*Bull. Nov. Comp. Center*, Num. Anal., 11 (2002), 1–7
© 2002 NCC Publisher

# On CSlR-ILU preconditioning and its implementation in GMRES(m)

M. Balandin, E. Shurina

An ILU(0) modification for sparse matrices storage technique called the CSlR is discussed and briefly analyzed. Details of program implementation for the GMRES(m) preconditioning are described for C language.

## 1. Introduction. On sparse matrices

When speaking about sparse matrices we first need to introduce some terms and notations.

Let $A$ be a matrix of the size $n$: $A = \{a_{ij}\}_{i,j=1}^{n}$. A subset $P_A$ of the set of complete indices $\{1,\dots,n\} \times \{1,\dots,n\}$, which corresponds only to *non-zero* elements, will be mentioned as *portrait* or *pattern* of the matrix $A$:

$$P_A = \{(i,j) \mid 1 \le i,j \le n, a_{ij} \ne 0\}.$$

In terms of patterns and non-zero elements all sparse matrices can be divided into three subclasses:

1. Symmetric matrices, for which $a_{ij} = a_{ji}$. It is obvious that the pattern of symmetric matrices is symmetric too, i.e.,

$$\forall (i,j) \in P_A : \quad (j,i) \in P_A. \tag{1}$$

2. "Slightly non-symmetric" matrices, for which (1) is still correct but

$$\exists (i,j) \in P_A : \quad a_{ij} \ne a_{ji}.$$

3. "Absolutely non-symmetric" matrices, for which

$$\exists (i,j) \in P_A : \quad (j,i) \notin P_A.$$

Actually, matrices of third kind can always be transformed into second kind by extension of their patterns until (1) is satisfied [4], but it also extends memory storage and we will not consider this method here.

The matrices of *second* kind are of our interest, since we always obtain them in finite element (FE) and finite volume (FV) discretizations of physical problems. Actually, the mesh connectivity graph is non-oriented and hence the $P_A$ (which represents this graph's structure in global SLE) satisfies (1) immediately.

Using a non-structured mesh with local refinements, we unavoidably obtain non-structured pattern of the global SLE matrix and therefore cannot take the advantages of the well-known storage methods widely used in finite differences methods [6]. Some other techniques should be used for these matrices.

For further estimations let us introduce a *mean number of non-diagonal non-zero elements in matrix' row for lower/upper triangle*[1], which will be referenced to as $m$. In estimations of storage requirements, we use *machine word* (4 bytes) as a unit, since it perfectly describes the case of single-length float numbers (for $a_{ij}$ etc) and double-length integers (for indices).

Probably the most used storage method for arbitrary sparse matrices is the CSR[2] [4, 5, 6], which is basically the list-oriented data organization. In this scheme, all non-zero elements are stored together with their positions in the matrix $A$ as three lists $\langle\{a_{ij}\},\{i\},\{j\}\rangle$. Total storage is therefore

$$V_{\mathrm{CSR}}(n) = 3n(2m+1) \tag{2}$$

words.

Obviously, the CSR is not effective for "slightly non-symmetrical" matrices as we described them before. The reason is abundance of $P_A$ clearly visible from (1). Actually, one half of this set can be easily obtained from another one, while diagonal elements $a_{ii}$ (they are always non-zero in the FE and the FV!) do not require any indices at all.

The solution is the modification of the CSR known as the CSlR[3]. Let us store the main diagonal of $A$, its lower and upper triangles $A_L$ and $A_U$ *separately*, then only half-size subset of $P_A$ should be stored to completely describe the matrix $A$:

$$P_A^* = \{(i,j) \mid (i,j) \in P_A,\ i \geq j\} \subset P_A.$$

Therefore, $A$ can be presented as formal structure

$$A^{\mathrm{CSlR}} = \langle D, I, J, L, U\rangle, \tag{3}$$

where

- $D = \{a_{ii}\}_{i=1}^n$ are diagonal elements;
- $L$ and $U$ store non-diagonal elements of lower and upper triangles, respectively, listed *in the same order* (row-oriented for $L$ and column-oriented for $U$);

[1]This number is closely related to geometrical quality of the mesh and the FE/FV order of approximation [6].

[2]Compressed Sparse Row.

[3]Compressed Sparse low-triangle Row.

- $J$ stores column indices for $a_{ij} \in L$ (and, at the same time, row indices for $a_{ij} \in U$);
- $I$ stores numbers of non-zero elements of each row in $L$ (and, at the same time, of each column in $U$).

One can easily calculate total storage for the CSlR, which is equal to

$$V_{\text{CSlR}}(n) = n(3m + 2) \tag{4}$$

words. Then, comparing (2) to (4) we see

$$\frac{V_{\text{CSR}}(n)}{V_{\text{CSlR}}(n)} = \frac{6m + 3}{3m + 2} \approx 2,$$

which means that the CSlR is *twice* more effective than the CSR.

## 2. The ILU(0) decomposition for the CSlR scheme

From programmer's viewpoint, the key feature of the Krylov sequence methods is the fact that they do not require the matrix $A$ "as it is", but only some ability to calculate the matrix-to-vector product $Ax \to y$ in order to find residual vectors [1, 6]. Implementation of corresponding procedure is not a problem for both the CSR and the CSlR (see algorithms in [4]) but the ILU preconditioning always leads to certain difficulties since its effectiveness depends very hard on operations with matrix' elements and sequential access to them.

In this section, we present modification of the ILU algorithm suitable for the CSlR scheme and auxiliary algorithms of upward and backward substitutions. The latter are required for "unpreconditioning" of residual vectors.

### 2.1. ILU itself

It is well-known that the *complete* LU factorization of dense matrices can be written in many modifications [7]. Their common bottleneck is calculating of sums like

$$\sum_k l_{ik} u_{kj}, \tag{5}$$

which can be really difficult and slow with sparse matrices[4].

Calculation of an ILU factorization can be stated in terms of pattern as the following formal problem: *find the matrices $L$, $U$, and $R$ satisfying the equation $A = LU + R$ and three conditions*:

[4]The reason is complicated access to elements of the matrix. Instead of direct access through indices we first need to locate corresponding fragment of list $L$ or $U$ and then search through it (probably unsuccessful, if $a_{ij} = 0$).

1. $P_L \subset P_A^*$ and $P_U \subset (P_A^*)^T$;
2. $\forall(i,j) \in P_A: \ [LU]_{ij} = a_{ij}$;
3. $P_R \cap P_A = \emptyset$.

It means that we should extend (3) with three additional lists $L'$, $U'$, and $D'$ keeping elements of decomposition:

$$\langle D, I, J, L, U\rangle \to \langle D, I, J, L, U, D', L', U'\rangle.$$

Taking into account the internal nature of presentation (3), we state two additional requirements to algorithm itself:

1. The order of calculating $l_{ij}$ and $u_{ij}$ must be the same as the order elements $a_{ij}$ listed in;
2. The sums of the kind (5) must be calculable as easily and fast as possible.

Using the technique presented in [4], we can derive the CSlR modification of the ILU(0) decomposition shown in Figure 1.

```
for $i = 1$ to $n$
    for $j = 1$ to $i-1$
        if $(i,j) \in P_A$ then $l_{ij} := a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}$
                                $u_{ji} := \frac{1}{l_{jj}}\left[a_{ji} - \sum_{k=1}^{j-1} l_{jk}u_{ki}\right]$
    increase $j$
    $l_{ii} := a_{ii} - \sum_{k=1}^{i-1} l_{ik}u_{ki}$
    $u_{ii} := 1$
increase $i$
```

**Figure 1.** CSlR-ILU(0) algorithm

The first requirement is satisfied since $(L, L')$ are both keeping elements in row-oriented order, and, symmetrically, $(U, U')$ in column-oriented one. Looking on the order of calculation one can see that $l_{ij}$ and $u_{ji}$ are accessed simultaneously.

The second condition is satisfied too: note how order of access to $l_{ij}$ (and in the same time to $u_{ij}$) corresponds to the order of storage in $L$ and $U$.

By this mean, the presented algorithm seems to be a good choice for the FE/FV matrices. Actually, in [3] it was shown that for these matrices the CSlR scheme is much more effective than the CSR in all senses.

Please note that sums in the algorithm of Figure 1 cannot be implemented *directly*; some indirect addressing from $I$ through $J$ to elements of $L$ and $U$ should be used instead. However, it is quite straightforward and is easily programming trick.

Note also that "$u_{ii} := 1$" couples $D'$ with the *lower* triangle $L'$, while for the upper one $U'$ no storage of diagonal elements is required.

## 2.2. Auxiliary algorithms

If one wants to use any factorization algorithm as preconditioner in the Krylov subspace methods, he also needs an algorithm for "unpreconditioning" [4, 6]. In the ILU case it means that *upward* and *backward* substitution procedures are required for $L^{-1}f \to z$ and $U^{-1}f \to z$, respectively.

Obviously, the basic requirement of previous subsection is still actual: *algorithms and their implementations should use the internal structure of data storage method (3) to be as fast and effective as possible.*

Luckily, in this case it does not lead to any troubles at all. Taking upward and backward substitution algorithms from [4] and changing them a bit according to our specific situation (particularly, "$u_{ii} := 1$" again), we obtain algorithms shown in Figures 2 and 3.

In order to better show the CSlR features and ideas of implementation linear algebra subroutines for this storage method, we present these algorithms in details.

Note how order of calculations differs for $L$ and $U$ lists according to the storage order.

```
Input: $f$, $D'$, $L'$, $I$, $J$, $n$
Output: $z = L^{-1}f$
Effects: changing of $f$
— BEGIN —
for $i = 1$ to $n$
    for $j = I_i$ to $I_{i+1} - 1$
        $f_i := f_i - z_{J_j} L'_j$
    increase $j$
    $z_i := f_i / D'_i$
increase $i$
— END —
```

**Figure 2.** Upward substitution algorithm for the CSlR

```
Input: $f$, $U'$, $I$, $J$, $n$
Output: $z = U^{-1}f$
Effects: changing of $f$
— BEGIN —
for $i = n$ to 1 step $(-1)$
    $z_i := f_i$
    for $j = I_i$ to $I_{i+1} - 1$
        $f_{J_j} := f_{J_j} - z_i U'_j$
    increase $j$
increase $i$
— END —
```

**Figure 3.** Backward substitution algorithm for the CSlR

## 3. Program implementation

The algorithms listed above were implemented by the authors in C language. The basic idea of implementation was *to take all advantages of the CSlR scheme while keeping in mind the ability to change preconditioner if required.*

The whole package consists of four functions: GMRES (GMRES(m) itself with ability of preconditioning); LU_itself (CSlR-ILU decomposition according to Figure 1); UPW_subst (upward substitution according to Figure 2); BACKW_subst (backward substitution according to Figure 3).

The prototypes should be described in user's program as follows:

```
void LU_itself(long n, long *I, long *J,
     float *D, float *L, float *U, float *D1,
     float *L1, float *U1);
void UPW_subst(float *f, float *D1, float *L1,
     long *J, long *I, long n, float *z);
void BACKW_subst(float *f, float *U1, long *J,
     long *I, long i, float *z);
int  GMRES(long R, int M, float *x, float *b,
     long n, float EPS);
```

where R is the restriction for the number of the GMRES iterations; M is the dimension of the Krylov subspace in GMRES; x is the solution (and, in the beginning, the initial guess); b is the right-hand part of the SLE $Ax = b$; EPS is the required solution accuracy ($\|r_k\|/\|r_0\| < \varepsilon$). The meaning of other variables is clear as they exactly correspond to notation we used above. GMRES returns 1 as the error code in case of insufficient memory and 0 as OK code.

The user's program should contain two functions: mat2vec (calculation of the matrix-to-vector product) and unLUsubst ("unpreconditioning" of the vector, normally described via UPW_subst and BACKW_subst):

```
void mat2vec(int n, float *X_input, float *X_output);
void unLUsubst(float *r);
```

Before calling to GMRES user's program should call LU_itself to calculate preconditioner matrix.

There is also an ability to use the package as usual unpreconditioned GMRES. To do it, a user just defines unLUsubst function as

```
void unLUsubst(float *r) { return; }
```

Calling to LU_itself before GMRES is not required in that case.

## References

[1] Balandin M., Chernyshev O., Shurina E. Analysis of methods for solving large-scale non-symmetrical linear systems with sparsed matrices // Proc. 4th Int. Conf. "Parallel Computing Technologies" (PaCT-97) / Lecture Notes in Computer Science. – Springer-Verlag, 1997. – Vol. 1277. – P. 336–343.

[2] Balandin M.Yu., Shurina E.P. Some estimations of efficiency for parallel SLE solving algorithms of Krylov sequence type // Computational Technologies. – 1998. – Vol. 3, № 1. – P. 23–30 (in Russian).

[3] Balandin M., Chernyshev O., Shurina E. The ILU preconditioning for systems of linear equations with sparse matrices of arbitrary structure // Proc. Int. Conf. Honour. Acad. Godunov "Mathematics in Applications". – Novosibirsk, 1999. – P. 26–27.

[4] Balandin M.Yu., Shurina E.P. Numerical Methods for Large-Scale SLEs. – Novosibirsk: NSTU Publishing, 2000 (in Russian).

[5] Chow E., Saad Y. ILUS: an incomplete LU preconditioner in sparse skyline format // Int. J. for Num. Meth. in Fluids. – 1997. – Vol. 25. – P. 739–748.

[6] Saad Y. Iterative Methods for Sparse Linear Systems. – PWS Publishing Company, 1996.

[7] Stewart G.W. A Survey of Matrix Algorithms. Vol. 1: Basic Decompositions. – University of Maryland, 1995.