cgs); }
```

```
        else
         { omei = 0.0; }  /* i.e.   lucky breakdown      */

        if (moni>1)
         {
          printf("\n cgstabp omei        %8.2e ",omei);
          compresidu(cgrhs,matrix,cgxi,beginres,n,nx,ny);
          inrr = innerxx(matrix,nynxn, beginres);
          printf("\n CGSTABP innerpr real residual %8.2e ",inrr);
         }

        xymalfz(matrix,nynxn, cgri, cgs, -omei, cgmint);

        inrr = innerxx(matrix,nynxn, cgri);
        if (moni>0)
         {
          printf("\n                        -1 ");
          printf("\n CGSTABP innerproduct K  ri %8.2e ",inrr);
          printf("\n                        -1 ");
          printf("\n CGSTABP norm         K  ri %8.2e ",sqrt(inrr));
         }

        go_on = (inrr > stopcrit);
        if ( ( ! go_on ) || (cgsweep % 5 == 0) )
         {
           stopcrit = innerxx(matrix,nynxn,cgxi);
           stopcrit *= unit_roundoff * unit_roundoff;
           if (moni>0)
            { printf("\n new stopcriterion     %8.2e ",stopcrit); }
           go_on = (inrr > stopcrit);
         }
        omei_1 = omei;
        rhoi_1 = rhoi;
        if (moni>0)
         { printf("\n End of Bi-CGSTAB-prec sweep %d ",cgsweep); };
        wucount += 2.0;
        fprintf(histf,"\n %17.9e %17.9e ",wucount,log10(sqrt(inrr)));
      }
    }

  printf("\n Res_norm %8.2e in %d Bi_CGstab_prec sw",
         sqrt(inrr),(cgsweep-1));

  return;
}
```