Saving Flops in LU Based Shift-and-Invert Strategy

Laura Grigori, Desire W. Nuentsa, Hua Xiang

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Saving Flops in LU Based Shift-and-Invert Strategy

Laura Grigori — Desire W. Nuentsa — Hua Xiang

N° 6553

Juin 2008

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6553--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Saving Flops in LU Based Shift-and-Invert Strategy

Laura Grigori* , Desire W. Nuentsa† , Hua Xiang‡

Thème NUM — Systèmes numériques
Équipe-Projet Grand-large

Rapport de recherche n° 6553 — Juin 2008 — 15 pages

Abstract: The shift-and-invert method is very efficient in eigenvalue computations, in particular when interior eigenvalues are sought. This method involves solving linear systems of the form $(A - \sigma I)z = b$. The shift σ is variable, hence when a direct method is used to solve the linear system, the LU factorization of $(A - \sigma I)$ needs to be computed for every shift change. We present two strategies that reduce the number of floating point operations performed in the LU factorization when the shift changes. Both methods perform first a preprocessing step that aims at eliminating parts of the matrix that are not affected by the diagonal change. This leads to 43% and 50% flops savings respectively.

Key-words: shift-and-invert, eigenvalue, divide and conquer, LU factorization

* INRIA Saclay-Ile de France, Laboratoire de Recherche en Informatique, Bât 490 Université Paris-Sud 11, 91405 Orsay Cedex France. (laura.grigori@inria.fr)
† Departement d'Informatique, Université de Yaoundé I, B.P. 812 Yaoundé Cameroun. (nuentsa@gmail.com)
‡ INRIA Saclay-Ile de France, Laboratoire de Recherche en Informatique, Bât 490 Université Paris-Sud 11, 91405 Orsay Cedex France. (hua.xiang@inria.fr)

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Préconditionnements basés sur une approximation du produit de Kronecker pour les problèmes modèles de convection-diffusion

Résumé : La méthode de la puissance inverse avec décalage est très efficace dans le calcul des valeurs propres, en particulier pour les valeurs propres intérieures. Cette méthode nécessite la résolution des systèmes linéaires de la forme $(A-\sigma I)z = b$. La valeur de σ est variable, ainsi quand une méthode directe est utilisée pour résoudre le système linéaire, la factorisation LU de $(A - \sigma I)$ doit être calculée pour chaque changement de σ. Nous présentons deux techniques pour réduire le nombre d'opérations flottantes effectuées pendant la factorisation LU lors du changement de σ. Les deux techniques effectuent d'abbord une étape de pretraitement pour éliminé des parties de la matrice qui ne sont pas affectées par le changement des éléments sur la diagonale. Ceci conduit à une réduction de 43% et 50% respectivement du nombre d'opérations flottantes.

Mots-clés : la méthode de la puissance inverse avec décalage, calcul de valeurs propres, factorisation LU

1 Introduction

The standard eigenvalue problem,

$$Ax = \lambda x, \tag{1}$$

where $x \neq 0, \lambda \in \mathbb{C}$, has broad applications in mechanics, physics, chemistry and economics, such as computing the energy level of atoms, vibration, flow instability, or principal components. The eigenvalue computation is still a very challenging problem, even if there are many practical methods and software available [1]. A basic method in eigenvalue computation is the power method. Although simple, many methods are rooted in it. One enhancement of the power method is the inverse iteration, which applies the power method to $(A - \sigma I)^{-1}$, where σ is a shift. This method can converge to any desired eigenvalue, especially when a few interior eigenvalues are sought. The iteration step can be expressed as

$$v_k = \frac{1}{\alpha_k}(A - \sigma I)^{-1} v_{k-1} \quad (k = 1, 2, \cdots), \tag{2}$$

where v_0 is the initial guess. The most expensive step in this computation is to find the solution of a linear system of the form

$$(A - \sigma I)z = b. \tag{3}$$

Note that $A - \sigma I$ is ill-conditioned when σ is close to the true eigenvalue. But most of the inaccuracies of the solution are in the direction of the eigenvector being approximated [8, 10]. If we have a better approximation of an eigenvalue, we can change the shift occasionally. Because α_k in (2) converges to $1/(\lambda_j - \sigma)$, it is natural to take $\sigma_{\text{new}} = \sigma_{\text{old}} + \frac{1}{\alpha_k}$ [13]. If the Rayleigh quotient is used as the shift, then this method is called Rayleigh quotient iteration (RQI). For non-Hermitian matrices, the generized Rayleigh quotient is used, $\sigma_j = y_j^* A x_j / y_j^* x_j$, where x and y are the approximate left eigenvector and right eigenvector respectively in j-th step [23]. RQI also appears in QR algorithm in a disguised form [19]. QR is the standard algorithm for computing the full spectrum of a dense matrix. But it is not appropriate for large sparse matrices, because it uses othorgonal transformations and it destroies the sparsity of the input matrix.

For large sparse eigenvalue problem, Krylov subspace methods are generally used, such as implicitly restarted Arnoldi (IRA) [17], or Bi-side Lanczos with look-ahead strategy [9]. Krylov subspace methods are good at computing the eigenvalues on the periphery of the spectrum [8, 13, 19, 22]. Usually these exterior eigenvalues are well-approximated first, and the interior eigenvalues follow much later. If we need the interior eigenvalues, spectral transformatin like shift-and-invert $(A - \sigma I)^{-1}$ is needed to find interior eigenvalues close to σ. For example, the Alfven spectrum is an interior part of the spectrum, without shift-and-invert it is almost impossible to compute this part with Krylov subspace methods [21]. When we apply the Arnoldi or Lanczos method to $(A - \sigma I)^{-1}$, we must solve a sequence of linear equations accurately in order to capture the desired eigenvalues. The shift-and-invert strategy is often used implicitly in Krylov subspace method. For example, the harmonic Rayleigh-Ritz procedure is related to the shift-and-invert $(A - \sigma I)^{-1}$, but it avoids the matrix inversion by its clever formulation to a projected generalized eigenvalue problem. Linear systems similar to (3) appear implicitly in the Jacobi-Davidson method [2, 3, 6, 16, 25]. This method expands the current subspace by computing an approximate solution t to a so-called correction equation, which is equivalent to $t = -u + \alpha(A - \sigma I)^{-1} u$, where α is chosen such that $t \perp u$ [16]. The shift-and-invert strategy also appears in the rational Krylov subspace method [11, 12] and the truncated RQ iteration [18, 24]

Hence, many methods are related to shift-and-invert, and this strategy is very efficient. The potential drawback is that linear systems as (3) need to be solved, which is the most expensive step of the strategy. The accuracy of the linear solver must be in accordance with the convergence tolerance of the eigensolver [5, 7, 15]. Otherwise, loss of accuracy in solving (3) may result in the corruption of the Krylov subspace. Since these linear systems are ill-conditioned, iterative methods will converge slowly. Usually an efficient preconditioner is not easy to find for these systems. Hence the direct methods are used in general to solve the system (3). In this paper, we focus on the shift-and-invert method with variable shifts as used in the dense standard eigenvalue problem. Traditionally, when the shift σ_j changes, the LU factorization needs to be performed again. We develop two strategies which consist in performing a pre-processing step such that the LU factorization is not computed from scratch when the shift changes. The pre-processing step annihilates some parts of the matrix A which are not influenced by the change of the diagonal elements. The first strategy is a divide and conquer strategy. We use a recursive 2×2 partition and symmetric permutation, and factorize the original matrix into a staircase shape. For each shift change, the factorization starts from this shape. In this process BLAS-3 operations can be used to achieve high performance, and 43% flops can be saved for each shift change. For the second strategy, we use two row permutations and one column permutation during each column elimination to control the position of original diagonal elements, such that their influence during updating is confined in the right part of the matrix. This strategy leads to 50% flops savings. These two strategies are discussed in detail in Section 2 and Section 3 respectively. We give numerical examples in Section 4 that check the numerical stability, the flops saving and the efficiency of the two strategies. We conclude in Section 5.

2 Strategy I: a divide and conquer approach

The classical Gaussian elimination with partial pivoting of the matrix $A \in \mathbb{R}^{n\times n}$ can be expressed as

$$L_{n-1}^{-1} P_{n-1} L_{n-2}^{-1} P_{n-2} \cdots L_1^{-1} P_1 A = U, \tag{4}$$

where U is an upper triangular matrix, $P_i (i = 1, \cdots, n-1)$ are permutation matrices, $L_i^{-1} (i = 1, \cdots, n-1)$ are Gauss transformation matrices computed as $L_i = I + l_i e_i^T$, and l_i is the Gauss vector [4]. The Gaussian elimination (4) can be rewritten as

$$L_{n-1}^{-1} \widehat{L}_{n-2}^{-1} \cdots \widehat{L}_1^{-1} PA = U, \tag{5}$$

where $\widehat{L}_i^{-1} = P_{n-1} \cdots P_{i+1} L_i^{-1} P_{i+1}^T \cdots P_{n-1}^T (i = 1, \cdots, n-2)$, $P = P_{n-1} P_{n-2} \cdots P_1$. For simplicity, we futher define $L = \widehat{L}_1 \widehat{L}_2 \cdots \widehat{L}_{n-2} L_{n-1}$, so we achieve the LU factorization $PA = LU$. Note that in a real implementation, L and U can be stored in place of the matrix A.

Clearly, if the diagonal elements of A change, this affects the whole factorization. That is the reason why $(A - \sigma I)$ needs to be factorized again when the shift σ changes. Our goal is to restrict the influence of the diagonal elements, and reuse some eliminations at the next step. We can achieve this goal by using a recursive 2×2 partition and symmetric permutations.

First we illustrate our approach on a simple case. Suppose that A is partitioned into 2×2 blocks. We denote it as $A = [A_1, A_3; A_2, A_4]$ (see Figure 1(a)). We can do some eliminations without modifying the diagonal elements. In the following we use three steps to explain the basic idea.

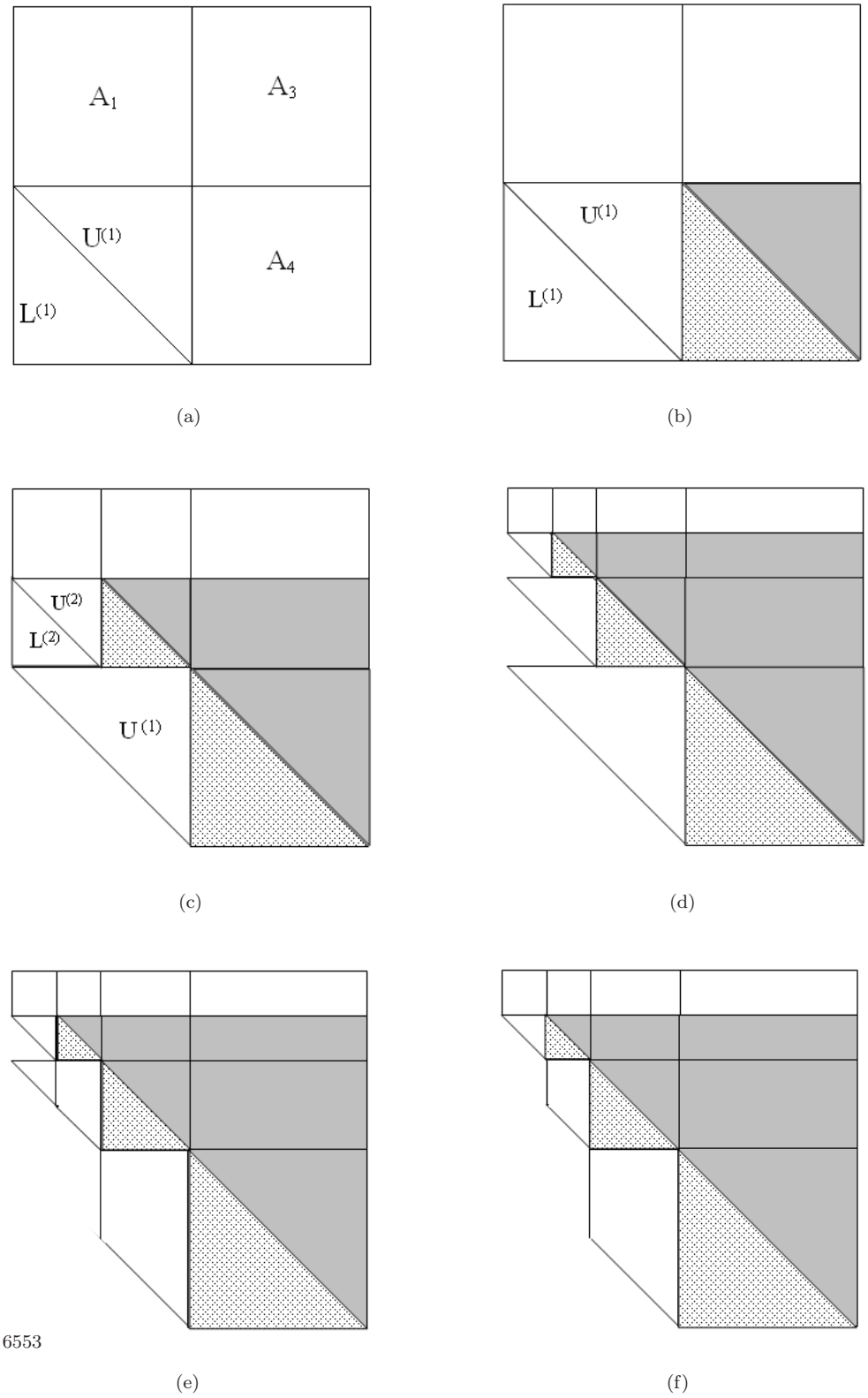

Figure 1: A recursive 2×2 partition strategy

Step 1. Local LU. We perform the LU factorization on (2,1) block. We formulate it as

$$PA_2 = L^{(1)}U^{(1)}, \tag{6}$$

where $L^{(1)}$ is stored in the lower part of A_2, which is shown in Figure 1(a).

Step 2. Symmetric permutation. In the previous step, the row permutations performed might change the position of the diagonal elements of A_4. If we apply symmetric permutation to the diagonal block A_4, that is, PA_4P^T, then the diagonal elements are still on the diagonal, though they are not necessarily in their original positions. Equivalently, we apply a symmetric permutation to A, that is,

$$\begin{bmatrix} I & \\ & P \end{bmatrix} \begin{bmatrix} A_1 & A_3 \\ A_2 & A_4 \end{bmatrix} \begin{bmatrix} I & \\ & P^T \end{bmatrix} = \begin{bmatrix} A_1 & A_3P^T \\ PA_2 & PA_4P^T \end{bmatrix} \tag{7}$$

Step 3. Updating. After LU factorization on A_2 and permutations on A, we update the (2,2) block A_4. Assume that $PA_4P^T = \Phi + \Delta + \Psi$, where Φ, Ψ are the strictly lower and upper triangular part of PA_4P^T, and Δ is the diagonal matrix. Then we have

$$L^{(1)^{-1}}[PA_2,\ PA_4P^T] = [U^{(1)},\ L^{(1)^{-1}}(\Phi + \Psi) + L^{(1)^{-1}}\Delta]. \tag{8}$$

We can calculate $L^{(1)^{-1}}(\Phi + \Psi)$, but we can not apply $L^{(1)^{-1}}\Delta$, because the diagonal elements will be changed for different shifts. This computation will be performed for every shift change. $L^{(1)^{-1}}\Delta$ is very easy to implement, since Δ is a diagonal matrix. As is shown in Figure 1 (b), the shadowed area is completely updated, while the meshed area is partly updated, that is, later we need to add $L^{(1)^{-1}}\Delta$ to this part.

We divide the matrix into 2×2 blocks in a recursive way, and use symmetric permutations to control the original diagonal elements in the diagonal line. Obviously it is a kind of divide and conquer method. We define the partition in Figure 1(a) or 1(b) as the level-1 partition. We can apply the same strategy to (1,1) block of the original matrix, and obtain the matrix as shown in Figure 1(c), where $L^{(2)}$ and $U^{(2)}$ are defined similarly to $L^{(1)}$ and $U^{(1)}$, and Figure 1(c) is a level-2 partition. Using this strategy again, we obtain Figure 1(d), which is a level-3 partition. Generally, at level i, the left upper block of level $i-1$ is partitioned into 2×2 blocks again. The algorithm is described in Algorithm 1 as follows.

Algorithm 1 Algorithm of preprocessing step in Strategy I

Define $A^{(0)}$, whose (1,1) block is A. Suppose N level partitions are used.
for $i := 1$ to N **do**
  Define $A^{(i)}$ as (1,1) block of $A^{(i-1)}$, and $A_{21}^{(i)}$ is (2,1) block of $A^{(i)}$.
  Local LU factorization: $P^{(i)}A_{21}^{(i)} = L^{(i)}U^{(i)}$.
  Row permutations and column permutations on A.
  Update the corresponding right part of A with $L^{(i)}$.
end for

Furthermore, we can use $U^{(2)}$ to eliminate the corresponding upper left part in $U^{(1)}$ as shown in Figure 1(e) (In fact, we can combine the (2,1) block of level-2 partition and the upper left part of $U^{(1)}$, and eliminate simultaneously, which gives better numerical stability). Then we can achieve a structure as shown in Figure 1(f). Note that we only need extra memory to store the diagonal elements and the permutation vectors. In Figure 1(f), the updating in the shadowed areas has been

completed. When the shift changes, we first complete the updating by adding the matrices of the form $L^{(1)^{-1}}\Delta$ to the meshed areas, then continue to eliminate the remaining part into an upper triangular part.

Suppose that the size of the matrix is a power of 2. The computation mainly consists of two aspects at level i: the local LU of a matrix of size $n/2^i$; the updating of its corresponding right part, which is the multiplication of a lower triangular matrix and a rectangular matrix of size $n/2^i \times \sum_{j=1}^{i}(n/2^j)$. Assuming that the matrix is divided recursively into $(\log_2 n - 1)$ levels until a 2×2 matrix arrives, the total flops are of the order as follows.

$$\sum_{i=1}^{\log_2 n-1}\left[\frac{2}{3}\left(\frac{n}{2^i}\right)^3+\left(\frac{n}{2^i}\right)^2\sum_{j=1}^{i}\frac{n}{2^j}\right]=n^3\sum_{i=1}^{\log_2 n-1}\left(\frac{1}{4^i}-\frac{1}{3}\cdot\frac{1}{8^i}\right)=\frac{2}{7}n^3+O(n).$$

Therefore, if we use this divide and conquer strategy, the initial stage to achieve the structure of Figure 1(d) will take $O(\frac{2}{7}n^3)$ flops. For the classical LU decomposition, we need $O(\frac{2}{3}n^3)$ flops. So theoretically we can nearly save $3/7 \simeq 43\%$ of floating point operations at each iteration.

In the classical Gaussian elimination, rank-one update is used for each column. In our approach, the recursion in Algorithm 1 can be stop after 3 levels in practice. Hence a low rank update is used for the elimination of each column. It is shown in [20] that the low rank update is an important property to keep the numerical stability in Gaussian elimination. Numerical examples show that the numerical stability of our approach is as good as the classical Gaussian elimination.

3 Strategy II: restricting the diagonal influence to the right

In this section we present a second strategy to save flops. The basic idea is still concentrated on how to control the influence of diagonal elements. Let's split the matrix A in two column blocks:

$$A=[A_1,A_2],\ \text{where } A_1=A(:,1:m), A_2=A(:,m+1:n) \text{ and } m=\left\lceil\frac{n}{2}\right\rceil. \tag{9}$$

We perform $m-1$ steps of eliminations on the left half part A_1, so that we can introduce zeros as many as possible in the left half part, together with the corresponding updating. To achieve this goal, the permutations are very important. In the following, we describe the elimination of column k consisting of two main steps.

Step 1. Permutations. Generally there are two row permutations and one column permutation in this step. Firstly, we use row permutation to move the original diagonal element in this column, denoted by 'X', to position $(2k-1,k)$. Secondly, the biggest element in magnitude in $A(2k:n,k)$ is permuted to position $(2k,k)$. This row permutation is made to keep numerical stability. Thirdly, if there is an element marked by uppercased 'X' in row $2k$ in the left half part of A, then we apply a column permutation to put it to the right half part. We will illustrate this step on a simple 6×6 example in the following.

Step 2. Elimination and updating. We apply Gaussian elimination and introduce zeros in $A(2k+1:n,n)$. Note that there is one original diagonal element in the right half part of row $2k$. Suppose $A(2k,j)(j>2k)$ is this element. Since $A(2k,j)$ changes, we cannot update $A(2k+1:n,j)$.

A 6×6 example. To illustrate the basic idea, we consider the algorithm on a 6×6 matrix A, where the original diagonal elements are uppercase bolded. We will discribe how to do eliminations

in the first and second columns.

$$A = \left(\begin{array}{ccc|ccc} \mathbf{X} & \times & \times & \times & \times & \times \\ \times & \mathbf{X} & \times & \times & \times & \times \\ \times & \times & \mathbf{X} & \times & \times & \times \\ \times & \times & \times & \mathbf{X} & \times & \times \\ \times & \times & \times & \times & \mathbf{X} & \times \\ \times & \times & \times & \times & \times & \mathbf{X} \end{array}\right)$$

1st Column. The diagonal element in column 1 is already in row 1, that is, its original position. So we do not need to permute the original diagonal element to row 1. We find the maximum in $A(2:6,1)$. Suppose that this element is $A(4,1)$. Then we permute row 2 and 4, which is used to keep the numerical stability. After this row permutation, the diagonal element in row 2 is already in the right half part. Hence the column permutation is not needed. Next, we use Gauss transformation matrix to introduce zeros in $A(3:6,1)$ and update the trailing submatrix without modifying the diagonal elements marked by 'X' and the elements in column 4. Note that the updated elements are tilded. It can be illustrated as follows.

- $A(2,:) \leftrightarrow A(4,:)$

$$\left(\begin{array}{ccc|ccc} \mathbf{X} & \times & \times & \times & \times & \times \\ \times & \times & \times & \mathbf{X} & \times & \times \\ \times & \times & \mathbf{X} & \times & \times & \times \\ \times & \mathbf{X} & \times & \times & \times & \times \\ \times & \times & \times & \times & \mathbf{X} & \times \\ \times & \times & \times & \times & \times & \mathbf{X} \end{array}\right)$$

- Eliminate and Update

$$\left(\begin{array}{ccc|ccc} \mathbf{X} & \times & \times & \times & \times & \times \\ \times & \times & \times & \mathbf{X} & \times & \times \\ 0 & \tilde{\times} & \mathbf{X} & \times & \tilde{\times} & \tilde{\times} \\ 0 & \mathbf{X} & \tilde{\times} & \times & \tilde{\times} & \tilde{\times} \\ 0 & \tilde{\times} & \tilde{\times} & \times & \mathbf{X} & \tilde{\times} \\ 0 & \tilde{\times} & \tilde{\times} & \times & \tilde{\times} & \mathbf{X} \end{array}\right)$$

2nd Column. The diagonal element of this column is located in row 4. Hence we first need to permute row 4 and row 3 to move the diagonal element to position $(3,2)$. To keep numerical stability, we try to find the pivot element in $A(4:6,2)$. Suppose that $A(4,2)$ is the largest element in magnitude, then we do not need to apply the second row permutation. Next we need one column permutation. The diagonal element in row 4 is in the left half part. Hence we need to permute column 3 and 5 , such that the updating on the column of the left half part can be completed. After these permutations, we can introduce zeros in this column and update its corresponding right part. The procedure can be shown as follows.

- $A(3,:) \leftrightarrow A(4,:)$

$$\left(\begin{array}{ccc|ccc} \mathbf{X} & \times & \times & \times & \times & \times \\ \times & \times & \times & \mathbf{X} & \times & \times \\ 0 & \mathbf{X} & \times & \times & \times & \times \\ 0 & \times & \mathbf{X} & \times & \times & \times \\ 0 & \times & \times & \times & \mathbf{X} & \times \\ 0 & \times & \times & \times & \times & \mathbf{X} \end{array}\right)$$

- $A(:,3) \leftrightarrow A(:,5)$

$$\left(\begin{array}{ccc|ccc} \mathbf{X} & \times & \times & \times & \times & \times \\ \times & \times & \times & \mathbf{X} & \times & \times \\ 0 & \mathbf{X} & \times & \times & \times & \times \\ 0 & \times & \times & \times & \mathbf{X} & \times \\ 0 & \times & \mathbf{X} & \times & \times & \times \\ 0 & \times & \times & \times & \times & \mathbf{X} \end{array}\right)$$

- Eliminate and Update

$$\left(\begin{array}{ccc|ccc} \mathbf{X} & \times & \times & \times & \times & \times \\ \times & \times & \times & \mathbf{X} & \times & \times \\ 0 & \mathbf{X} & \times & \times & \times & \times \\ 0 & \times & \times & \times & \mathbf{X} & \times \\ 0 & 0 & \mathbf{X} & \times & \times & \tilde{\times} \\ 0 & 0 & \tilde{\times} & \times & \times & \mathbf{X} \end{array}\right)$$

For the general case, A will have a form as shown in Figure 2 after this initial phase. The positions of diagonal elements in the right half part can be regular (Figure 2(a)) or irregular (Figure 2(b)). Note that the shadowed area is completely updated, while the meshed area is only partially updated. When the shift changes, that is, the diagonal elements change, we start from the matrix with the shape as shown in Figure 2(a) or (b) to achieve the final LU factorization.

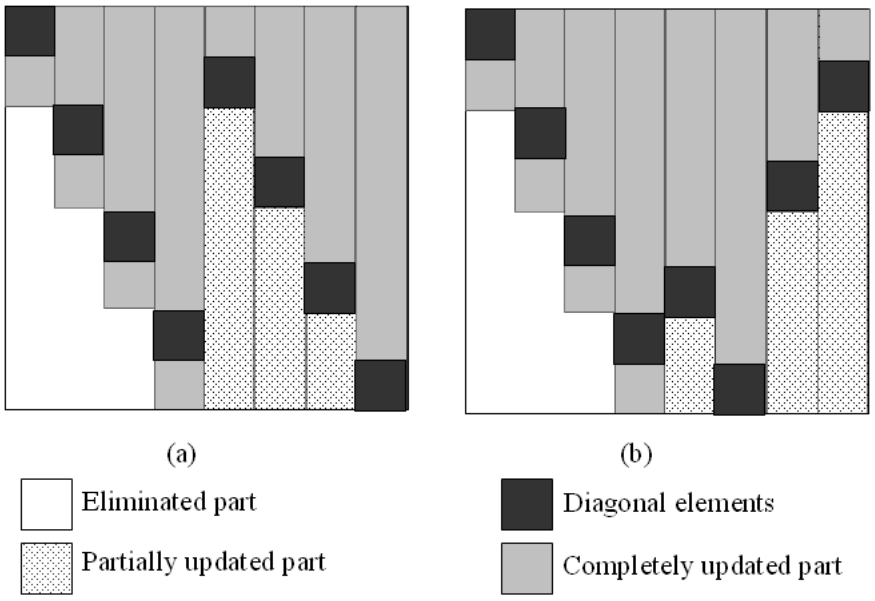

Figure 2: Initial stage of elimination and updating

This strategy is described in Algorithm 2. In this algorithm *diagidxc* stores the row position of the diagonal element of each column, that is, $diagidxc(j) = i$ means that $A(i,j)$ is the original diagonal element. Meanwhile, *diagidxr* is its counterpart for column position. In order to know

the position of the permuted columns in the right half part, we define the set $idxcolSet$ to record the column index.

Algorithm 2 Restricting the diagonal influence to the right half part in Strategy II

$idxcolSet = \{\ \}$; $diagidxc = (1:n)'$; $diagidxr = (1:n)'$; $m = \left\lceil \frac{n}{2} \right\rceil$.
for $k := 1$ to $m-1$ **do**

% Row permutation 1: Permuting the original diagonal element
$d_r = diagidxc(k)$;
$A(2k-1,:) \leftrightarrow A(d_r,:)$;

% Row permutation 2: Partial pivoting
$[q,p] = max(abs(A(2k:n,k)))$;
$p = p + 2k - 1$
$A(2k,:) \leftrightarrow A(p,:)$

% Column permutation
$d_c = diagidxr(2k)$
if $(d_c \leq m)$ **then**
choose $idxcol$ in $\{\{m+1,\ldots,n\} - idxcolSet\}$;
$A(:, diagidxr(2k)) \leftrightarrow A(:, idxcol)$;
$idxcolSet \leftarrow idxcolSet + \{idxcol\}$;
end if

% Elimination and updating
$A(2k+1:n,k) = A(2k+1:n,k)/A(2k,k)$;

% Update first half
for $j = k+1:m$ **do**
$d = diagidxc(j)$;
$A(2k+1:d-1,j) = A(2k+1:d-1,j) - A(2k+1:d-1,k) \times A(2k,j)$;
$A(d+1:n,j) = A(d+1:n,j) - A(d+1:n,k) \times A(2k,j)$;
end for

% Update second half
for $j \in \{\{m+1,\ldots,n\} - idxcolSet\}$ **do**
$d = diagidxc(j)$;
$A(2k+1:d-1,j) = A(2k+1:d-1,j) - A(2k+1:d-1,k) \times A(2k,j)$;
$A(d+1:n,j) = A(d+1:n,j) - A(d+1:n,k) \times A(2k,j)$;
end for

end for

The flops can be counted as follows.

$$flop = \sum_{k=1}^{m-1} [\underbrace{(n-2k)}_{scale} + \underbrace{2(n-2k)\times(n-2k)}_{update}] = \frac{1}{3}n^3 - O(n^2).$$

Since the classical LU decompositon need $O(\frac{2}{3}n^3)$ flops, it means that we can save 50% flops. Note that we apply a serials of rank-one updates to achieve the matrix in Figure 2. After the shift changes, we use another rank-one updates to obtain the final upper triangular matrix. Hence we apply rank-two updates during the elimination of each column. Thus we still use low rank updates, which are important to the numerical stability [20].

4 Numerical experiments

In this section we give numerical examples to check the numerical stability, the flops saving and the computation time. The experiments are performed with MATLAB R2006b on a PC (Intel(R) Core(TM)2 CPU 6600@2.40GHz, 3.21GB SDRAM). In the following, we first examine the numerical stability of the two strategies. Second, we check the actual flops saving in one shift-and-invert iteration. Third, we use these strategies to solve a dense standard eigenvalue problem, check the flops savings, and compare with the standard shift-and-invert using classical LU factorization.

Numerical stability tests. Note that in our approaches we do not use rank-one update as the classical Gaussian elimination. We perform the eliminations locally and the pivot is chosen in a constraint range, but we still use low rank update. We investigate the numerical stability by numerical examples. Here we use the following growth factor suggested by J. Demmel,

$$g_D := \max_j \left\{ \frac{\max_i |u_{ij}|}{\max_i |a_{ij}|} \right\},$$

which is a modification of the growth factor used in xGESVX of LAPACK. Otherwise we can take the famous example with 2^n pivot growth [23], scale the last column down by a factor 2^{-n}, and can not see any pivot growth.

We test random matrices with size of 2^k, where $k = 2:12$. In Figure 3, we give the growth factors of these two strategies. The growth factor of classical Gaussian elimination with partial pivoting (GEPP) is also given for comparison. It clearly shows that the numercial stability of our two strategies is almost as good as GEPP.

Flop savings in LU factorization. For each method, we count the flops (addition and multiplication) during the factorization. For the first strategy, only 3 level partitions are used. We denote the flops of the initial step by $flops1$, and by $flops2$ the flops needed to finish updating and continue the factorization in each iteration. When the shift changes, we do not need to perform the $flops1$ operations in the initial stage. Therefore the percentage of the saved flops is given by $saving = \frac{flops1}{flops1+flops2}$. In each inverse iteration, LU factorization will only need to take $(1 - saving) \times 100\%$ flops of the classic LU. The actual percentages of the saved flops are given in Table 1, which is in accordance with our theoretical analysis in Section 2 and Section 3. For example, for the random matrix with size of 8192, we can nearly save 43% and 50% flops respectively for these two strategies.

Flop savings in the dense eigenproblem. We apply the new strategies to a dense eigenvalue problem. The test matrix is generated like the example 5.5 in [2]. Assume that $A =$

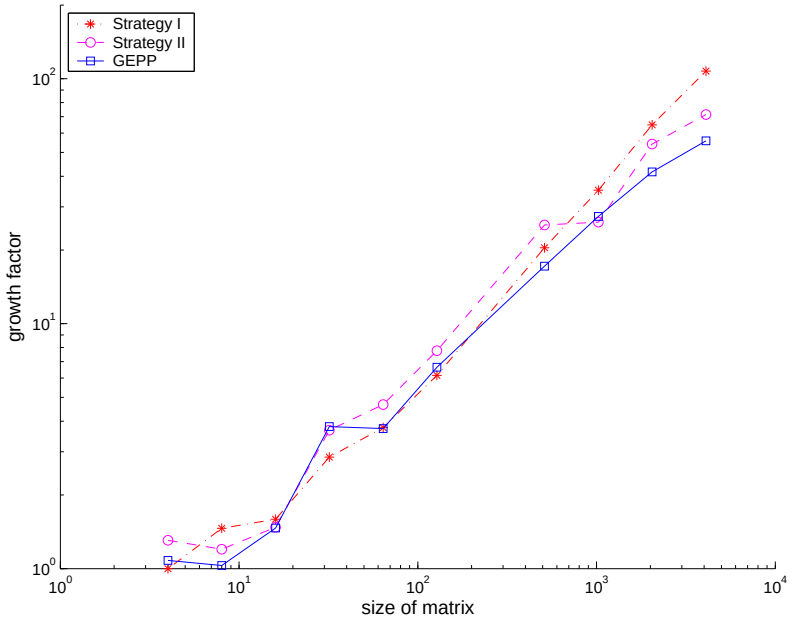

Figure 3: Comparison of growth factors

Size of matrix	Strategy I (%)	Strategy II (%)
512	42.65	49.71
1024	42.69	49.85
2048	42.70	49.93
4096	42.71	49.96
8192	42.71	49.98

Table 1: Actual percentages of flop savings in one LU factorization

$\mathsf{binornd}(1, \alpha, n, n) .* \mathsf{rand}(n) + 10 * \mathsf{diag}(\mathsf{rand}(n, 1))$, where binornd, rand and diag are MATLAB notations. We choose $\alpha = 0.8$ such that the test matrices are dense, and we set the initial eigenvalue as $\sigma_0 = 5.0$ to seek the interior eigenvalue near 5.0. We compare the flops in Table 2, where λ is the computed eigenvalue, ITR stands for the iteration number. For the classical method, we use LU to solve a linear system in each iteration, and the corresponding LU factorization is obtained by Gaussian elimination with partial pivoting (GEPP). Clearly we can see that the new strategies can save nearly half of the flops than the classical method.

n	λ	ITR	Flops count		
			Strategy I	Strategy II	GEPP
512	3.78	6	4.09e08	3.59e08	7.15e08
1024	3.84	6	2.45e09	2.15e09	4.29e09
2048	4.19	8	2.61e10	2.29e10	4.58e10

Table 2: Comparison of the flops in the eigenvalue computation

5 Conclusions

The shift-and-invert is an important technique in eigenvalue computation, especially for the interior spectrum. To avoid redoing LU factorization from scratch in every shift change, we present two LU based approaches to save flops. The first strategy uses a divide and conquer approach, in which the matrix is divided into 2-by-2 blocks recursively, and the parts of the blocks are eliminated. The second strategy uses two row permutations and one column permutation for each column elimination to control the position of original diagonal elements, such that the influence of diagonal elements is confined in the right part. For both methods, when the shift changes we first complete the updating, and then start the elimination from this point. The first method can save 43% flops, and the second can save 50% flops. Only little extra memory is needed. For the first method, we need extra memory to store the diagonal elements and the permutation vector. For the second method, we need extra memory to store some indices. We test both methods with numerical examples. The numerical stability is almost as good as the classical LU factorization. In this paper we only consider dense eigenvalue computations. In the sparse case, a time and memory efficient sparse direct solver is very important for the shift-and-invert strategy. For example, the symmetric indefinite sparse direct solver for the shift-and-invert technique is successfully used in the computation of interior eigenvalues and eigenvectors for the Anderson problem [14]. Generally the situation is more complicated for sparse matrices. Reordering is used in sparse LU to reduce the number of fill-in elements introduced during the elimination. Usually the matrix is very sparse at the beginning of the elimination, but it gets denser and denser as the factorization proceeds. In our two approaches, most of the savings come from annihilating elements in the left bottom corner of the matrix in the pre-processing step. Hence we expect that they will not lead to important savings in the sparse case.

References

[1] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. van der Vorst, editors, Templates for

the solution of Algebraic Eigenvalue Problems: A Practical Guide . SIAM, Philadelphia, 2000.

[2] M. Crouzeix, B. Philippe, M. Sadkane, *The Davidson method*, SIAM J. Sci. Comput., 15 (1994), pp. 62-76.

[3] E. R. Davidson, *The iteration calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices*, J. Comp. Phys., 17 (1975), pp. 87-94.

[4] G. H. Golub, C. F. Van Loan, Matrix Computations, 3rd edition, The Johns Hopkins University Press, 1996.

[5] G. H. Golub, Q. Ye , *Inexact inverse iteration for generalized eigenvalue problems*, BIT, 40 (2000), pp. 671-684.

[6] M. E. Hochstenbach, Y. Notay, *The Jacobi-Davidson method*, GAMM-Mitteilungen, 29 (2006), pp. 368-382.

[7] Y.-L. Lai, K.-Y. Lin, W.-W. Lin, *An inexact inverse iteration for large sparse eigenvalue problems*, Numer. Linear Algebra Appl., 4 (1997), pp. 425-437.

[8] B. N. Parlett, The Symmetric Eigenvalue Problem. Prentice-Hall, Englewood Cliffs, NJ, 1980. Reprinted as Classics in Applied Mathematics 20, SIAM, Philadelphia, 1997.

[9] B. N. Parlett, D. R. Taylor, Z. A. Liu, *A look-ahead Lanczos algorithm for unsymmetric matrices*, Math. Comp., 44 (1985), pp. 105-124.

[10] G. Peters, J. H. Wilkinson, *Inverse iteration, ill-conditioned equations and Newton's method*, SIAM Review, 21 (1979), pp. 339-360.

[11] A. Ruhe, *Rational Krylov sequence methods for eigenvalue computation*, Lin. Algebra Appl., 58 (1984), pp. 391-405.

[12] A. Ruhe, *Rational Krhlov: a practical algorithm for large sparse nonsymmetric matrix pencils*, SIAM J. Sci. Comput., 19 (1998), pp. 1535-1551.

[13] Y. Saad, Numerical Methods for Large Eigenvalue Problems, Manchester University Press, Manchester, 1992.

[14] O. Schenk, M. Bollhöfer, R. A. Römer, *On large-scale diagonalization techniques for the Anderson model of localization,* SIAM Review, 50 (2008), pp. 91-112, also appears in SIAM J. Sci. Comput., 28 (2006), pp. 963-983.

[15] V. Simoncini and L. Elden, *Inexact Rayleigh quotient-type methods for eigenvalue computations*, BIT, 42 (2002), pp. 159-182.

[16] G. L. G. Sleijpen, H. A. van der Vorst, *A Jacobi-Davidson iteration method for linear eigenvalue problems*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 401-425. Reprinted in SIAM Review, 42 (2000), pp. 267-293.

[17] D. C. Sorensen, *Implicit application of polynomial filters in a k-step Arnoldi method*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 357-385.

[18] D. C. Sorensen, C. Yang, *A truncated RQ iteration for large scale eigenvalue calculation,* SIAM J. Matrix Anal. Appl., 19 (1998), pp. 1045-1073.

[19] G. W. Stewart, Matrix Algorithms, Volume II: Eigensystems, SIAM, Philadelphia, PA, 2001.

[20] L. N. Trefethen, R. S. Schreiber, *Average-case stability of Gaussian elimination*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 335-360.

[21] H. A. van der Vorst, *Computational methods for large eigenvalue problems*, in P. G. Ciarlet and J. L. Lions (eds), Handbook of Numerical Analysis, Volume VIII, North-Holland (Elsevier), Amsterdam, 2002, pp. 3-179.

[22] D. S. Watkins , The Matrix Eigenvalue Problem: GR and Krylov Subspace Methods, SIAM, Philadelphia, PA, 2008.

[23] J. H. Wilkinson, The Algebraic Eigenvalue Problem, Clarendon Press, Oxford, 1965.

[24] C. Yang, *Convergence analysis of an inexact truncated RQ-iteration*, Electronic Transactions on Numerical Analysis, 7 (1998), pp. 40-55.

[25] Y. Zhou, Y. Saad, *A Chebyshev-Davidson algorithm for large symmetric eigenproblems*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 954-971.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399