# ITSOL : an evolving routine for the iterative solution of symmetric positive definite systems

Y. Notay [1]

V. Gheur, Z. Ould Amar, S. Petrova, P. Saint-Georges

Service de Métrologie Nucléaire
Université Libre de Bruxelles (C.P. 165)
50, Av. F.D. Roosevelt, B-1050 Brussels, Belgium.
email : ynotay@ulb.ac.be

Report GANMN 95-02

June-October 1995
Revised November 1996

[1]Supported by the "Fonds National de la Recherche Scientifique", Chercheur qualifié.

**Abstract**

This report presents the ITSOL package for the iterative solution of symmetric positive definite systems by PCG with incomplete factorizations as preconditioner.

ITSOL is a tool devised to quickly implement and test standard or new preconditioning techniques, without having to rewrite a specific program in each case.

Therefore, it does not provide the best implementation scheme for each of the proposed technique. For instance, ITSOL uses indirect addressing even in cases where direct addressing is possible and would probably be more efficient.

By making ITSOL available to the community, we give everybody the opportunity to test the methods proposed in ITSOL for their own applications. We hope also that many users will enjoy develop their own methods in ITSOL and appreciate the facilities provided in the package.

Indeed, in our mind, ITSOL is not only a package to be used as it stands, but also an evolving tool for the development of preconditioning techniques. In this respect, we will be happy to hear from any method newly implemented in ITSOL, even if their developers do not want to make it available to the community.

ITSOL is available by

- anonymous ftp to :

  mnserver.ulb.ac.be (164.15.13.22).

  (cd /pub/itsol)

- www :

  http: //homepages.ulb.ac.be/ ynotay

The file *README* contains release and copyright information.
The file *userguide.ps* contains this report.
The file *itsol.f* contains the main routines of the package.
The file *sparkit.f* contains the routines from SPARSKIT that are used by the package.

# 1 General considerations

The ITSOL routine allows to solve large sparse systems of linear equations of the form

$$Ax = f$$

using the Preconditioned Conjugate Gradient (PCG) method with incomplete factorizations as preconditioner. The routine is progressively updated with new solution procedures, preconditioners and reordering techniques.

The routine is called as follows:

**call itsol(n,a,ja,ia,d,f,S,iparam,userout)**

- **n** is the order of the matrix
- The vectors **a**, **ja** and **ia** contain information for the strictly upper triangular part of the matrix $A$ stored in Compressed Sparse Row (CSR) format [9]. The vector **a** consists of the nonzero entries of $A$, ordered row per row; **ja** consists of the columns indices corresponding to each element of **a**; and **ia** contains indices indicating position of the first entry of each row in **a**. CSR format is explained in detail in Section 6.
- The vector **d** contains the diagonal of $A$.
- The vector **f** contains the right-hand side; on output, it contains the solution.
- **S** is a working vector in ITSOL. Each time when ITSOL needs a vector, it reserves a part of **S** for its entries and respectively, frees place when it does not use this vector anymore. Some more details are given in Section 6.
- The vector **iparam** contains numerous parameters which control the solution process. These parameters will appear further in the sections and will be finally listed.
- **userout** is a routine provided by the user. According to the value of certain parameters, ITSOL call **userout** to obtain some particular information like the list of initial nodes in some reordering algorithms. This will be explained further in the sections, and we give also at the end of this report a summary of the situations which entail a call to **userout**.
- Besides, in order to report timing results, ITSOL makes the following call :

  **call cputm(etm)** ,

  where **etm** is a `real*8` variable which the routine **cputm** should set equal to the user CPU time (or the real time, according user's preference) elapsed since the begin of the execution of the program (or any other arbitrary fixed reference).

  We do not provide this routine with ITSOL because such routines are usually not portable.

Note that ITSOL may also be called when $A$ is only positive semidefinite, provided that the system is compatible i.e. that $f \in \mathcal{R}(A)$. The PCG algorithm is then directly applied to the singular system; according the results in [5] (see also the references therein), this causes no trouble except possible instabilities which are removed by projecting the search direction vector at each step onto $\mathcal{R}(A)^{\perp}$. When the parameter **iproj** is set equal to 1, ITSOL projects all search direction vectors onto $e^{\perp}$, where $e = (11\dots 1)^t$, allowing to prevent instabilities when $\mathcal{N}(A) = \text{span}\{e\}$. In any other case, **iproj** should be set equal to zero.

The preconditioner computed is of the form

$$B = U^t P^{-1} U$$

with $P = diag(U)$. The Jacobi preconditioning is included for comparison purpose as the special case for which

$$U^T = U = P = diag(A)$$

If the parameter **nvecp** is positive, some correction is added to the action of $B$, i.e. instead of $B$ the preconditioner used is the matrix $\hat{B}$ such that

$$\hat{B}^{-1} = B^{-1} + V(V^t AV)^{-1} V^t$$

where $V$ is an **n** $\times$ **nvecp** matrix. ITSOL makes the following call to **userout**

**call userout ('v',n ,a,ja,ia,nvecp,V)**

to ask the user to provide this matrix.

Before entering the incomplete factorization process, the user may optionally ask a reordering of the system. This will be described in Section 2.

The incomplete factorization is carried out in three steps. First, ITSOL determines the matrix to factorize, which is not necessarily the system matrix itself. Next, a symbolic factorization is performed to determine the structure of the triangular factors, i.e. fix the set of locations in which fill-in will be permitted. Finally, the numerical factorization is performed according to either the IC, the (unperturbed) MIC, the RIC [1], the DMIC [3] or the DRIC [7] algorithm. This will be further described in Section 3.

ITSOL is much embedded to the SPARSKIT toolkit by Y. Saad [9]. Most routines for general sparse matrix computation have been retrieved from that module. Some of them have been slightly modified to better match the needs of ITSOL, and several new routines, which participate from the same philosophy, have been added in a "contributed module".

These new routines as well as the routines used from SPARSKIT have been collected in the file *sparkit.f*. To prevent conflicts, we have given a new name to the modified routines. The use and the distribution of the routines in the file *sparkit.f* are subject to the copyright rules from SPARSKIT. The copyright notice have been reproduced on the top of the file *sparkit.f*.

The other routines, that is the ones which are specific to the purpose of ITSOL, have been collected in the file *itsol.f*. The file *README* contains our own copyright notice for the routines in that file.

## 2 Ordering techniques

To reorder the nodes of the system, ITSOL needs only to know the permutation to be applied. Then all the modifications of the vectors are done automatically by ITSOL.

To fix the permutation, the parameter **iord** allows to choose one of the following possibilities:

| iord | Corresponding permutation |
|---|---|
| 0 | No reordering |
| 1 | The permutation is entirely specified by the user through the call:<br>**call userout('p',n,a,ja,ia,riord,levels,nlev)** ;<br>ITSOL provides $A$ in **n**, **a**, **ja** and **ia** and waits for<br>the permutation to apply in the integer vector **riord**;<br>a level structure may optionally be specified by returning<br>**nlev** positive; then, **levels** is an integer array of indexes<br>indicating the beginning of each level in **riord**. |
| 10 | RRB ordering (see below); **lastnode** specifies the last level,<br>**nstep** the number of levels and **nx** the number of nodes per row;<br>works only if the initial ordering is lexicographic. |
| $10 <$ **iord** $< 100$ | Reverse Cuthill-McKee (RCM) (or reverse breadth-first-search) orderings;<br>**iord** is then understood as a two digit number;<br>the first digit (**iord**/10) determines the choice of the starting level;<br>the second digit (mod(**iord**,10)) determines the tie breaking strategy; |
| **iord**/10 | |
| 1 | initial node(s) chosen by **userout**; calling:<br>**call userout('l',n,a,ja,ia,riord,nfirst)** ;<br>on return **nfirst** is the number of initial nodes whose<br>list is given in the integer array **riord**; |
| 2 | initial node = **lastnode**; |
| 3 | initial node randomly chosen among the nodes<br>connected most strongly to their neighbors,<br>i.e. those corresponding to the rows with maximal<br>sum of the modulus of the offdiagonal entries; |
| 4 | initial node = **n**; |
| mod(**iord**,10) | |
| 1 | no specific tie breaking (SPARSKIT original implementation of BFS); |
| 2 | priority to the node having the smallest proportion of nonnumbered<br>neighbors (useful for incomplete factorization preconditioners [6]); |
| 3 | random tie breaking; |
| 4 | order neighbors according initial ordering; |
| 5 | order neighbors according the reverse initial ordering. |

"RRB" means the Recursive Red-Black ordering as described in [8], assuming a regular **nx** $\times$ **n/nx** rectangular grid. **nstep** is then the number of steps, that is the total number of levels minus one; if **nstep**=0, this number of levels is evaluated by ITSOL in such a way that the number of nodes in the last level is approximately equal to the square root of **n**; if **nstep**$<0$, full RRB reduction is performed, i.e. successive red-black partitionings are defined until there is only one node left in the black set. **lastnode**, whenever positive, is the initial number of a node that has to be left anyway in the last level; whenever nonpositive, ITSOL uses the algorithm in [8] with $\hat{\imath} = \hat{\jmath} = 0$. To minimize fill-in in case **levfac**$=-2$ (i.e. in case using the method in [8]), the nodes in the last level are permuted according a nested dissection algorithm [4].

# 3 Incomplete factorization

**Determination of the matrix to factorize** The matrix $A_0$ to factorize is chosen by the parameter **matfac**. We have:

| **matfac** | Matrix to factorize |
|---|---|
| 0 | It is the matrix $A$ from the system which is factorized, i.e. $A_0 = A$. |
| 1 | The matrix $A_0$ to factorize is provided by the calling<br><br>**call userout('S',n,a,ja,ia,d,u,ju,iu,p,nzum,nzu)**<br><br>where ITSOL provides the matrix $A$ with **n**, **a**, **ja**, **ia**, **d** and waits for the matrix $A_0$ in **u**, **ju**, **iu**, **p**. The maximal length of **u** and **ju** is given by **nzum**. The real length is to be returned in **nzu**. |
| $\geq 2$ | The matrix $A_0$ is obtained from $A$ using Stieltjes approximation of $A$ with diagonal compensation, i.e. $$offdiag(A_0) = \min(offdiag(A), 0)$$ and $$A_0 e = Ae\ .$$ |

**Symbolic factorization** The fill-in strategy is chosen as specified by **levfac** in accordance with the following table

| **levfac** | fill-in strategy |
|---|---|
| -2 | fill-in 'by levels' |
| -1 | level $0^*$ factorization (generalized SSOR version) |
| 0 | level 0 factorization (without fill-in) |
| 1 | level 1 factorization |
| 2 | level 2 factorization |
| $\vdots$ | $\vdots$ |

If the level of incomplete factorization is 0 or $0^{\star}$, the structure of the upper triangular factor $U$ is identical to that of the upper triangular part of $A_0$.

In other cases, a symbolic factorization is done to foresee in the structure **u**, **ju**, **iu** the nonzero entries which will appear in the preconditioner. For this, the upper triangular part is converted in LNK format (see Section 6) and the algorithm of symbolic factorization is applied. After that the matrix is reconverted back in CSR format .

The algorithm implemented to control fill-in when **levfac** $> 0$ is the one given in [6]. On the other hand, fill-in by levels means that, if a level structure has been specified during the reordering step, fill-in will be allowed between any two nodes not belonging to a same level as well as inside the last level. This strategy is efficient whenever coupled with an RRB ordering and a MIC factorization, see [8].

**Numerical factorization** The parameter **ifact** gives the type of the chosen preconditioner defined as follows:

| **ifact** | Chosen preconditioner |
|---|---|
| 0 | Jacobi preconditioner |
| 1 | Incomplete factorization of type IC |
| 2 | Incomplete factorization of type MIC |
| 3 | Incomplete factorization of type RIC |
| 4 | Incomplete factorization of type DMIC |
| 5 | Incomplete factorization of type DRIC |

**Note:** The symbolic factorization is skipped when the Jacobi preconditioner is selected.

During the factorization, ITSOL checks for nonpositive pivots. Whenever **ising**=1, the last pivot is allowed to be zero and, in the latter event, exchanged for the last diagonal entry of $A$. This allows to use MIC factorizations when $A$ is semidefinite, see [5]. In any other case (**ifact**$\neq$2 or $A$ positive definite), one should set **ising** equal to 0.

The RIC, DMIC and DRIC factorizations require an input parameter which is evaluated in function of **ibound**. The used rule is as follows : for the three methods, it can be proved that if the factorized matrix $A_0$ is a (weakly) diagonally dominant Stieltjes matrix, then the largest eigenvalue of $B^{-1}A_0$ is bounded by a given function of the input parameter [7]. Conversely, the parameter may be chosen in function of a prescribed "target" upper bound. This is the philosophy followed by ITSOL, where the parameter is evaluated so that the upper bound is equal to $\mu$ given by

$$\mu = \begin{cases} \mathbf{n}^{\frac{1}{\mathbf{ibound}}} & \text{if } \mathbf{ibound} < 0 \\ \mathbf{ibound} & \text{if } \mathbf{ibound} > 0 \\ \mathbf{n}^{\frac{1}{2}} & \text{if } \mathbf{ibound} = 0\ . \end{cases}$$

For completeness, it means that $\alpha = \mu^{-1}$ in the DMIC and the DRIC algorithms, and $w = 1 - 2\mu^{-1}$ in the RIC algorithm.

When solving discrete PDE's, a standard choice is obtained by letting $\mu = \mathbf{n}^{1/d}$ (i.e. **ibound** $= -d$), where $d$ is the dimension of the space in which the PDE is solved.

# 4 Implementation of PCG algorithm in ITSOL

The used routine is a modified version of the templates routine [2]. It implements the standard version of the PCG algorithm with external calls for the matrix vector multiplication, the preconditioning step, and the measure of the error.

Zero initial approximation is used.

Presently, two measures of the error have been implemented: $\varepsilon_k = \sqrt{\frac{(r_k, B^{-1}r_k)}{(b, B^{-1}b)}}$ where $r_k$ is the residual (default) and $\varepsilon_k = \frac{\|r_k\|}{\|b\|}$ (**itest** = 2). Since $(r_k, B^{-1}r_k)$ and $(b, B^{-1}b)$ are anyway computed by the CG algorithm, the first $\varepsilon_k$ requires no additional inner product.

The stopping test is

$$\varepsilon_k \leq 10^{-\mathbf{itol}} .$$

# 5 Informations provided by ITSOL

**Print the ordering** When one chooses reordering the nodes, i.e. **iord**$\neq 0$ and when **ioutord**$>0$, ITSOL sends a print of the new ordering to the unit **ioutord**. The result is readable only if the initial grid is a regular rectangular grid lexicographically ordered with **nx** nodes per line.

**Plot the structure of triangular factors** When one utilizes an incomplete factorization, with or without symbolic factorization, one can ask for a plot of the structure of the upper triangular factor. It suffices to make **ioutnoz**$\neq 0$ and ITSOL will generate a Postscript file in unit **ioutnoz**.

**More information about the solution process** In addition, ITSOL provides to the display or in a file (according to the value of **iout**) numerous informations about the different algorithms used. (Note that the absence of some message may be an information : for example, when **iord**=0, the reordering step is skipped and no message is produced with that respect).

Each time the relative error is reduced by a factor ten, ITSOL reports on the number of iterations and on the number of floating point operations (flops) per unknown (+ and $*$) which were necessary to reach this stage. Flops are not reported for the preprocessing step because to evaluate this would imply too much overhead.

ITSOL also reports on memory usage. More precisely, ITSOL reports at several stages how much it has used from the working vector **S**, including the portion of it which was already occupied at the time ITSOL was called (according the value of **minuse**, see below). This report of memory usage does however *not* take into account space needed for user vectors **a**, **ja**, **ia**, **d**, **f**.

**Roots of the polynomial generated by the conjugate gradient algorithm** The **nroot** largest and smallest roots of the polynomial generated by the conjugate gradient algorithm are computed and printed on unit **iout**. These roots are the extreme eigenvalues of the companion Lanczos process, i.e. are approximation of the extreme eigenvalues of the preconditioned system, see e.g. [10].

**Output parameters** On output, **iparam(7)** (=**itmax** on input) is set equal to the number of iterations actually performed, with a negative sign if the prescribed accuracy was not reached. On the other hand, **iparam(19)** is set equal to the number of flops executed during the solution phase.

# 6 Memory management

**Storage format CSR** The methods developed in ITSOL can be applied to large sparse matrices. That is why we use format which allows to store only the nonzero entries of the triangular factors. The *Compressed Sparse Row* (CSR) format turns out to be rather good for this reason.

Before describing CSR format, let recall that $A$ is symmetric. Hence only the entries of strictly upper triangular part of $A$ are store in CSR format in the vectors **a**, **ja**, **ia**. Its diagonal entries are stored in vector **d** (they are all nonzero because $A$ is positive definite).

To store a matrix in CRS format the following vectors are necessary ($n$ is the order of the matrix and $nz$ be the number of nonzero entries):

- $a(nz)$, which contains the $nz$ nonzero entries ordered row per row, i.e. $a$ contains firstly the nonzero entries from the first row, then from the second one, etc.
- $ja(nz)$, contains the column numbers of the corresponding entries of $a(nz)$, i.e. $ja(k)$ is the column number of element stored in $a(k)$.
- $ia(n+1)$, such that $ia(i)$ points to the beginning of row $i$ in array $a$. Therefore, the elements from row $i$ are $a(k)$ for $k$ between $ia(i)$ and $ia(i+1)-1$ (i.e. $ia(i) = ia(i+1)$ means that there are no elements in row $i$). We have $nz = ia(n+1) - 1$.

**Note:** The column indices have not to be in increasing order: inside each row, the order of the elements can be arbitrary.

Note that storing of upper triangular part of a matrix in CSR format is equivalent to store the lower triangular part in CSC (*Compressed Sparse Column*) format. Some routine from SPARSKIT allows to transpose the matrix, i.e. to convert it from CSR format to CSC format.

**Storage format LNK** LNK is abbreviation of *Linked List* format. It is much more memory-consuming than CSR format but allows to easily add and remove in the structure because the nonzero entries need not anymore be grouped by rows.

It is used during the symbolic factorization.

The LNK format needs the following vectors:

- $a$, contains $nz$ nonzero entries.
- $ja$, such that $ja(k)$ is the column index of the element stored as $a(k)$.
- $ia(n+1)$, such that $ia(i)$ points to the first element of row $i$ in $a$. $ia(i) = 0$ if there are no elements in the $i$-th row.

- $link(k)$: if the index of the row of element stored in $a(k)$ is $i$, $link(k)$ points to the next element of row $i$. $link(k) = 0$ if we have reached the end of the row.

We can add an element $a_{i_0,j_0}$ to the end of the matrix as follows

$$\left.\begin{array}{l} link(k) = nz + 1 \\ link(nz+1) = 0 \\ ja(nz+1) = j_0 \\ a(nz+1) = a_{i_0,j_0} \\ nz = nz + 1 \end{array}\right\} \begin{array}{l} \text{when the line } i_0 \text{ already contains at least} \\ \text{one element and if } k \text{ is the last element} \\ \text{of the row } i_0 \ (link(k) = 0) \end{array}$$

and

$$\left.\begin{array}{l} ia(i_0) = nz + 1 \\ link(nz+1) = 0 \\ ja(nz+1) = j_0 \\ a(nz+1) = a_{i_0,j_0} \\ nz = nz + 1 \end{array}\right\} \text{when } a_{i_0,j_0} \text{ is the first element of the row } i_0$$

We can remove an element $a_{i_0,j_0}$ located by element $k$ of vectors $a, ja$ and $link$ as follows

$$\begin{array}{r} link(k_1) = link(k) \text{ if } a_{i_0,j_0} \text{ is not the first element of row } i_0 \\ (k_1 \text{ locates the position of the previous element from the same} \\ \text{row and we have therefore had } link(k_1) = k \text{ before the modification)} \end{array}$$

and

$$ia(i_0) = link(k) \text{ when } a_{i_0,j_0} \text{ is the first element of row } i_0$$

**Note:** The element disappears from the structure but it continues to occupy a place in the memory. Hence, the length of the vectors $a$ and $ja$ can be larger than $nz$.

**Dynamic allocation of the memory** According to the order of the matrix to solve and the incomplete factorization chosen , the memory space necessary to ITSOL can vary in very large proportions. Hence, reserving working space in module ITSOL would be not practical.

That is why the user is asked to provide to ITSOL the real working vector **S** of length **memory**, and to inform ITSOL about the number of integer words required to store one real (parameter **ninteg**). ITSOL then reserves in **S** all the working vectors it needs during the solution process. Besides **memory** and **ninteg**, the user has to set **minuse** equal to the number of entries already occupied in **S** and which should not be overwritten by ITSOL (thus set **minuse**=0 if **S** is not used in other parts of the program).

If the size of **S** is not sufficient, an error message is issued and the user has to either increase the size of **S** or to choose a less memory consuming method. With the error message, a code number is printed which can help to identify at which stage the error occured.

For factorizations without fill **S** should have at least the same size as the total size of arrays **a**, **ja**, **ia**, **d** (to store the preconditioner) plus 3 **n** entries (working vector for the CG process) plus **n/ninteg** entries if reordering in asked (to store the permutation array).

More memory is needed if fill-in is allowed, but less if the factorized matrix is sparser than the system matrix.

# 7 External subroutines in ITSOL

We have already mentioned the contribution from SPARSKIT as well as the use of the "templates" routine CG.

Besides, ITSOL makes some calls to BLAS and LAPACK routines. Most computers have their own, optimized version of the BLAS while the full installation of the LAPACK library is a common service. We advise the use of these preinstalled versions. However, for the sake of completeness, we distribute with the package FORTRAN codes for the needed routines, as they were retrieved from Netlib.

# 8 Summary of different arguments and parameters to implement ITSOL

The arguments needed for ITSOL are the following:

- **n** , the order of the matrix
- **a**, **ja**, **ia** , vectors containing the upper triangular part of $A$ written in CSR format
- **d** , vector containing the diagonal of $A$
- **f** , vector containing the right-hand side of the system on input and the solution on output
- **S** , working vector for ITSOL
- **iparam**, vector of 25 entries containing the parameters to control the execution of ITSOL
- **userout**, a routine provided by the user which ITSOL can call (according to the parameters in **iparam**)

The vector **iparam** contains the following parameters.

*Input parameters*

- **iparam(1)=minuse**: number of entries already occupied in **S** at the time ITSOL is called (Section 6).
- **iparam(2)=memory**: length of the vector **S** (Section 6).
- **iparam(3)=ninteg**: quotient between the memory used for real double precision data and that for integers (Section 6).
- **iparam(4)=iout**: logical unit for printing current messages from the execution of ITSOL and messages for errors (Section 5).
- **iparam(5)=macheps** indicates the machine accuracy: real numbers less than $10^{-\textbf{macheps}}$ are assumed to be zero.

- **iparam(6)=ireset** (not yet implemented): should allow to ask ITSOL to reset the arguments into their initial state before calling to ITSOL.
- **iparam(7)=itmax**: maximal number of iterations for the Preconditioned Conjugate Gradient (PCG) method.
- **iparam(8)=itol**: accuracy in stopping criterion for PCG method is $\epsilon = 10^{-\mathbf{itol}}$ (Section 4).
- **iparam(9)=itest**: determines the norm used in PCG method to control error (Section 4).
- **iparam(10)=iord**: determines the ordering chosen (Section 2).
- **iparam(11)=matfac**: determines the matrix to factorize (Section 3).
- **iparam(12)=levfac**: determines the level of the desired incomplete factorization (Section 3).
- **iparam(13)=ifact**: determines the factorization method (Section 3).
- **iparam(14)=ibound**: determines the value of the parameter used for RIC, DMIC et DRIC (Section 3).
- **iparam(15)=nroot**: number of extremal roots of the polynomial used by PCG that are to be computed and displayed (Section 5).
- **iparam(16)=nvecp**: if positive, implies that the preconditioner will be corrected by a projection on certain vectors (Section 1).
- **iparam(17)=iproj**: parameter to be set equal to 1 if one wants to project all search direction vectors onto $e^{\perp}$ (useful whenever $\mathcal{N}(A) = \text{span}\{e\}$) and equal to zero otherwise (Section 1).
- **iparam(18)=ising**: parameter to be set equal to 1 if one wants to allow the preconditioner to be singular (useful whenever $A$ is semidefinite and **ifact**=2) and equal to zero otherwise (Section 3).
- **iparam(19)**: not used.
- **iparam(20)=lastnode**: initial node for some of the reorderings (Section 2).
- **iparam(21)=ioutnoz**: unit to print the Postscript code for the plot of the structure of the upper triangular factor of the preconditioner (Section 5).
- **iparam(22)=nx**: allows to specify the number of nodes in direction $x$ in case of rectangular grid. The parameter **nx** will simply fix the number of columns in the file in which the permutation is printed (Section 5). This value is also needed in case of RRB ordering (section 2).
- **iparam(23)=ioutord**: logical unit to print the permutation array (Section 5).
- **iparam(24)=nstep**: number of steps for the reordering RRB (section 2).

- **iparam(25)**: not used.

*Output parameters*

- **iparam(7)**: number of iterations actually performed, with a negative sign if the desired accuracy was not reached.
- **iparam(19)**: number of flops during the solution phase.
- **other**: not changed on output.

Finally we check the format to call routine **userout**. The first argument is a character*1, which indicates what information is expected by ITSOL.

- **call userout('p',n,a,ja,ia,riord,levels,nlev)**
  fix whole permutation of the system (if **iord** = 1, see Section 2).
- **call userout('l',n,a,ja,ia,riord,nfirst)**
  fix the initial node(s) for reordering (if **iord** = 11,12,13,14 or 15 , see Section 2).
- **call userout('S',n,nza,ja,ia,d,nzu,ju,iu,p,nzumax,nzu)**
  fix the matrix $U$ to factorize coming from the matrix $A$ of the system (if **matfac** = 1, see Section 3).
- **call userout ('v',n,a,ja,ia,nvecp,V)**
  fix the matrix $V$ according to which some correction is added to the preconditioner (if **nvecp** $> 0$, see Section 1).

# References


[1] O. Axelsson and G. Lindskog, *On the eigenvalue distribution of a class of preconditioning methods*, Numer. Math., 48 (1986), pp. 479–498.

[2] Barrett, Berry, Chan, Demmel, Donato, Dongarra, Eijkhout, Pozo, Romine, and van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Publications, Philadelphia, 1993.

[3] R. Beauwens, *Modified incomplete factorization strategies*, in Preconditioned Conjugate Gradient Methods, O. Axelsson and L. Kolotilina, eds., Lectures Notes in Mathematics No. 1457, Springer-Verlag, 1990, pp. 1–16.

[4] A. George and J. L. Liu, *Computer solution of large sparse positive definite systems*, Prentice-Hall, Englewood Cliffs, 1981.

[5] Y. Notay, *Solving positive (semi)definite linear systems by preconditioned iterative methods*, in Preconditioned Conjugate Gradient Methods, O. Axelsson and L. Kolotilina, eds., Lectures Notes in Mathematics No. 1457, Berlin Heidelberg New York, 1990, Springer-Verlag, pp. 105–125.

[6] Y. Notay, *Ordering methods for approximate factorization preconditioning*, Tech. Rep. IT/IF/14-11, Université Libre de Bruxelles, 1993.

[7] Y. Notay, *DRIC : a dynamic version of the RIC method*, Journ. Num. Lin. Alg. with Appl., 1 (1994), pp. 511–532.

[8] Y. Notay and Z. Ould Amar, *A nearly optimal preconditioning based on recursive red-black orderings*. submitted for publication, 1995.

[9] Y. Saad, *SPARSKIT: a basic tool kit for sparse matrix computations*, tech. rep., Univesity of Minesota, Minneapolis, 1994.

[10] A. van der Sluis and H. A. van der Vorst, *The rate of convergence of conjugate gradients*, Numer. Math., 48 (1986), pp. 543–560.