Parallelizing Kogbetliantz Method

Vjeran Hari* and **Vida Zadelj-Martić****

[1] Department of Mathematics, University of Zagreb, P.O. Box 335, 10002 Zagreb, Croatia.
[2] Faculty of Geodesy, University of Zagreb, P.O. Box 335, 10002 Zagreb, Croatia.

Received November 30, 2005

A way how the two-sided Jacobi-type method for computing the singular value decomposition of triangular matrices, known as Kogbetliantz method, can be adapted for use with parallel computers, is presented. It is also shown how the method can be further modified to work with blocks. In both cases, the initial, possibly rectangular matrix, has to be brought to a special butterfly-like form. In the iterative part of the algorithm, this special form gradually changes, but after a fixed number of parallel steps, which corresponds to two standard sweeps, the initial butterfly-like form is retained.

Keywords: singular value decomposition, Kogbetliantz method, parallel algorithm
AMS subject classification: 65F16

0 Introduction

Recently, Drmač and Veselić [8, 9] have modified the one-sided Jacobi method for computing the singular value decomposition (SVD) of general matrices, to become faster than the QR method and almost as fast as the Divide and Conquer (DC) method. Since the Jacobi method has been proved to be relatively accurate (see [5]), which cannot be said for the QR and DC methods, the diagonalization-type methods have drawn new attention of researchers in the field of matrix eigenvalue computation.

Several key ideas from [8, 9] can be used in connection with the Kogbetliantz method for computing the SVD of triangular matrices [21, 22, 15]. Typically, the preprocessing (see [9]) which uses one or two QR factorizations, can be used with the Kogbetliantz method. Computation of the right or left singular vectors a posteriori (using the starting triangular matrix and the computed data, see [7, 9]) can be done in the same way with the Kogbetliantz method. We note that the Kogbetliantz method is relatively accurate [20] (the proof is partly based on the proof in [23]). Therefore, Kogbetliantz method is in many respects similar to the two-sided Jacobi method for positive definite matrices. But, this comes also with the main weakness of the method when compared to the one-sided Jacobi. One-sided methods are fast because they operate only with columns. BLAS 1 operations with columns are several times faster than the same operations with rows. And in the Kogbetliantz method the columns and the rows are rotated. Although, this is a severe drawback, the two-sided methods have also some advantages over one-sided methods. In particular, they can easily terminate the process, without any extra cost. It is known, that for one-sided Jacobi methods, each check for the convergence, costs around $n^2/2$ dot products.

Another advantage of the one-sided method lies in its inherent parallelism, which makes it amenable for distributed memory computing. Here, we show how Kogbetliantz method can be adapted for parallel computing with shared memory. It is interesting, that our approach makes it possible to almost eliminate the slowdown coming from the operations with rows.

We finally note, that one-sided Jacobi methods can easily be turned into BLAS 3 algorithms [19]. This means working with block-columns instead of columns alone. Here we show that Kogbetliantz can also be made BLAS 3 algorithm in a similar way.

The rest of the paper is divided into three sections. In Section 1, we introduce the butterfly form and briefly describe how to reduce an arbitrary rectangular matrix to this form. In Section 2, we describe the parallel strategy

* e-mail: hari@math.hr, [1]
** e-mail: vzadelj@geodet.geof.hr, [2]

and the parallel Kogbetliantz method. In Section 3, we show how to modify Kogbetliantz method to become a BLAS 3 algorithm.

1 Reduction to Butterfly Form

In this sectio we define the butterfly form and show how to reduce a general rectangular matrix to this form using standard tools such as Householder reflectors and Givens rotations. Since our aim is to construct a fast SVD algorithm, it should use the same preprocessing (which also serves as preconditioning) as in [9]. It means using the QR factorization with column pivoting, followed possibly by the LQ factorization. Therefore, we also show how to obtain the butterfly form from a matrix in triangular form.

1.1 Butterfly Form

For $n = 6$ and $n = 7$, the butterfly form of a square matrix B of order n, has the following appearance

$$B = \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & x \\ x & x & x & 0 & 0 & x & x \\ x & x & x & x & x & x & x \\ x & x & x & 0 & x & x & x \\ x & x & 0 & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}.$$

Here x denotes an element of B which need not be zero.

On conventional computers it is easy to reduce any rectangular matrix A to butterfly form. The algorithm is very similar to the one for computing the QR factorization. Let A be $m \times n$, $m \geq n$. One can use a finite set of Householder reflectors or the appropriate sequence of Givens rotations, applying them from the left side. This approach will preserve the relative accuracy of the singular values and vectors in the presence of rounding errors provided that A is well-behaved.

We shall denote by H_i and H_i' the Householder matrices (the direct sum of the appropriate identity matrices and the Householder reflectors) which are used at step i of the reduction process. The algorithm has the following form

$$B = \begin{cases} H_k' H_k \cdots H_2' H_2 H_1' H_1 \, A, & k = n/2, \ \ n \text{ even} \\ B = H_k' H_k \cdots H_2' H_2 \, H_1 A \,, & k = (n+1)/2, \ \ n \text{ odd.} \end{cases}$$

Here B is the matrix in butterfly form. Each matrix H_i (and H_i') makes an appropriate number of zeros in the current column. The procedure is illustrated for $m = 7$, $n = 6\,, \, 5$. The symbols $*$ and $\star$ denote the elements which will be zeroed by H_i and H_i', respectively, in the ith step of the algorithm

$$\begin{bmatrix} x & x & \star & * & x & x \\ x & x & \star & * & x & x \\ x & x & x & * & x & x \\ x & x & x & x & x & x \\ x & x & \star & * & x & x \\ x & x & \star & * & x & x \\ x & x & \star & * & x & x \end{bmatrix} \to \begin{bmatrix} x & \star & 0 & 0 & * & x \\ x & x & 0 & 0 & * & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & \star & 0 & 0 & * & x \\ x & \star & 0 & 0 & * & x \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & 0 & * \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \\ \star & 0 & 0 & 0 & 0 & * \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} x & x & * & x & x \\ x & x & * & x & x \\ x & x & x & x & x \\ x & x & * & x & x \\ x & x & * & x & x \\ x & x & * & x & x \\ x & x & * & x & x \end{bmatrix} \to \begin{bmatrix} x & \star & 0 & * & x \\ x & x & 0 & * & x \\ x & x & x & x & x \\ x & x & 0 & x & x \\ x & \star & 0 & * & x \\ x & \star & 0 & * & x \\ x & \star & 0 & * & x \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & * \\ x & x & 0 & 0 & x \\ x & x & x & x & x \\ x & x & 0 & x & x \\ x & 0 & 0 & 0 & x \\ \star & 0 & 0 & 0 & * \\ \star & 0 & 0 & 0 & * \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & x \\ x & x & x & x & x \\ x & x & 0 & x & x \\ x & 0 & 0 & 0 & x \\ \hline 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The matrix in butterfly form is square, so we have separated it from the remaining zero rows by the line.

If A has more columns than rows, we can work with A^T, as is usual in the SVD computation.

We shall skip an attempt to parallelize these algorithms (which use Householder or Givens matrices) for two reasons. First, these algorithms should be similar to the appropriate parallel QR factorization algorithms for the given multiprocessor machine. Second, and more important, we want to precondition the initial matrix for the iterative part of the SVD computation. This can be accomplished (c.f. [9]) by applying an appropriate parallel QR factorization algorithm followed possibly by an appropriate parallel LQ factorization algorithm for the given multi-processor machine. We assume that efficient implementations of such algorithms are at disposal.

Let T be a triangular matrix of order n (obtained by QR or by QR followed by LQ factorization). We shall show how the butterfly form can easily be obtained from T. We can assume that T is upper-triangular. Otherwise we can work either with the transpose of T or we can construct a similar algorithm for a lower-triangular T.

We shall show that T can be reduced to butterfly form using acheap and exact similarity transformation by a permutation matrix: $B = P^T T P$. The permutation matrix P is constructed as the following product of transposition matrices,

$$P = \begin{cases} I_{12}I_{13}(I_{14}I_{23})(I_{15}I_{24})(I_{16}I_{25}I_{34}) \cdot (I_{1,n}I_{2,n-1} \cdots I_{k,k+1}) & \text{if } n = 2k \\ I_{12}I_{13}(I_{14}I_{23})(I_{15}I_{24})(I_{16}I_{25}I_{34}) \cdot (I_{1,n}I_{2,n-1} \cdots I_{k,k+2}) & \text{if } n = 2k+1\,. \end{cases} \tag{1.1}$$

Here, $I_{pq} = [e_1, \ldots, e_q, \ldots, e_p, \ldots, e_n]$ where e_j, $1 \le j \le n$ are the columns of the identity matrix I_n. The parentheses are used to emphasize that the corresponding transformations in the reduction process can be made in parallel. For example, for $n = 6$ we have $k = n/2 = 3$, and the procedure takes the form

$$\begin{bmatrix} x & * & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \to \begin{bmatrix} x & 0 & * & x & x & x \\ x & x & x & x & x & x \\ 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & * & x & x \\ x & x & * & x & x & x \\ x & 0 & x & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix}$$

$$\to \begin{bmatrix} x & 0 & 0 & 0 & * & x \\ x & x & 0 & * & x & x \\ x & x & x & x & x & x \\ x & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & 0 & * \\ x & x & 0 & 0 & * & x \\ x & x & x & * & x & x \\ x & x & 0 & x & x & x \\ x & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & x \end{bmatrix} \to \begin{bmatrix} x & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & x \\ x & x & x & 0 & x & x \\ x & x & x & x & x & x \\ x & x & 0 & 0 & x & x \\ x & 0 & 0 & 0 & 0 & x \end{bmatrix}$$

Here each $*$ denotes the position of one pivot element. Its subscripts determine the rows and columns which are to be swapped. For example, in the 5th displayed matrix of order 6, the pivot positions are $(1, 6)$, $(2, 5)$, $(3, 4)$. So, the rows (columns) 1 and 6, 2 and 5, 3 and 4 will be swapped. We see that all these transformations can be performed instantaneously if three processors are available. From the formula above, we conclude that $n - 1$ "parallel steps" are needed provided that around $n/2$ processors are available.

To trace the movement of the non-zero elements, one can use the following code which is the essential part of the FORTRAN 77 subroutine for this reduction.

```
      DO 1 K=2,N
C                SWAPPING COLUMNS
        DO 2 I=1,K/2
         J=K+1-I
         CALL SSWAP(N,A(1,I),1,A(1,J),1)
   2    CONTINUE
C                  SWAPPING ROWS
        DO 3 I=1,K/2
         J=K+1-I
         CALL SSWAP(N,A(I,1),LDA,A(J,1),LDA)
   3    CONTINUE
 1     CONTINUE
```

Here, `N` is order of the two-dimensional array `A`, `LDA` is the leading dimension of `A` and `SSWAP` is the BLAS1 routine which performs a swap. We can notice that swapping columns requires unit stride, while swapping rows requires stride `LDA`. Note that the inner loops allow for parallel processing.

The effect of this transformation can be seen in the following example

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 0 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ 0 & 0 & 33 & 34 & 35 & 36 & 37 & 38 \\ 0 & 0 & 0 & 44 & 45 & 46 & 47 & 48 \\ 0 & 0 & 0 & 0 & 55 & 56 & 57 & 58 \\ 0 & 0 & 0 & 0 & 0 & 66 & 67 & 68 \\ 0 & 0 & 0 & 0 & 0 & 0 & 77 & 78 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 88 \end{bmatrix} \longrightarrow \begin{bmatrix} 88 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 68 & 66 & 0 & 0 & 0 & 0 & 0 & 67 \\ 48 & 46 & 44 & 0 & 0 & 0 & 45 & 47 \\ 28 & 26 & 24 & 22 & 0 & 23 & 25 & 27 \\ 18 & 16 & 14 & 12 & 11 & 13 & 15 & 17 \\ 38 & 36 & 34 & 0 & 0 & 33 & 35 & 37 \\ 58 & 56 & 0 & 0 & 0 & 0 & 55 & 57 \\ 78 & 0 & 0 & 0 & 0 & 0 & 0 & 77 \end{bmatrix}.$$

2 Kogbetliantz Method under Modulus Strategy

The Kogbetliantz method is a known two-sided Jacobi-type method for computing the SVD of a square matrix. Since its discovery [21, 22] the method has been studied with respect to its global and quadratic convergence [12], [14], [29], [3] and implementation details [4]. Later, it has been discovered [15], [4] that the method becomes simpler and more efficient if it is applied to triangular matrices under the serial (the row- or column-cyclic) pivot strategies. Then it behaves similar to the standard Jacobi method for symmetric matrices. The global convergence properties of this method have been studied in [15], [10] and the quadratic convergence in [16, 27, 17]. Its relative accuracy has been recently proved [20, 23]. Typically, this method is applied to triangular matrices obtained after the QR factorization. It can be shown that the method preserves the non-negativity (and positivity) of the diagonal elements. So, if Φ is an appropriate diagonal matrix of signs, it is advantageous to work with ΦR, where $\mathrm{diag}(\Phi R) \geq 0$. This choice simplifies the algorithm even further.

Here we show that all these good properties of Kogbetliantz method can be preserved if the method is applied to a matrix in butterfly form. But the method has to be defined by a special pivot strategy known as (see [26]) *modulus strategy*. In addition, the method becomes amenable for parallel computing. We shall refer to it as Kogbetliantz method under the modulus strategy or shorter KMMS. We note, that perhaps the first approach to parallelizing the Kogbetliantz method is due to Luk [24]. We note also the papers [1], [26, 25], [13] and [28, 2, 1].

Starting from the matrix B in butterfly form, KMMS generates a sequence of matrices $B^{(k)} = (b_{lm}^{(k)})$, $k \geq 0$, using the rule

$$B^{(k+1)} = (U^{(k)})^T B^{(k)} V^{(k)}, \quad k \geq 0, \quad B^{(0)} = B\,.$$

Here $U^{(k)} = U_{ij}^{(k)}$ and $V^{(k)} = V_{ij}^{(k)}$, are simple rotations in the (i,j)-plane, where $i = i(k)$ and $j = j(k)$, $i < j$, is the *pivot pair*. If $b_{ij}^{(k)} \neq 0$ or $b_{ji}^{(k)} \neq 0$ the rotation angles $\varphi^{(k)}$, $\psi^{(k)}$ of $U^{(k)}$, $V^{(k)}$, respectively, are determined from the requirement

$$b_{ij}^{(k+1)} = b_{ji}^{(k+1)} = 0, \quad \text{which implies} \quad \|\Omega(B^{(k+1)})\|^2 = \|\Omega(B^{(k)})\|^2 - |b_{ij}^{(k)}|^2 - |b_{ji}^{(k)}|^2.$$

Here, $\Omega(X) = X - \mathrm{diag}(X)$ denotes the off-diagonal part of X and $\|X\|$ is the Frobenius norm of X. In the following subsections we first show how to compute the essential elements of $U^{(k)}$ and $V^{(k)}$, then we describe the modulus strategy and finally, we show how the initial zero pattern in the matrix changes with the iterations. Then we shortly describe the sequential and parallel algorithm.

2.1 The Kogbetliantz algorithm for 2×2 triangular matrices

We start with the case of an upper-triangular 2×2 matrix T whose non-zero elements are denoted by f, g and h. A single Kogbetliantz step diagonalizes T through the orthogonal transformation

$$T' = \begin{bmatrix} f' & 0 \\ 0 & h' \end{bmatrix} = \begin{bmatrix} c_\varphi & s_\varphi \\ -s_\varphi & c_\varphi \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} c_\psi & -s_\psi \\ s_\psi & c_\psi \end{bmatrix} = U^T T V. \tag{2.1}$$

Here c_φ, s_φ, c_ψ and s_ψ denote $\cos(\varphi)$, $\sin(\varphi)$, $\cos(\psi)$ and $\sin(\psi)$, respectively. We shall also use t_φ and t_ψ for $\tan(\varphi)$ and $\tan(\psi)$, respectively. It can be easily deduced (see [20]) that

$$f' = \frac{c_\varphi}{c_\psi} f = \frac{s_\psi}{s_\varphi} h = \frac{c_\psi}{c_\varphi} f + \frac{s_\psi}{c_\varphi} g = \frac{s_\varphi}{s_\psi} h + \frac{c_\varphi}{s_\psi} g, \tag{2.2}$$

$$h' = \frac{c_\psi}{c_\varphi} h = \frac{s_\varphi}{s_\psi} f = \frac{c_\varphi}{c_\psi} h - \frac{s_\varphi}{c_\psi} g = \frac{s_\psi}{s_\varphi} f - \frac{c_\psi}{s_\varphi} g. \tag{2.3}$$

There are several formulas for computing c_φ, s_φ, c_ψ and s_ψ. If the right transformation is applied to T first, we obtain (using the notation from [20]) the UR (Upper-triangular, Right transformation first) algorithm. If the left transformation is applied first, we obtain the UL algorithm.

UL Algorithm

$$\tan 2\varphi = \frac{2gh}{f^2+g^2-h^2}\,.$$

$$\tan\psi = \frac{g+ht_\varphi}{f} = \frac{ft_\varphi}{h-gt_\varphi}\,.$$

UR Algorithm

$$\tan 2\psi = \frac{2fg}{f^2-g^2-h^2}\,.$$

$$\tan\varphi = \frac{ft_\psi-g}{h} = \frac{ht_\psi}{f+gt_\psi}\,.$$

The above formulas are used for computing c_ψ, s_ψ, c_φ, s_φ. The formulas (2.2) and (2.3) hold with both, UR and UL algorithms.

If T is lower-triangular, a single Kogbetliantz step takes form

$$T' = \begin{bmatrix} f' & 0 \\ 0 & h' \end{bmatrix} = \begin{bmatrix} c_\varphi & s_\varphi \\ -s_\varphi & c_\varphi \end{bmatrix} \begin{bmatrix} f & 0 \\ g & h \end{bmatrix} \begin{bmatrix} c_\psi & -s_\psi \\ s_\psi & c_\psi \end{bmatrix} = U^T T V.$$

Transposing this equation, one can invoke the above formulas and obtain (see [20])

LL Algorithm

$$\tan 2\varphi = \frac{2fg}{f^2-g^2-h^2}\,,$$

$$\tan\psi = \frac{ft_\varphi-g}{h} = \frac{ht_\varphi}{f+gt_\varphi}\,,$$

LR Algorithm:

$$\tan 2\psi = \frac{2gh}{f^2+g^2-h^2}\,,$$

$$\tan\varphi = \frac{g+ht_\psi}{f} = \frac{ft_\psi}{h-gt_\psi}\,.$$

The formulas (2.2) and (2.3) are translated into

$$f' = \frac{c_\psi}{c_\varphi}f = \frac{s_\varphi}{s_\psi}h = \frac{c_\varphi}{c_\psi}f + \frac{s_\varphi}{c_\psi}g = \frac{s_\psi}{s_\varphi}h + \frac{c_\psi}{s_\varphi}g\,, \tag{2.4}$$

$$h' = \frac{c_\varphi}{c_\psi}h = \frac{s_\psi}{s_\varphi}f = \frac{c_\psi}{c_\varphi}h - \frac{s_\psi}{c_\varphi}g = \frac{s_\varphi}{s_\psi}f - \frac{c_\varphi}{s_\psi}g\,. \tag{2.5}$$

Note that formulas (2.2)–(2.5) imply that the non-negativity (positivity) of f and g implies the non-negativity (positivity) of f' and g'.

As is shown in [20], it is always possible to chose between UR and UL (LR and LL) algorithms to achieve that c_ψ, s_ψ, c_φ, s_φ are computed with small relative error. This is the basis of the accuracy proof for the serial [20] and modulus Kogbetliantz method. The accuracy proof naturally extends to the modulus strategy.

2.2 Modulus pivot strategy

This pivot strategy was first described in [15] and the name is first mentioned in [26]. For simplicity, we can regard it as a special cyclic (pivot) strategy. The *modulus ordering* of the set $\mathbf{P}_n = \{\,(p,q)\;:\;1\le p<q\le n\,\}$ which defines the strategy, will be depicted using the upper-triangular part of the matrix of order $n=7$. The displayed numbers in the leftmost matrix represent the ordering in which the pivot elements are annihilated within one cycle. Also, the position of each number in the matrix determines the position of the appropriate pivot element.

By $\mathcal{S}_t$, $1\le t\le 7$ we denote the so called *rotation sets*, which are the sets of index pairs of those pivot elements which can be simultaneously annihilated in the current "parallel step". This means that, using three processors, each triple of elements b_{ij}, b_{ii}, b_{jj}, $(i,j)\in\mathcal{S}_t$ can be fetched by one processor, so that all rotation matrices $U_{ij}^{(k)}$, $V_{ij}^{(k)}$ can be computed simultaneously.

In the rightmost matrix, we have used the same number to denote pivot positions of elements that are simultaneously annihilated. In other words, these numbers denote the schedule of the parallel steps.

$$\begin{bmatrix} \cdot & 6 & 9 & 11 & 14 & 16 & 19 \\ & \cdot & 12 & 15 & 17 & 20 & 1 \\ & & \cdot & 18 & 21 & 2 & 4 \\ & & & \cdot & 3 & 5 & 7 \\ & & & & \cdot & 8 & 10 \\ & & & & & \cdot & 13 \\ & & & & & & \cdot \end{bmatrix} \qquad \begin{aligned} \mathcal{S}_1 &= \{(2,7),(3,6),(4,5)\} \\ \mathcal{S}_2 &= \{(3,7),(4,6),(1,2)\} \\ \mathcal{S}_3 &= \{(4,7),(5,6),(1,3)\} \\ \mathcal{S}_4 &= \{(5,7),(1,4),(2,3)\} \\ \mathcal{S}_5 &= \{(6,7),(1,5),(2,4)\} \\ \mathcal{S}_6 &= \{(1,6),(2,5),(3,4)\} \\ \mathcal{S}_7 &= \{(1,7),(2,6),(3,5)\} \end{aligned} \qquad \begin{bmatrix} \cdot & 2 & 3 & 4 & 5 & 6 & 7 \\ & \cdot & 4 & 5 & 6 & 7 & 1 \\ & & \cdot & 6 & 7 & 1 & 2 \\ & & & \cdot & 1 & 2 & 3 \\ & & & & \cdot & 3 & 4 \\ & & & & & \cdot & 5 \\ & & & & & & \cdot \end{bmatrix}$$

For example, in the third parallel step the following computation has to be done. First, the elements of the rotation matrices which annihilate the elements at positions $(4,7)$, $(5,6)$ and $(1,3)$ are simultaneously computed. Then all three processors simultaneously apply the right transformations. Finally, all the processors simultaneously apply the left transformations. If the left and the right singular vectors are wanted, then the right rotations can be accumulated while the left singular vectors are easily computed a posteriori, like in [9]. Many ideas and implementation details from [9] can be used here.

2.3 The general parallel algorithm

In order to derive an efficient modulus Kogbetliantz algorithm for general n, we first have to make a partition of the set $\mathbf{P}_n$ into n subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$, where each subset has around $n/2$ elements. These are called *rotation sets*. Each one of them has the property that all pairs contained in it are mutually disjoint. Two pairs (i,j) and (p,q) from $\mathbf{P}_n$ are *disjoint* or *commuting* if the corresponding sets $\{i,j\}$ and $\{p,q\}$ are disjoint. Hence, the rotation sets in parallel Jacobi-type methods take over the role of pivot pairs in sequential Jacobi-type methods.

At parallel step t (t can be considered as a time step), the rotation set $\mathcal{S}_t$ determines those elements which will be rotated. Consider the following sequence of rotation sets:

$$\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n, \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n, \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n, \mathcal{S}_1, \ldots$$

Let $Piv(t),\ t \geq 1$, denote the *pivot set* that is currently used as a rotation set. In the first (parallel) step it coincides with $\mathcal{S}_1$, in the second step it coincides with $\mathcal{S}_2$ and so on. At step n it coincides with $\mathcal{S}_n$, at step $(n+1)$ it coincides with $\mathcal{S}_1$, at step $(n+2)$ with $\mathcal{S}_2$ and so on. Thus, $Piv(t)$ runs through the sequence of rotation set $\mathcal{S}_1$, $\mathcal{S}_2$, … ,$\mathcal{S}_n$ in a cyclic fashion.

We set $B^{[1]} = B$.

At time step t, $t \geq 1$, all rotation matrices $U_{ij}^{[t]}$, $V_{ij}^{[t]}$, $(i,j) \in Piv(t)$ are computed using the elements of the same matrix $B^{[t]}$. Then the transformation

$$B^{[t+1]} = {U^{[t]}}^T B^{[t]} V^{[t]}, \qquad U^{[t]} = \prod_{(i,j)\in Piv(t)} U_{ij}^{[t]}, \quad V^{[t]} = \prod_{(i,j)\in Piv(t)} V_{ij}^{[t]}, \quad t \geq 1, \tag{2.6}$$

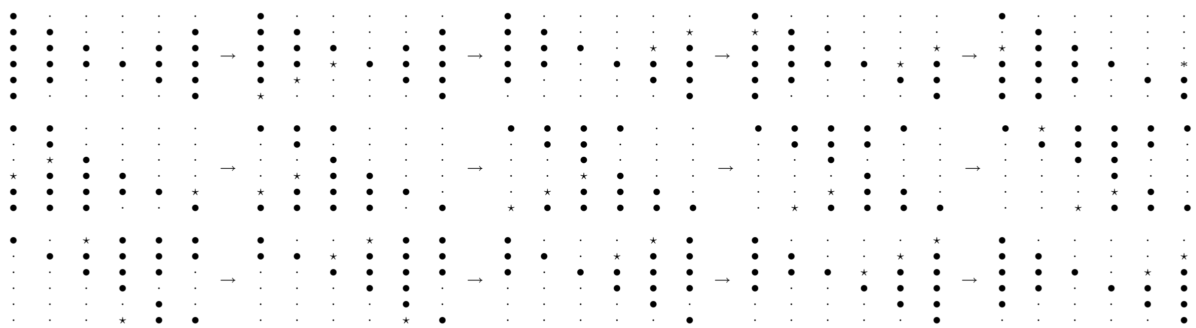
is performed. Here, $U^{[t]}$ and $V^{[t]}$ are not explicitly computed. Instead, all the factors $V_{ij}^{[t]}$, $(i,j) \in Piv(t)$ are simultaneously applied to $B^{[t]}$, and afterwards the same is done with $U_{ij}^{[t]}$, $(i,j) \in Piv(t)$.

Note that in the above relation the matrices $U^{[t]}$ are well defined, because all rotation matrices $U_{ij}^{[t]}$, $(i,j) \in Piv(t)$ mutually commute (the order of the factors is not relevant). The same is true for $V^{[t]}$ which is defined by $V_{ij}^{[t]}$, $(i,j) \in Piv(t)$. It is irrelevant whether the transformation $B^{[t]} \mapsto {U^{[t]}}^* B^{[t]}$ or $B^{[t]} \mapsto B^{[t]} V^{[t]}$ in (2.6) is performed first.

At time step t, the following tasks are performed: computation of the angle functions, application of the right transformation and application of the left transformation. If the right or the left or both singular vector matrices of B are wanted, then the right transformations $V^{[t]}$ can be accumulated into V, and then U can safely be computed a posteriori, from the equation $BV = U\Sigma$. Alternatively, one can accumulate $U^{[t]}$ into U and compute V a posteriori.

2.4 Using the butterfly form

If B is in butterfly form, then it is permutationally similar to an upper-triangular matrix (we say that B is PST). Therefore, it is also essentially triangular (ET) since $b_{pq}b_{qp} = 0$, $p < q$ holds. Butterfly form and modulus strategy go together since all generated matrices by the algorithm are PST. Indeed, the following figure depicts what happens with the zero structure during two consecutive cycles:

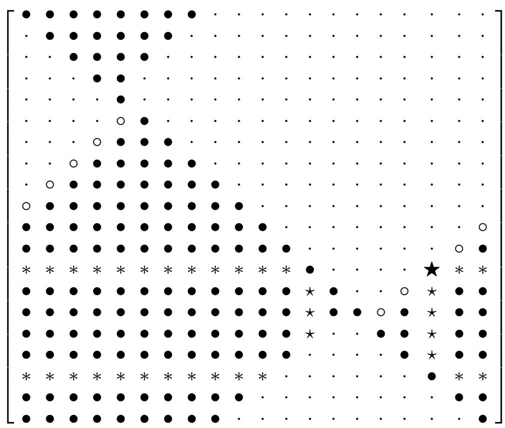


Note that we have used a preparatory "step zero" in the first cycle, which has annihilated the elements at positions $(6,1)$, $(5,2)$ and $(4,3)$. These pairs belong to the rotation set $\mathcal{S}_6$. The symbol $\bullet$ denotes a possibly nontrivial element, $\cdot$ a null-element and $\star$ a pivot element (which becomes zero). So, each cycle (or sweep) incorporates 6 parallel steps.

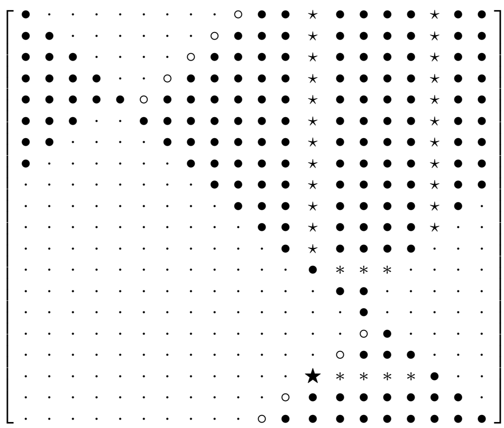
More generally, when n is odd, all rotation sets will contain the same number of $[\frac{n}{2}]$ pairs. For even n they contain in turn $[\frac{n}{2}]$ and $[\frac{n-1}{2}]$ pairs. Each cycle comprises n parallel steps.

Let us look closer at the matrix $B^{[t]}$. As an example, we have taken $n = 20$. In order to understand the effect of each left and right rotation within one parallel step, we have used $*$ to denote the elements which are transformed by ${U_{13,18}^{[11]}}^T$, and $\star$ to denote the elements which are transformed by $V_{13,18}^{[11]}$. The "non-zero" pivot element $b_{13,18}^{[11]}$ is denoted by $\bigstar$. The other (possibly) non-zero pivot elements for this parallel step are denoted by $\circ$. Diagonal elements are changed twice, but we use for them special transformation formulas, as noted earlier. Therefore, they are denoted by $\bullet$. The same symbol is used to denote any possibly nontrivial element.

$B^{[11]}$



$B^{[31]}$



In this example, all matrix elements have changed during the parallel step. Within each parallel step, almost all matrix elements will be transformed twice. For example, first the rows are transformed by left rotations and then the columns by right rotations. For $t = 31$, the structure of zeros takes form which can be considered as the "transpose" of that for $t = 11$.

In general, $B^{[t]}$ and $B^{[t-n]}$ have zero structures which are transposed to each other. It is known [15] that each matrix $B^{[t]}$ is PST. Therefore it is ET, so all matrices $B^{[t]}$ can be compactly saved in the upper triangle of a square array. Therefore, we can introduce the upper-triangular matrix $G^{[t]}$ by the requirement

$$G^{[t]} + G^{[t]T} = B^{[t]} + B^{[t]T}, \quad t \geq 0\,.$$

The question arises whether the KMMS can be described in terms of the matrices $G^{[t]}$. The answer is not only affirmative, but also the transformation formulas for $G^{[t]}$ are simple and similar to those for the symmetric Jacobi method, when only the upper-triangle of the symmetric matrix is used.

2.5 Modulus algorithm on sequential machines

The first thing to do in deriving the algorithm which uses $G^{[t]}$ is to settle $B^{[0]}$ in the upper triangle of $G^{[0]}$. In the following simple code which performs this task, the same array A is used to accommodate $B^{(0)}$ and $G^{(0)}$. On input A accommodates $B^{(0)}$ and $G^{[0]}$ is its output value. The code (in the left box) uses the fact that a matrix in butterfly form is ET.

If $B^{[0]}$ is obtained from some triangular matrix T (as discussed earlier) we can write a code which directly transforms the upper-triangular T into the upper-triangular $G^{[0]}$. Its essential part is displayed below in the right box.

```
    ZERO=0.0
    DO 1 K=1,N
      DO 2 I=K,(N+K-1)/2
       J=N+K-I
       TEMP=A(I,J)+A(J,I)
       A(I,J)=TEMP
       A(J,I)=ZERO
2     CONTINUE
      DO 3 I=1,(K-1)/2
       J=K-I
       TEMP=A(I,J)+A(J,I)
       A(I,J)=TEMP
       A(J,I)=ZERO
3     CONTINUE
1   CONTINUE
```

```
  DO 1 K=2,N
   DO 2 I=1,K/2
    J=K+1-I
    CALL SSWAP(I-1,A(1,I),1,A(1,J),1)
    CALL SSWAP(J-I-1,A(I,I+1),LDA,A(I+1,J),1)
    CALL SSWAP(N-J,A(I,J+1),LDA,A(J,J+1),LDA)
    TEMP=A(I,I)
    A(I,I)=A(J,J)
    A(J,J)=TEMP
    IF(SVECL)CALL SSWAP(N,U(1,I),1,U(1,J),1)
    IF(SVECR)CALL SSWAP(N,V(1,I),1,V(1,J),1)
2  CONTINUE
1 CONTINUE
```

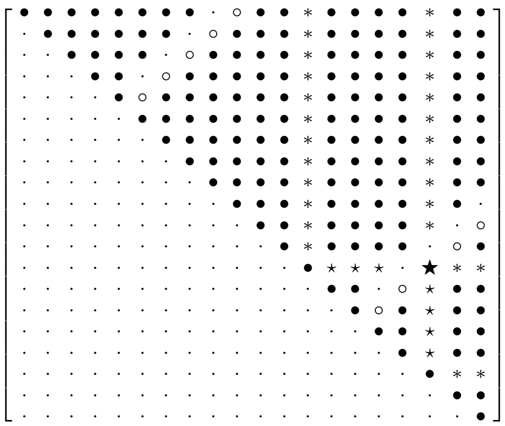
The latter code has the following effect

$$T=\begin{bmatrix} 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18\\ 0 & 22 & 23 & 24 & 25 & 26 & 27 & 28\\ 0 & 0 & 33 & 34 & 35 & 36 & 37 & 38\\ 0 & 0 & 0 & 44 & 45 & 46 & 47 & 48\\ 0 & 0 & 0 & 0 & 55 & 56 & 57 & 58\\ 0 & 0 & 0 & 0 & 0 & 66 & 67 & 68\\ 0 & 0 & 0 & 0 & 0 & 0 & 77 & 78\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 88 \end{bmatrix} \mapsto \begin{bmatrix} 88 & 68 & 48 & 28 & 18 & 38 & 58 & 78\\ 0 & 66 & 46 & 26 & 16 & 36 & 56 & 67\\ 0 & 0 & 44 & 24 & 14 & 34 & 45 & 47\\ 0 & 0 & 0 & 22 & 12 & 23 & 25 & 27\\ 0 & 0 & 0 & 0 & 11 & 13 & 15 & 17\\ 0 & 0 & 0 & 0 & 0 & 33 & 35 & 37\\ 0 & 0 & 0 & 0 & 0 & 0 & 55 & 57\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 77 \end{bmatrix} = G^{[0]}\,.$$

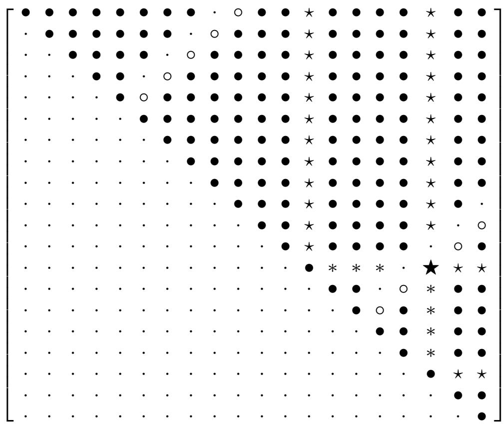
If T were lower-triangular, this code could easily be modified, or it could be used after transposing T. Note that after transposing T, left and right singular vectors have to be swapped.

Let us now consider the transformation of $G^{[t]}$. If we go back to the example matrix of order 20, we can trace how the transformation of $B^{[11]}$ influences $G^{[11]}$. We use the same notation as before.

$G^{[11]}$



$G^{[31]}$



From the displayed matrices, we can derive the transformation formulas for one rotational step. To simplify notation, let the essential elements of $U_{ij}^{[t]}$ and $V_{ij}^{[t]}$ be denoted as in (2.1): $c_{\phi_{ij}}$, $s_{\phi_{ij}}$ and $c_{\psi_{ij}}$, $s_{\psi_{ij}}$, respectively. Let us consider the transformation $A \mapsto [U_{ij}^{[t]}]^T A V_{ij}^{[t]} = A'$. If $A = (a_{lm})$ and $A' = (a'_{lm})$, then for the first cycle, we have:

Standard step:

$$\left.\begin{aligned} a'_{li} &= c_{\phi_{ij}} a_{li} - s_{\phi_{ij}} a_{lj}\\ a'_{lj} &= s_{\phi_{ij}} a_{li} + c_{\phi_{ij}} a_{lj} \end{aligned}\right\} \quad 1 \le l \le i-1 \qquad \left.\begin{aligned} a'_{il} &= c_{\psi_{ij}} a_{il} - s_{\psi_{ij}} a_{lj}\\ a'_{lj} &= s_{\psi_{ij}} a_{il} + c_{\psi_{ij}} a_{lj} \end{aligned}\right\} \quad i+1 \le l \le j-1$$

$$\left.\begin{aligned} a'_{il} &= c_{\phi_{ij}} a_{il} - s_{\phi_{ij}} a_{jl}\\ a'_{jl} &= s_{\phi_{ij}} a_{il} + c_{\phi_{ij}} a_{jl} \end{aligned}\right\} \quad j+1 \le l \le n \qquad \begin{aligned} & a'_{ii} = \frac{c_{\phi_{ij}}}{c_{\psi_{ij}}} a_{ii}\,, \quad a'_{jj} = \frac{c_{\psi_{ij}}}{c_{\phi_{ij}}} a_{jj}\\ & a'_{ij} = 0 \end{aligned}$$

In the second cycle the roles of angles ϕ_{ij} and ψ_{ij} are interchanged. If we speak in terms of parallel steps, then one cycle includes n such steps.

The above transformation formulas hold for odd cycles, while for even cycles each appearances of $c_{\psi_{ij}}$ and $s_{\psi_{ij}}$ should be replaced by $c_{\phi_{ij}}$ and $s_{\phi_{ij}}$, respectively, and vice versa. To keep the programming code compact, we shall assume that during odd cycles the variables `CSR` and `SNR` (`CSL` and `SNL`) take the values of $c_{\psi_{ij}}$ and $s_{\psi_{ij}}$ ($c_{\phi_{ij}}$ and $s_{\phi_{ij}}$) and during even cycles they take the values of $c_{\phi_{ij}}$ and $s_{\phi_{ij}}$ ($c_{\psi_{ij}}$ and $s_{\psi_{ij}}$). Then the code which transforms the array `A`, which accommodates the matrices $G^{[t]}$, becomes identical for all cycles. Some extra care has to be taken for updating the singular vector matrices.

Next, we display the essential parts of the programming code for KMMS, written in FORTRAN 77, double precision. This code is not yet adapted for parallel processing, but it is very simple and gives an idea how to write a code for standard computers. It uses subroutine `KOGSTEP` which performs one standard "rotational step" as described above. In `KOGSTEP` the BLAS 1 routine `DROT` (for applying a rotation to `A`) and the LAPACK routine `DLASV2` (for computing the SVD of an upper-triangular matrix of order two) have been used. To make the exposition clean, we stick with the industry-standard routines which are well documented. In a later stage we can replace `DLASV2` with a routine based on the UR, UL, LR and LL formulas described above. Logical variables `VL` and `VR` indicate whether left and right singular vectors are computed. The process is terminated when all off-diagonal elements become zero. The pivot element `A(I,J)` is set zero if `ABS(A(I,J))` is not larger than `EPS*ABS(A(I,I))` and `EPS*ABS(A(J,J))`, where `EPS` is machine precision.

Essential part of `KMMS`:

```
 ODD   = .TRUE.
*
  DO 10 ISWEEP=1,20
   IC=0
    DO 1 K=1,N
*      THE NEXT LOOP CAN BE PERFORMED IN PARALLEL
     DO 2 I=K,(N+K-1)/2
      CALL KOGSTEP(LDA,N,I,N+K-I,A,VL,U,VR,V,IC)
  2  CONTINUE
*      THE NEXT LOOP CAN BE PERFORMED IN PARALLEL
     DO 3 I=1,(K-1)/2
      CALL KOGSTEP(LDA,N,I,K-I,A,VL,U,VR,V,IC)
  3  CONTINUE
*
  1 CONTINUE
*
     IF(IC .EQ. 0)  RETURN
     ODD=.NOT.ODD
 10 CONTINUE
```

Essential part of `KOGSTEP`:

```
  IF(A(I,J).NE.ZERO)THEN
   IC=IC+1
   CALL DLASV2(A(I,I),A(I,J),A(J,J),
  1          SMIN,SMAX,SNR,CSR,SNL,CSL)
   CALL DROT(J-I-1,A(I,I+1),LDA,A(I+1,J),1,CSR,SNR)
   CALL DROT(I-1,A(1,I),1,A(1,J),1,CSL,SNL)
   CALL DROT(N-J,A(I,J+1),LDA,A(J,J+1),LDA,CSL,SNL)
   A(I,I)=SMAX
   A(J,J)=SMIN
   A(I,J)=ZERO
   IF(ODD)THEN
C           UPPER-TRIANGULAR PIVOT SUBMATRIX
    IF(VL) CALL DROT(N,U(1,I),1,U(1,J),1,CSL,SNL)
    IF(VR) CALL DROT(N,V(1,I),1,V(1,J),1,CSR,SNR)
   ELSE
C         LOWER-TRIANGULAR PIVOT MATRIX
    IF(VL) CALL DROT(N,U(1,I),1,U(1,J),1,CSR,SNR)
    IF(VR) CALL DROT(N,V(1,I),1,V(1,J),1,CSL,SNL)
   ENDIF
  ENDIF
```

Some global and quadratic convergence convergence considerations of this method are presented in [30]. It can be shown that a hybrid method, combining the KMMS with the Hestenes one-sided Jacobi method, proposed in [6], can be adapted in the same way to work with butterfly matrices (see [30].

2.6 Parallel algorithm

Before we propose a way how to parallelize the Kogbetliantz algorithm, let us note that a slowdown can be expected in executing the commands which access elements by rows. These are: `CALL DROT(J-I-1,A(I,I+1),LDA,A(I+1,J),1,CSR,SNR)` and `CALL DROT(N-J,A(I,J+1),LDA,A(J,J+1),LDA,CSL,SNL)` because the stride `LDA` is not unit. A way how to remove this bottleneck is to transform the rows into columns, prior to the transformation. Since the algorithm applies in each parallel step around $n/2$ left rotations, this fast transpose should be considered as a negligible cost.

To explain this idea, we return to the butterfly-like form of $B^{[t]}$ from page 7. Since $B^{[t]}$ is ET, an interchange of the elements $b_{ij}^{[t]} \neq 0$ and $b_{ji}^{[t]} = 0$ reduces to copying $b_{ij}^{[t]} \mapsto b_{ji}^{[t]}$ and copying $0 \mapsto b_{ij}^{[t]}$. Suppose we have at disposal a fast, parallel routine `transpose(t,A)` which can make a fast transpose of $B^{[t]}$ at any time t (provided `A` represents $B^{[t]}$). In describing the parallel algorithm, we shall use a Pascal-like meta language and the following notation:

$$
\begin{aligned}
\mathtt{S(t)} &= \left\{1,\dots\left[\frac{t-1}{2}\right]\right\} \cup \left\{t,\dots\left[\frac{n+t-1}{2}\right]\right\}, \quad 1 \le t \le n \quad ([z] \text{ is the largest integer} \le z),\\
\mathtt{U(t)} &= \{U_{ij}^{[t]} \,:\, (i,j) \in \mathtt{S(t)}\}, \qquad \mathtt{V(t)} = \{V_{ij}^{[t]} \,:\, (i,j) \in \mathtt{S(t)}\},\\
\mathtt{VECL} &= \text{array, matrix of left singular vectors,} \qquad \mathtt{VECR} = \text{array, matrix of right singular vectors,}\\
\mathtt{left} &= \text{logical variable, true if left rotations are accumulated in VECL,}\\
\mathtt{right} &= \text{logical variable, true if right rotations are accumulated in VECR,}
\end{aligned}
$$

We recall, even if both singular vector matrices are wanted, only one will be computed. Because the other one can be cheaply and safely computed a posteriori. So, at most one of the matrices `VECL` and `VECR` has to be computed via iterations. An outline of the parallel algorithm follows:

```
repeat
for t=1:n do
 compute concurrently parameters of all rotations from U(t) and V(t)
  if (right) apply concurrently all right rotations to VECR
 apply concurrently all right rotations on appropriate parts of columns of A
 transpose(t,A)
  if (left) apply concurrently all left rotations to VECL
 apply all left rotations on appropriate parts of columns of A
 transpose(t,A)
end{t}
until convergence
```

Such an algorithm can be implemented on computer architectures with shared memory. It can be enhanced if the data structures are suitably chosen. Say, if `VECL` (`VECR`) and `A` are contained in the same array, then left (right) rotations can be used more efficiently (less calls to `DROT` routine). However, the algorithm above presumes that at least $n/2$ processors are available. For larger matrices this will not be an option. To remove this shortcoming, we can try to work with blocks instead of elements alone.

3 Blocking the Algorithm

Working with blocks means making algorithms more efficient. They become BLAS 3 algorithms which can better use the computing resources of contemporary processors. This approach to parallelizing Kogbetliantz method is more promising (see [11]) because it can adapt to the number of available processors and to their inherent characteristics like internal memory on different hierarchy levels.

The derivation of the block version of the modulus Kogbetliantz method is related to ideas described in [18, 19].

We can start again with the triangular T which is obtained after one or two QR factorizations of the initial matrix. We shall use the following partition of T and we shall associate it with all matrices which will be obtained from T.

$$T = \begin{bmatrix} T_{11} & T_{12} & \cdots & T_{1m} \\ 0 & T_{22} & \cdots & T_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & T_{mm} \end{bmatrix}$$

Here, each diagonal block T_{ii} is of order $n_i \geq 1$, so that $\mathcal{M} = (n_1, n_2, \ldots, n_m)$ makes a partition of n. The choice of $\mathcal{M}$ depends on computing resources, especially on the available fast (cache) memory of each processor. To make exposition simpler, we shall assume $n_1 = n_2 = \cdots = n_m = n/m$.

To reduce T to "block-butterfly form" we can use permutational similarity quite similar to (1.1). However, we have to assume that I_{ij} is product of simple transpositions. The effect of $I_{ij}^T T$ is to swap block-rows i and j and of $T\, I_{ij}$ is to swap block-columns i and j. So, the left-hand permutations in (1.1) have to be written transposed.

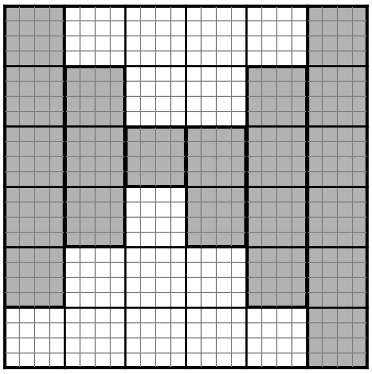
3.1 Parallel step zero

Let $B^{[0]} = B$ be a matrix in block-butterfly form. Before commencing the iteration, we shall need a simple preparation of the starting matrix, which we call *parallel step zero*. It consists of annihilating the blocks $B_{1m}^{[0]}, B_{2,m-1}^{[0]}, \ldots$, and of diagonalizing all diagonal blocks. Using the notation from the relation (2.6), this initial step can be described as follows

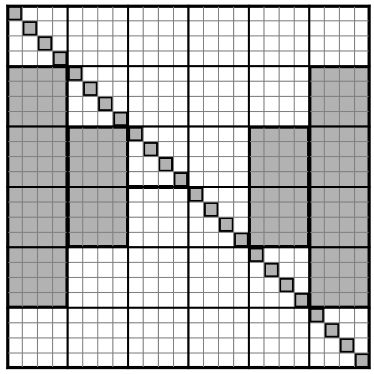
$$B^{[1]} = U^{[0]*} B^{[0]} V^{[0]}, \quad U^{[0]} = \prod_{(i,j)\in \mathbf{piv}(m)} \mathbf{U}_{ij}^{[0]}, \quad V^{[0]} = \prod_{(i,j)\in \mathbf{piv}(m)} \mathbf{V}_{ij}^{[0]},$$

where $\mathbf{piv}(m) = \{(1,m), (2,m-1)\ldots\}$ is the mth *pivot set* associated with the block-algorithm. The figure below indicates the changes in the matrix. The iterative part then generates the matrices $B^{[2]}$, $B^{[3]}, \ldots$ subsequently in a similar fashion.

To make the block-algorithm efficient (see [18]), we shall represent $B^{[t]}$ in factored form: $B^{[t]} = {E^{[t]}}^T C^{[t]} F^{[t]}$, where $E^{[t]}$ and $F^{[t]}$ are block-diagonal and orthogonal. In addition, we shall keep the diagonal of the current iterate $B^{[t]}$ separately in a vector $\gamma^{[t]}$.



Parallel
Step 0
$\mapsto$



$B^{[0]}$ in butterfly form

$B^{[1]}$ in butterfly form

Consequently, we require more outputs from the parallel step zero. In particular, its output should be: $C^{[1]}$, $E^{[1]}$, $F^{[1]}$ and $\gamma^{[1]}$. We use a block-column partition of some matrices. In particular, let $B^{[0]} = [B_1^{[0]}, B_2^{[0]}, \ldots, B_m^{[0]}]$ be the block-column partition of $B^{[0]}$. It follows an algorithmic description of the step described above.

Parallel step zero:

for $i = 1$ `to` div$(m, 2)$ do in parallel

set $j = m + 1 - i$

(a) Compute SVD: $\begin{bmatrix} B_{ii}^{[0]} & B_{ij}^{[0]} \\ 0 & B_{jj}^{[0]} \end{bmatrix} = \mathbf{U}_{ij}^{[0]} \,\mathbf{\Gamma}_i\, \mathbf{V}_{ij}^{[0]T}$ and CS decomposition of $\mathbf{U}_{ij}^{[0]}$ and $\mathbf{V}_{ij}^{[0]}$:

$$\mathbf{U}_{ij}^{[0]} = \begin{bmatrix} \dot{U}_{ii} & 0 \\ 0 & \dot{U}_{jj} \end{bmatrix} \Theta_{ij} \begin{bmatrix} \ddot{U}_{ii} & 0 \\ 0 & \ddot{U}_{jj} \end{bmatrix}, \quad \mathbf{V}_{ij}^{[0]} = \begin{bmatrix} \dot{V}_{ii} & 0 \\ 0 & \dot{V}_{jj} \end{bmatrix} \Phi_{ij} \begin{bmatrix} \ddot{V}_{ii} & 0 \\ 0 & \ddot{V}_{jj} \end{bmatrix}$$

(b) Apply: $B_i' = B_i \dot{V}_{ii}, \quad B_j' = B_j \dot{V}_{jj}$ and afterwards: $[B_i'', B_j''] = [B_i', B_j']\Phi_{ij}$

(c) Transpose: $\bar{B} = B''^T$ (let $\bar{B} = [\bar{B}_1, \bar{B}_2, \ldots, \bar{B}_m]$ be the block-column partition of $\bar{B}$)

(d) Apply: $\bar{B}_i' = \bar{B}_i \dot{U}_{ii}, \quad \bar{B}_j' = \bar{B}_j \dot{U}_{jj}$ and afterwards: $[\bar{B}_i'', \bar{B}_j''] = [\bar{B}_i', \bar{B}_j']\Theta_{ij}$

(f) Transpose: $C^{[1]} = B''^T$

(g) Copy: $E_{ii}^{[1]} = \ddot{U}_{11}, \quad E_{jj}^{[1]} = \ddot{U}_{jj}, \quad F_{ii}^{[1]} = \ddot{V}_{11}, \quad F_{jj}^{[1]} = \ddot{V}_{jj}$
Copy the first n_i and last n_j diagonal elements of $\mathbf{\Gamma}_i$ into the appropriate parts of the vector $\gamma^{[1]}$.

(h) If (*left*) form: $UU^{[1]} = E^{[1]T}$
If (*right*) form: $VV^{[1]} = F^{[1]T}$

end for

The command "transpose" presumes that each processor will do its partial job like transposing block columns and block-rows i and j. It is understood that block-partition takes into account the cache memory capacity, i.e., that all small matrices (those with two subscripts) can be contained in the internal memory of each processor, which is several times faster than the main (shared) memory.

3.2 Iteration

Now, let us consider the iterative part of the algorithm. The block-Kogbetliantz method defined by the block-modulus strategy, when applied to $B^{[1]}$, which is in block-butterfly form, behaves similarly as the simple modulus algorithm when applied to matrix in butterfly form. The essential part of the algorithm, which achieves this harmony is the annihilation of the off-block-diagonal pivot submatrices $B_{ij}^{[t]}$, $(i, j) \in \mathbf{piv}(t)$. In addition, the diagonal blocks $B_{ii}^{[t]}$, $B_{jj}^{[t]} \in \mathbf{piv}(t)$ are diagonalized. So, after each parallel step, the off-norm of the current matrix is reduced:

$$\|\Omega(B^{[t+1]})\|^2 = \|\Omega(B^{[t]})\|^2 - \sum_{(i,j)\in\mathbf{piv}(t)} \|B_{ij}^{[t]}\|^2 \,.$$

Since $B^{[t]}$ is kept factored as ${E^{[t]}}^T C^{[t]} F^{[t]}$, we have to derive how $C^{[t]}$, $E^{[t]}$, $F^{[t]}$ and the vector $\gamma^{[t]}$ are updated.

In its raw form, the parallel block-Kogbetliantz algorithm takes the form (2.6), where $U_{ij}^{[t]}$ and $V_{ij}^{[t]}$ are "block rotations" whose essential parts are $(n_i + n_j) \times (n_i + n_j)$ orthogonal matrices $\mathbf{U}_{ij}^{[t]}$ and $\mathbf{V}_{ij}^{[t]}$ which satisfy

$$\mathbf{B}_{ij}^{[t]} = \begin{bmatrix} B_{ii}^{[t]} & B_{ij}^{[t]} \\ 0 & B_{jj}^{[t]} \end{bmatrix} = \mathbf{U}_{ij}^{[t]}\, \mathbf{\Gamma}_i\, {\mathbf{V}_{ij}^{[t]}}^T, \qquad \mathbf{\Gamma}_i \text{ diagonal}. \tag{3.1}$$

We assume that the diagonal blocks $B_{ii}^{[t]}$ and $B_{jj}^{[t]}$ are diagonal. This is certainly true for the initial matrix $B^{[1]}$. Since $B^{[t]}$ is given in factored form, we first have to see how to compute $\mathbf{B}_{ij}^{[t]}$. We assume that $\mathbf{B}_{ij}$ is upper-triangular,

$$\begin{bmatrix} B_{ii}^{[t]} & B_{ij}^{[t]} \\ 0 & B_{jj}^{[t]} \end{bmatrix} = \begin{bmatrix} E_{ii}^{[t]} & 0 \\ 0 & E_{jj}^{[t]} \end{bmatrix}^T \begin{bmatrix} C_{ii}^{[t]} & C_{ij}^{[t]} \\ 0 & C_{jj}^{[t]} \end{bmatrix} \begin{bmatrix} F_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \end{bmatrix} = \begin{bmatrix} {E_{ii}^{[t]}}^T C_{ii}^{[t]} F_{ii}^{[t]} & {E_{ii}^{[t]}}^T C_{ij}^{[t]} F_{jj}^{[t]} \\ 0 & {E_{jj}^{[t]}}^T C_{jj}^{[t]} F_{jj}^{[t]} \end{bmatrix}.$$

Since, $B_{ii}^{[t]}$ and $B_{jj}^{[t]}$ are diagonal, we can first copy zeros in this part of $\mathbf{B}_{ij}^{[t]}$ and then copy diagonal elements from the vector $\gamma^{[t]}$ onto the diagonal of $B_{ii}^{[t]}$ and $B_{jj}^{[t]}$. After that, we have to compute ${E_{ii}^{[t]}}^T C_{ij}^{[t]} F_{jj}^{[t]}$. This should be computed within the processor which is associated with the pair $(i, j) \in \mathbf{piv}(t)$. The matrix multiplications can be computed using an appropriate fast routine, say the BLAS 3 routine `DGEMM`. If $\mathbf{B}_{ij}$ is lower-triangular, then $B_{ji}^{[t]} = {E_{jj}^{[t]}}^T C_{ji}^{[t]} F_{ii}^{[t]}$, but the latter procedure is the same.

Next, the SVD of $\mathbf{B}_{ij}^{[t]}$ is computed as indicated in (3.1). Here, one can choose among several fast and accurate methods, say the one-sided Jacobi or the Kogbetliantz (serial or modulus) method. After that, CS decompositions of $\mathbf{U}_{ij}^{[t]}$ and $\mathbf{V}_{ij}^{[t]}$ are computed (see [19]). We shall write it in the form

$$\mathbf{U}_{ij}^{[t]} = \begin{bmatrix} \dot{U}_{ii}^{[t]} & 0 \\ 0 & \dot{U}_{jj}^{[t]} \end{bmatrix} \Theta_{ij}^{[t]} \begin{bmatrix} \ddot{U}_{ii}^{[t]} & 0 \\ 0 & \ddot{U}_{jj}^{[t]} \end{bmatrix}, \quad \mathbf{V}_{ij}^{[t]} = \begin{bmatrix} \dot{V}_{ii}^{[t]} & 0 \\ 0 & \dot{V}_{jj}^{[t]} \end{bmatrix} \Phi_{ij}^{[t]} \begin{bmatrix} \ddot{V}_{ii}^{[t]} & 0 \\ 0 & \ddot{V}_{jj}^{[t]} \end{bmatrix}.$$

Here $\Theta_{ij}^{[t]}$ and $\Phi_{ij}^{[t]}$ are orthogonal matrices, actually, products of at most $\min\{n_i, n_j\}$ commuting plane rotations.

Next, $B^{[t+1]}$ is computed as ${U^{[t]}}^T (B^{[t]} V^{[t]})$. For this purpose, let us first consider the transformation $B^{[t]} \mapsto B^{[t]} V^{[t]}$. It is useful to introduce $J_{ij} = [J_i, J_j]$ where $I_n = [J_1, \ldots, J_m]$ is the block-column partition of the identity matrix. Then $B^{[t]} J_{ij} = [B_i^{[t]}, B_j^{[t]}]$. If we post-multiply the equation $B^{[t+1]} = {U^{[t]}}^T (B^{[t]} V^{[t]})$, or better ${E^{[t+1]}}^T C^{[t+1]} F^{[t+1]} = {U^{[t]}}^T ({E^{[t]}}^T C^{[t]} F^{[t]}) V^{[t]}$, by J_{ij}, we obtain

$$\begin{aligned} {E^{[t+1]}}^T [C_i^{[t+1]}, C_j^{[t+1]}] \begin{bmatrix} F_{ii}^{[t+1]} & 0 \\ 0 & F_{jj}^{[t+1]} \end{bmatrix} &= {U^{[t]}}^T {E^{[t]}}^T [C_i^{[t]}, C_j^{[t]}] \begin{bmatrix} F_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \end{bmatrix} \mathbf{V}_{ij}^{[t]} \\ &= {U^{[t]}}^T {E^{[t]}}^T [C_i^{[t]}, C_j^{[t]}] \left(\left(\begin{bmatrix} F_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \end{bmatrix} \begin{bmatrix} \dot{V}_{ii}^{[t]} & 0 \\ 0 & \dot{V}_{jj}^{[t]} \end{bmatrix} \right) \Phi_{ij}^{[t]} \right) \begin{bmatrix} \ddot{V}_{ii}^{[t]} & 0 \\ 0 & \ddot{V}_{jj}^{[t]} \end{bmatrix}. \end{aligned}$$

Hence, we can make the following updates:

$$F_{ii}^{[t+1]} = \ddot{V}_{ii}^{[t]}, \quad F_{jj}^{[t+1]} = \ddot{V}_{jj}^{[t]}, \quad [\bar{C}_i^{[t]}, \bar{C}_j^{[t]}] = [C_i^{[t]}, C_j^{[t]}] \begin{bmatrix} F_{ii}^{[t]} \dot{V}_{ii}^{[t]} & 0 \\ 0 & F_{jj}^{[t]} \dot{V}_{jj}^{[t]} \end{bmatrix} \Phi_{ij}^{[t]}.$$

These updates can be performed concurrently for all $(i, j) \in \mathbf{piv}(t)$. This results in the matrix $F^{[t+1]}$ and in auxiliary matrix $\bar{C}$. Now, let us consider the left transformation. Note that $J_i^T X$ is the ith block-row of X. Pre-multiplying the equation ${E^{[t+1]}}^T C^{[t+1]} = {U^{[t]}}^T {E^{[t]}}^T \bar{C}^{[t]}$ by J_{ij}^T, we obtain

$$\begin{aligned} \begin{bmatrix} E_{ii}^{[t+1]} & 0 \\ 0 & E_{jj}^{[t+1]} \end{bmatrix}^T \begin{bmatrix} J_i^T C^{[t+1]} \\ J_j^T C^{[t+1]} \end{bmatrix} &= {\mathbf{U}_{ij}^{[t]}}^T \begin{bmatrix} E_{ii}^{[t]} & 0 \\ 0 & E_{jj}^{[t]} \end{bmatrix}^T \begin{bmatrix} J_i^T \bar{C}^{[t]} \\ J_j^T \bar{C}^{[t]} \end{bmatrix} \\ &= \begin{bmatrix} \ddot{U}_{ii}^{[t]} & 0 \\ 0 & \ddot{U}_{jj}^{[t]} \end{bmatrix}^T \left({\Theta_{ij}^{[t]}}^T \begin{bmatrix} \dot{U}_{ii}^{[t]} & 0 \\ 0 & \dot{U}_{ii}^{[t]} \end{bmatrix}^T \begin{bmatrix} E_{ii}^{[t]} & 0 \\ 0 & E_{ii}^{[t]} \end{bmatrix}^T \begin{bmatrix} J_i^T \bar{C}^{[t]} \\ J_j^T \bar{C}^{[t]} \end{bmatrix} \right). \end{aligned}$$

Hence, we can make the following updates

$$E_{ii}^{[t+1]} = \ddot{U}_{ii}^{[t]}, \quad E_{jj}^{[t+1]} = \ddot{U}_{jj}^{[t]}, \quad [\tilde{C}_i^{[t+1]}, \tilde{C}_j^{[t+1]}] = [\tilde{\bar{C}}_i^{[t]}, \tilde{\bar{C}}_j^{[t]}] \begin{bmatrix} E_{ii}^{[t]} \dot{U}_{ii}^{[t]} & 0 \\ 0 & E_{jj}^{[t]} \dot{U}_{jj}^{[t]} \end{bmatrix} \Theta_{ij}^{[t]}.$$

where $\tilde{\bar{C}}^{[t]} = \bar{C}^{[t]\,T}$ and $\tilde{C}^{[t+1]} = C^{[t+1]\,T}$. Again these updates can be performed concurrently, for all $(i,j) \in \mathbf{piv}(t)$. Hence, after computing $\bar{C}$, we have to transpose it. Then we update $\bar{C}^T$ from the right side and $C^{[t+1]^T}$ is obtained. Finally, we have to copy the diagonal elements of Γ_i to the appropriate places in $\gamma^{[t+1]}$.

After all these preparations, we can write down the iterative part of the algorithm. We shall use an $n \times n$ array A and two $n \times nb$ arrays E and F for the matrices $E^{[t]}$ and $F^{[t]}$. Here, nb is not smaller than $\max_i n_i$. We shall use the notation E_i for $E_{ii}^{[t]}$ etc. Block columns of A are denoted by $A_1, \ldots A_m$ and A_{ij} is the (i,j)th block of A. The vector g will play the role of $\gamma^{[t]}$ and several smaller $2nb \times 2nb$ arrays B, U, V will be used. To make the algorithm simpler, we use some additional, even smaller arrays, although they are not needed (one can use parts of the larger, already mentioned arrays). The matrices of singular vectors can be updated in the arrays VECL or VECR, depending on the values of the logical variables *left* or *right*.

Parallel Block-Kogbetliantz:

repeat

for t=1:m do
  for $(i,j) \in \mathbf{piv}(t)$ do in parallel
    (a1) Compute: $B_{12} = E_i^T A_{ij} F_j$
    (a1) Copy: from g to $\mathrm{diag}(B_{11})$ and $\mathrm{diag}(B_{22})$ and form: $B = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}$ B_{11} is diagonal, B_{22} is diagonal
    (b1) Compute SVD: $B = U\,\Gamma\,V^T$
    (b2) Update: g from Γ
    (b3) Compute CS decomposition: $U = \begin{bmatrix} U1 & 0 \\ 0 & U2 \end{bmatrix} H \begin{bmatrix} U3 & 0 \\ 0 & U4 \end{bmatrix}$
    (b4) Compute CS decomposition: $V = \begin{bmatrix} V1 & 0 \\ 0 & V2 \end{bmatrix} K \begin{bmatrix} V3 & 0 \\ 0 & V4 \end{bmatrix}$
    (c1) Compute: $X = F_i\,V1, \quad Y = F_j\,V2$
    (c2) Update: $A_i \leftarrow A_i\,X, \quad A_j \leftarrow A_j\,Y;$
    If (*right*) update: $\mathrm{VECR}_i \leftarrow \mathrm{VECR}_i\,X, \quad \mathrm{VECR}_j \leftarrow \mathrm{VECR}_j\,Y$
    (c3) Update: $[A_i, A_j] \leftarrow [A_i, A_j]K;$ if (*right*) update: $[\mathrm{VECR}_i, \mathrm{VECR}_j] \leftarrow [\mathrm{VECR}_i, \mathrm{VECR}_j]K$
    (c4) Update: $F_i \leftarrow V3, \quad F_j \leftarrow V4$
    (d1) Transpose: $A \leftarrow A^T$
    (e1) Compute: $X = E_i\,U1, \quad Y = E_j\,U2$
    (e2) Update: $A_i \leftarrow A_i X, \quad A_j \leftarrow A_j Y;$
    If (*left*) update: $\mathrm{VECL}_i \leftarrow \mathrm{VECL}_i\,X, \quad \mathrm{VECL}_j \leftarrow \mathrm{VECL}_j\,Y$
    (e3) Update: $[A_i, A_j] \leftarrow [A_i, A_j]H;$ If (*left*) update: $[\mathrm{VECL}_i, \mathrm{VECL}_j] \leftarrow [\mathrm{VECL}_i, \mathrm{VECL}_j]K$
    (e4) Update: $E_i \leftarrow U3, \quad E_j \leftarrow U4$
    (d2) Transpose: $A \leftarrow A^T$
  end for
end for

until convergence

We note that the parallel step zero can easily be rewritten as algorithm with the above notation.

It is important that all small arrays like B, U, V can be stored in the fast memory. Then the bookkeeping of these matrices together with all needed factorizations (SVD, CS) is performed at low extra cost in computational time. The gain, which lies in updating the block-columns (which is the slowest part of the algorithm), justifies all this extra work. The above modification of the original algorithm can be seen as the block version of the fast-scaled rotations (see [18, 19]). In contrast to the fast-scaled rotations, there is no danger of potential growth of the elements since the transformation is always done with orthogonal matrices. However, for such purposes the existing algorithms for computing the CS decomposition are not appropriate and new algorithms are needed.

A proper testing of this algorithm is needed. The purpose of this paper was to give evidence that there is a good potential for parallelizing Kogbetliantz method in this way.

Acknowledgements. The authors are thankful to D. Kressner for reading and improving the paper.

References

[1] Bečka M. and Okša G.: On Variable Blocking Factor in a Parallel Dynamic Block: Jacobi SVD Algorithm. Parallel Computing 29 Issue 9 (2003) 1153 - 1174.

[2] Bečka M., Okša G., Vajteršic M.: Dynamic Ordering for a Parallel Block-Jacobi SVD algorithm Parallel Computing 28 Issue 2 (2002) 243 - 262.

[3] Charlier, J.P. and Van Dooren, P.: On Kogbetliantz's SVD Algorithm in the Presence of Clusters. Linear Algebra and Its Applications 95 (1987) 135–160.

[4] Charlier J.P., Vanbegin M. and Van Dooren P.: On Efficient Implementations of Kogbetliantz's Algorithm for Computing the Singular Value Decomposition. Numerische Mathematik 52 (1988) 279-300.

[5] Demmel J. and Veselić K.: Jacobi's method is more accurate than QR. SIAM J. Matrix Anal. Appl. 13, 1204–1245 (1992).

[6] Drmač Z.: Computing the singular and the generalized singular values. Ph.D. University of Hagen, Germany 1994.

[7] Drmač Z.: A posteriori computation of the singular vectors in a preconditioned Jacobi SVD algorithm. IMA J. Numer. Anal. 19 (1999) 191-213.

[8] Drmač Z. and Veselić K.: New fast and accurate Jacobi SVD algorithm I. Preprint 2005, (LAPACK Working note 169).

[9] Drmač Z. and Veselić K.: New fast and accurate Jacobi SVD algorithm II. Preprint 2005, (LAPACK Working note 170).

[10] Fernando K. V.: Linear Convergence of the Row Cyclic Jacobi and Kogbetliantz Methods. Numer. Math. 56 (1989) 71-91.

[11] Fernando K. V. and Hammarling S.: On Block Kogbetliantz Methods for Computation of the SVD. SVD and signal processing: algorithms, applications and architectures, 349 - 355, 1989, ISBN:0-444-70439-6.

[12] Forsythe G. E. and Henrici P.: The Cyclic Jacobi Method for Computing the Principle Values of a Complex Matrix. Trans. Amer. Math. Soc. 94 (1960) 1–23.

[13] Götze J.: On the Parallel Implementation of Jacobi and Kogbetliantz Algorithms. SIAM J. SC 15, Issue 6 (1994) 1331 - 1348.

[14] Hari V.: On the Quadratic Convergence of Jacobi Algorithms. Radovi Mathematički 2 (1986) 127–146.

[15] Hari V. and Veselić K.: On Jacobi methods for singular value decompositions. SIAM J. Sci. Stat. Comput. Vol. 8, No. 5 (1987) 741–754.

[16] Hari V.: On Sharp Quadratic Convergence Bounds for the Serial Jacobi Methods. Numer. Math. 60 (1991) 375–406.

[17] Hari V. and Matejaš J.: Quadratic Convergence of Scaled Iterates by Kogbetliantz Method. Computing [Suppl] 16 (2003) 83-105.

[18] Hari V.: On Some New Applications of the CS Decomposition. ICNAAM 2004: Extended Abstracts (Editor T. E. Simos and Ch. Tsitouras) ISBN: 3-527-40563-1, Hardcover.

[19] Hari V.: Accelerating the SVD Block-Jacobi Method. Computing 75, Issue 1 (2005) 27–53.

[20] Hari V. and Matejaš J.: Accuracy of the Kogbetliantz method. Preprint, University of Zagreb, 2005.

[21] Kogbetliantz E.: Diagonalization of General Complex Matrices as a New Method for Solution of Linear Equations, Proc. Imtern. Congr. Math. Amsterdam 2 (1954) 356–357.

[22] Kogbetliantz E.: Solutions of Linear Equations by Diagonalization of Coefficient Matrices, Quart. Appl. Math. 13 (1955) 123–132 .

[23] Londre T. and Rhee N. H.: Numerical Stability of the Parallel Jacobi Method. SIAM J. MAA. Vol. 26, No. 4 (2005) 985–1000.

[24] Luk F. T.: A Triangular Processor Array for Computing Singular Values. Linear Algebra and Its Applications 77 (1986) 259–273.

[25] A Proof of Convergence for Two Parallel Jacobi SVD Algorithms. IEEE Transactions on Computers. Vol. 38 , Issue 6 (1989) 806 - 811

[26] Luk F. T. and Park H.: On parallel Jacobi orderings. SIAM J. Sci. Statist. Comput. 10 (1989) 18–26.

[27] Matejaš J. and V. Hari: Scaled Iterates by Kogbetliantz Method, Proceedings of the 1st Conference on Applied Mathematics and Computations. Dubrovnik, Croatia, September 13–18, 1999. Publisher Dept. of Mathematics, University of Zagreb, 2001. pp. 1–20.

[28] Okša G. and Vajteršic M.: Systolic Block-Jacobi SVD Algorithm for Processor Meshes. Highly parallel computaions: algorithms and applications, 211-235 WIT Press 2001.

[29] Paige C. C. and Van Dooren P.: On the Quadratic Convergence of Kogbetliantz's Algorithm for Computing the Singular Value Decomposition. Linear Algebra and Its Applications 77 (1986) 301–313.

[30] Zadelj-Martić V.: Diagonalization Methods for Butterfly Matrices. MS thesis, University of Zagreb, 1999 (Croatian language).