Lanczos bidiagonalization with partial reorthogonalization

Rasmus Munk Larsen
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK-8000 Aarhus C
e-mail: rmunk@daimi.aau.dk

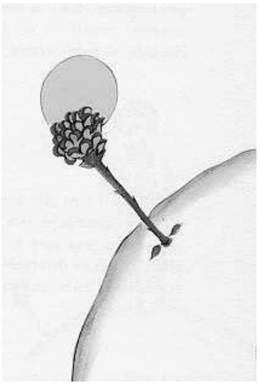
October 1998

Abstract

A partial reorthogonalization procedure (BPRO) for maintaining semi-orthogonality among the left and right Lanczos vectors in the Lanczos bidiagonalization (LBD) is presented. The resulting algorithm is mathematically equivalent to the symmetric Lanczos algorithm with partial reorthogonalization (PRO) developed by Simon, but works directly on the Lanczos bidiagonalization of A. For computing the singular values and vectors of a large sparse matrix with high accuracy, the BPRO algorithm uses only half the amount of storage and a factor of 3–4 less work compared to methods based on PRO applied to an equivalent symmetric system. Like PRO, the algorithm presented here is based on simple recurrences, which enable it to monitor the loss of orthogonality among the Lanczos vectors directly without forming inner products. These recurrences are used to develop a Lanczos bidiagonalization algorithm with partial reorthogonalization, which has been implemented in a MATLAB package for sparse SVD and eigenvalue problems called PROPACK. Numerical experiments with the routines from PROPACK are conducted using a test problem from inverse helioseismology to illustrate the properties of the method. In addition, a number of test matrices from the Harwell-Boeing collection are used to compare the accuracy and efficiency of the MATLAB implementations of BPRO and PRO with the `svds` routine in MATLAB 5.1, which uses an implicitly restarted Lanczos algorithm.

Contents

1 Introduction



"Iterative algorithms are delicate flowers."
— V. A. Barker

Sah ein Knab' ein Röslein stehn,
Röslein auf der Heiden.
War so jung und morgenschön,
Lief er schnell, es nah zu sehn,
Sah's mit vielen Freuden.
Röslein, Röslein, Röslein rot,
Röslein auf der Heiden.
— Johann Wolfgang von Goethe

The Lanczos bidiagonalization, which is due to Golub and Kahan [26], and its block extension has been used as the computational kernel in a number of methods, and has proved to be an efficient tool for computing the singular value decomposition (SVD) of large and sparse or structured matrices [4, 27, 16]. It is also widely used for solving large sparse linear least squares problems, and it has been shown (cf. [7]) that methods based on the LBD, such as the LSQR algorithm by Paige and Saunders [49], belong to the most stable iterative algorithms for problems where the coefficient matrix is ill-conditioned. Moreover, LBD is used in a number of regularization algorithms for the solution of discrete ill-posed problems, see e.g. [5, 8, 11, 31, 42, 46]. Finally, the LBD is very useful for solving least squares problems with multiple right-hand sides, see [41, 46].

Like the symmetric Lanczos process (e.g. [15, 51]), the Lanczos bidiagonalization is plagued by the influence of rounding errors. When it is carried out in finite precision arithmetic this leads to the loss of orthogonality among the Lanczos vectors and the appearance of spurious singular values. The algorithm presented in this paper solves this problem by implementing a compact variant of the *partial reorthogonalization* (PRO) scheme, developed by Simon [63] for the symmetric Lanczos algorithm. It differs from Simon's method by taking advantage of the particular form of the recurrences in the Lanczos bidiagonalization and works directly on the $m \times n$ matrix A without forming an equivalent symmetric system. This leads to a more efficient algorithm, as discussed below.

It was demonstrated by Berry [4], that sparse SVD calculations based on the symmetric Lanczos algorithm with PRO are among the most efficient methods for computing the largest singular values and corresponding singular vectors of large sparse matrices. In SVDPACK, the subroutine lanso routine from the LANSO[1] package developed by Parlett and co-workers, is used to compute explicitly the eigenvalues of one of the symmetric matrices $A^T A$, or

$$C = \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix},$$

from which the singular values of A can be found. However, working with $A^T A$ can lead to poor accuracy of the computed singular values when A is ill-conditioned. This problem is avoided by computing the eigenvalues of C, but unfortunately this significantly increases the amount of storage and work required by the reorthogonalization, because Lanczos vectors of

[1] LANSO should not be confused with the LASO package also available from Netlib, which implements the *Selective reorthogonalization* algorithm by Parlett and Scott [55].

length $m+n$ must be used. Also, the number iterations required to compute a given number of singular values is doubled. More recently, methods based on the implicitly restarted Lanczos algorithm have appeared (cf. [12, 65]). Various studies, see e.g. [23], indicate that these are robust and efficient tools for computing a few of the largest singular values and vectors. Since these methods are also developed with eigenvalue problems in mind, they too compute the SVD by working explicitly with $A^T A$ or C and consequently share the problems described above with algorithms based on the symmetric Lanczos process.

The BPRO algorithm presented here is designed specifically with large sparse SVD calculations in mind, and by taking advantage of the special form of the recurrences in the LBD, it is possible to combine good accuracy with the speed obtained by methods working with $A^T A$ as will be demonstrated by the experiments reported in Section 7.

Parallel implementations of both the lanso subroutine (cf. [67]) and the ARPACK package (cf. [43]), which implements the implicitly restarted Lanczos algorithms, are now available. The routines in SVDPACK (cf. [4]), which build on an older sequential version of lanso are available from Netlib. This software is freely available, and in Table 1 we have listed the relevant Web addresses from where it may be downloaded.

Table 1: *Available software for large sparse SVD computations.*

PARPACK	:	`http://www.caam.rice.edu/software/ARPACK`
PLANSO	:	`http://www.nersc.gov/research/SIMON/planso.html`
SVDPACK	:	`http://www.netlig.org/svdpack`

The algorithms discussed in this paper, have been implemented in a MATLAB package called PROPACK. The package is described in appendix B and is available from the author upon request.

This paper is organized as follows: In Section 2 we describe the LBD algorithm, and review how it may be used for SVD calculations and for solving linear least squares problems in Sections 3 and 4. In Section 5 we discuss the behavior of LBD in finite precision arithmetic. Following this, in Section 6 we show how a set of simple recurrences can be used for monitoring the loss of orthogonality among the Lanczos vectors. A partial reorthogonalization algorithm for the LBD process based on estimates of the level of orthogonality computed using these recurrences is described. In Section 7 we illustrate the properties of the resulting algorithm, by applying a MATLAB implementation of the algorithm to a number of test problems from the Harwell-Boeing collection, in addition to a discrete ill-posed problem from inverse helioseismology (see [62] for a general introduction), which was also used as a test problem in [14, 36, 42]. We compare the accuracy of the computed results and the number of operations with

- symmetric Lanczos and LBD with *full reorthogonalization* (FRO and BFRO),
- symmetric Lanczos with partial reorthogonalization (PRO),
- the eigs routine in MATLAB 5.1, which uses an implicitly restarted Lanczos algorithm,
- and the standard QR algorithm from the LINPACK routine _SVDC used by the svd routine in MATLAB.

1.1 Notation

In the following we will use uppercase Roman letters $A, B, C, \ldots$ to denote matrices, lowercase Roman letters $x, y, z, \ldots$ will denote (column) vectors, and lowercase Greek letters $\alpha, \beta, \gamma, \ldots$ are used for scalars. We use e_i to denote the ith column of the identity matrix. The symbol $\mathbb{R}^{m\times n}$ will denote the set of matrices with m rows and n columns, and $\mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$ will denote the $n\times n$ diagonal matrix $D = (d_{ij}), \quad d_{ii} = \sigma_i$. For vectors we use $\|x\|_p$ to denote the p-norm $\|x\|_p = (\sum (x_i)^p)^{1/p}$ and for matrices $\|A\|_p$ will denote the induced matrix p-norm

$$\|A\|_p = \max_{x\neq 0} \frac{\|A\,x\|_p}{\|x\|_p}$$

In particular $\|x\|_2$ is the usual euclidian norm, and $\|A\|_2 = \sigma_1(A)$ is the largest singular value of A. We use $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$ to denote the eigenvalues and $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$ the singular values of the matrix A. The following shorthand notation will be used to denote the set of eigenvalues or singular values a matrix: $\lambda(A) \equiv \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, $\sigma(A) \equiv \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. Furthermore

$$\angle(x, y) = \arccos \frac{x^T y}{\|x\|_2 \|y\|_2}$$

will denote the acute angle between non-zero vectors x and y, and

$$\angle(x, V) = \min_{v\in \mathrm{span}(V)} \angle(x, v)$$

the acute angle between the vector x and the subspace spanned by the columns of V. Finally $A^\dagger \equiv (A^T A)^{-1} A^T$ will denote the More-Penrose pseudo-inverse of A.

When discussing effects of finite precision arithmetic $\mathit{fl}(x)$ will denote machine number closest to x. Furthermore we use the symbol $\mathbf{u}$ to denote the unit of machine round-off, which in IEEE double precision is $2^{-53} \approx 1.11 \cdot 10^{-16}$, and the symbol $\gamma_n = n\mathbf{u}/(1 - n\mathbf{u})$ introduced by Higham in [37]. When discussing different variants of the Lanczos process with different kinds of reorthogonalization we shall use the following abbreviations

LBD	:	Lanczos bidiagonalization
FRO	:	Symmetric Lanczos with full reorthogonalization
BFRO	:	Lanczos bidiagonalization with full reorthogonalization
PRO	:	Symmetric Lanczos with Partial reorthogonalization
BPRO	:	Lanczos bidiagonalization with partial reorthogonalization,

where by "Symmetric Lanczos" we mean the original Lanczos algorithm from [40] for reducing a symmetric matrix to tridiagonal form. The Lanczos bidiagonalization algorithm is defined in the following section.

2 Lanczos bidiagonalization

We begin this section by stating the fundamental recurrences that define the Lanczos bidiagonalization. In the paragraphs below we describe in further detail how this iterative process may be used in SVD calculations and for solving least squares problems. We use consistently throughout the paper the variant described by Paige and Saunders [49], which is the appropriate version for solving least squares problems (cf. [7]). For this reason, our formulas in the section on SVD calculations differ slightly from the presentation in e.g. [29], however in that context there are no differences in computational efficiency or accuracy between the two forms.

For a rectangular $m \times n$ matrix A the Lanczos bidiagonalization computes a sequence of *Lanczos vectors* $u_j \in \mathbb{R}^m$ and $v_j \in \mathbb{R}^n$ and scalars α_j and β_j for $j = 1, 2, \ldots, k$ as follows:

1. Choose a starting vector $p_0 \in \mathbb{R}^m$, and let
   $\beta_1 = \|p_0\|_2$, $u_1 = p_0/\beta_1$ and $v_0 \equiv 0$
2. **for** $j = 1, 2, \ldots, k$ **do**
   - $r_j = A^T u_j - \beta_j v_{j-1}$
   - $\alpha_j = \|r_j\|_2$
   - $v_j = r_j/\alpha_j$
   - $p_j = Av_j - \alpha_j u_j$
   - $\beta_{j+1} = \|p_j\|_2$
   - $u_{j+1} = p_j/\beta_{j+1}$

   end

In the following we will refer to one passage through the **for** loop at 2. as a *Lanczos step* (or an *iteration*). After k steps, we have generated the **lower** bidiagonal matrix

$$B_k = \begin{pmatrix} \alpha_1 & & & \\ \beta_2 & \alpha_2 & & \\ & \beta_3 & \ddots & \\ & & \ddots & \alpha_k \\ & & & \beta_{k+1} \end{pmatrix} . \tag{2.1}$$

In exact arithmetic the Lanczos vectors are orthonormal such that

$$U_{k+1} = (u_1, u_2, \ldots, u_{k+1}) \in \mathbb{R}^{m\times(k+1)} , \qquad U_{k+1}^T U_{k+1} = I_{k+1} , \tag{2.2}$$

and

$$V_k = (v_1, v_2, \ldots, v_k) \in \mathbb{R}^{n\times k} , \qquad V_k^T V_k = I_k , \tag{2.3}$$

where I_k is the $k \times k$ identity matrix. By construction the columns of U_{k+1} and V_k satisfy the recurrences

$$\alpha_j v_j = A^T u_j - \beta_j v_{j-1} \tag{2.4}$$
$$\beta_{j+1} u_{j+1} = Av_j - \alpha_j u_j , \tag{2.5}$$

and we can write this in a compact matrix form as

$$AV_k = U_{k+1} B_k \tag{2.6}$$
$$A^T U_{k+1} = V_k B_k^T + \alpha_{k+1} v_{k+1} e_{k+1}^T , \tag{2.7}$$

which is illustrated in Figure 1.

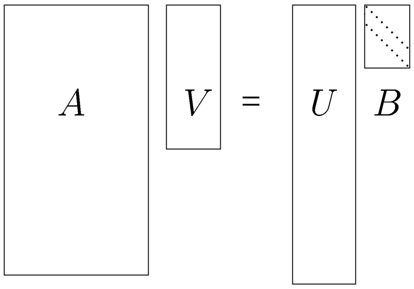


Figure 1: Lanczos bidiagonalization of A.

Since (2.6) can also be written $U_{k+1}^T A V_k = B_k$, we will refer to the columns of U_{k+1} as *left Lanczos vectors* and the columns of V_k as *right Lanczos vectors.* Moreover,

$$u_{k+1} \in \mathcal{K}_{k+1}(AA^T, u_1) \equiv \left\{u_1, AA^T u_1, \ldots, (AA^T)^k u_1\right\}, \tag{2.8}$$

$$v_k \in \mathcal{K}_k(A^T A, v_1) \equiv \left\{v_1, A^T A v_1, \ldots, (A^T A)^{k-1} v_1\right\} , \tag{2.9}$$

and therefore $u_1, u_2, \ldots, u_{k+1}$ and $v_1, v_2, \ldots, v_k$ form an orthogonal basis for these two Krylov subspaces.

In the following we discuss how the bidiagonalization can be used to calculate approximations to the singular values of A and to solve linear least squares problems.

3 The Lanczos algorithm and sparse SVD calculations

The Lanczos bidiagonalization of A is closely related to the original Lanczos process applied to an equivalent symmetric matrix and below it is reviewed how this leads to an SVD algorithm based on the recurrences described in the previous section. A similar presentation for the block Lanczos bidiagonalization is given in [27]. Before we proceed, let us review some fundamental results for the symmetric eigenvalue problem and the symmetric Lanczos process, which we must apply to understand the properties of the Lanczos bidiagonalization.

3.1 Equivalent symmetric problems

The singular value decomposition of A is closely related to the Schur decomposition of the symmetric matrices $A^T A \in \mathbb{R}^{n\times n}$, $AA^T \in \mathbb{R}^{m\times m}$ and

$$C \equiv \begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \in \mathbb{R}^{(m+n)\times(m+n)} , \tag{3.1}$$

and these relations form the basis of any SVD algorithm:

Theorem 1 *Let A be an $m \times n$ matrix and assume without loss of generality that $m \geq n$. Let further the singular value decomposition of A be*

$$U^T A V = \mathrm{diag}(\sigma_1, \ldots, \sigma_n) . \tag{3.2}$$

Then

$$V^T (A^T A) V = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_n^2) , \tag{3.3}$$

$$U^T(AA^T)U = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_n^2, \underbrace{0, \ldots, 0}_{m-n}) \ . \tag{3.4}$$

Moreover, if U is partitioned as

$$U = \begin{array}{cc} [\ U_1 & U_2 \ ] \\ n & m-n \end{array} ,$$

then the orthonormal columns of the $(m+n) \times (m+n)$ matrix

$$Y = \frac{1}{\sqrt{2}} \begin{pmatrix} U_1 & U_1 & \sqrt{2}U_2 \\ V & -V & 0 \end{pmatrix} \tag{3.5}$$

form an eigenvector basis for the matrix C defined in (3.1), and

$$Y^T C Y = \mathrm{diag}(\sigma_1, \ldots, \sigma_n, -\sigma_1, \ldots, -\sigma_n, \underbrace{0, \ldots, 0}_{m-n}) \ . \tag{3.6}$$

Proof. The theorem follows immediately by substituting the SVD of A in (3.3), (3.4) and (3.6) followed by application of (3.5) and the orthogonality of U and V. □

The theorem tells us, that we can obtain the singular values and vectors of A by computing the eigenvalues and corresponding eigenvectors of one of the equivalent symmetric matrices. This forms the basis of any SVD algorithm. As an example, the standard algorithm, which in its original form is due to Golub and Kahan [26] and used in e.g. LAPACK, computes the SVD by *implicitly* applying the QR algorithm for the symmetric eigenvalue problem to A^TA.

3.2 Fundamental error analysis for SVD calculations

The fact that A^TA is formed only implicitly is crucial to the numerical stability of the algorithm. This is true for any SVD algorithm: Unless A has a very small condition number, the rounding errors that occur when A^TA is formed perturb the matrix such that we cannot expect to determine the smallest singular values of the original matrix A from the smallest eigenvalues of A^TA with any accuracy. To state this more quantitatively, we regard the forward error bound for matrix multiplication (see [37, Section 3.5]):

$$fl(A^TA) = A^TA + E \ , \qquad \|E\|_2 \leq \gamma_m \, \|A^T\|_2 \, \|A\|_2 \leq \gamma_m \cdot \sigma_1^2 \ , \tag{3.7}$$

where σ_1 is the largest singular value of A and $\gamma_m = m\mathbf{u}/(1-m\mathbf{u}) \leq 1.01m\mathbf{u}$ when $m\mathbf{u} \leq 0.01$ (for IEEE double precision this holds as long as A has fewer than $4.5 \cdot 10^{13}$ rows). Now, perturbation theory for the symmetric eigenvalue problem (see e.g. [29, Sections 8.1, 8.3]) can tell us how much the eigenvalues of $fl(A^TA)$ differ from those of A^TA. The result, expressed in terms of the singular values of A, is

$$|\hat{\sigma}_i^2 - \sigma_i^2| \leq m \cdot \mathbf{u} \cdot \sigma_1^2 \ , \tag{3.8}$$

where $\hat{\sigma}_i, \ i = 1, \ldots, n$ are the eigenvalues of $fl(A^TA)$. This shows that no matter how accurate the algorithm we subsequently use to calculate $\hat{\sigma}_i$, we cannot hope to recover the smallest singular values with high relative accuracy. Any information about the small singular values has drowned in the rounding errors. The bound in (3.8) is rather difficult to interpret, since

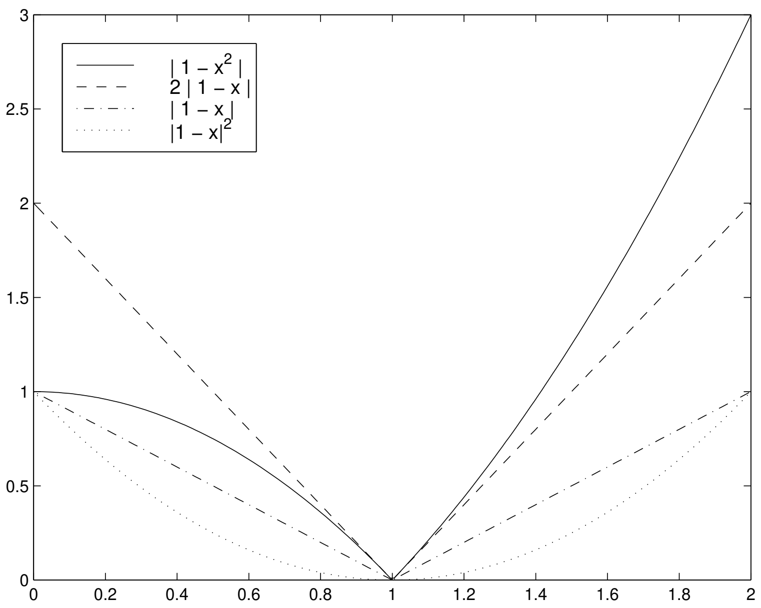


Figure 2: *Illustration of the bounds on the relative error in $\hat{\sigma}_i$. The value on the abscissa represents $x = \hat{\sigma}_i/\sigma_i$ in the expressions for the relative error bounds, so $x = 1$ means that the absolute error is zero.*

it gives an expression for the error in $\hat{\sigma}_i^2$ rather than $\hat{\sigma}_i$. We can obtain a more useful bound noting that both $\hat{\sigma}_i$ and σ_i are both positive numbers, from which it follows that

$$\frac{|\hat{\sigma}_i - \sigma_i|}{\sigma_i} \leq \frac{|\hat{\sigma}_i^2 - \sigma_i^2|}{\sigma_i^2} \leq m \cdot \mathbf{u} \cdot \frac{\sigma_1^2}{\sigma_i^2} \ . \tag{3.9}$$

The inequality is illustrated in Figure 2, where the expression on the left side of (3.9) is shown as the dot-dashed, and the original bound given by the middle expression is shown as the solid line. By the following argument:

$$\begin{aligned}
\left(\frac{|\hat{\sigma}_i - \sigma_i|}{\sigma_i}\right)^2 &= \left|1 - \frac{\hat{\sigma}_i}{\sigma_i}\right|^2 \leq \left|1 - \frac{\hat{\sigma}_i}{\sigma_i}\right| \left|1 + \frac{\hat{\sigma}_i}{\sigma_i}\right| \\
&= \left|\left(1 - \frac{\hat{\sigma}_i}{\sigma_i}\right)\left(1 + \frac{\hat{\sigma}_i}{\sigma_i}\right)\right| = \left|1 - \left(\frac{\hat{\sigma}_i}{\sigma_i}\right)^2\right| \\
&= \frac{|\hat{\sigma}_i^2 - \sigma_i^2|}{\sigma_i^2} \ ,
\end{aligned}$$

we get a different bound

$$\frac{|\hat{\sigma}_i - \sigma_i|}{\sigma_i} \leq m^{1/2} \cdot \sqrt{\mathbf{u}} \cdot \frac{\sigma_1}{\sigma_i} \ , \tag{3.10}$$

but this is even weaker than (3.9) except when the relative error is larger than 2, as shown by the dotted line in Figure 2. The figure can however inspire us to find a slightly tighter bound. The dashed line indicates that when $\hat{\sigma}_i \geq \sigma_i$ we can reduce the bound by a factor of 2:

$$\frac{|\hat{\sigma}_i - \sigma_i|}{\sigma_i} \leq \frac{|\hat{\sigma}_i^2 - \sigma_i^2|}{2\sigma_i^2} \leq 0.5 \cdot m \cdot \mathbf{u} \cdot \frac{\sigma_1^2}{\sigma_i^2} \ , \tag{3.11}$$

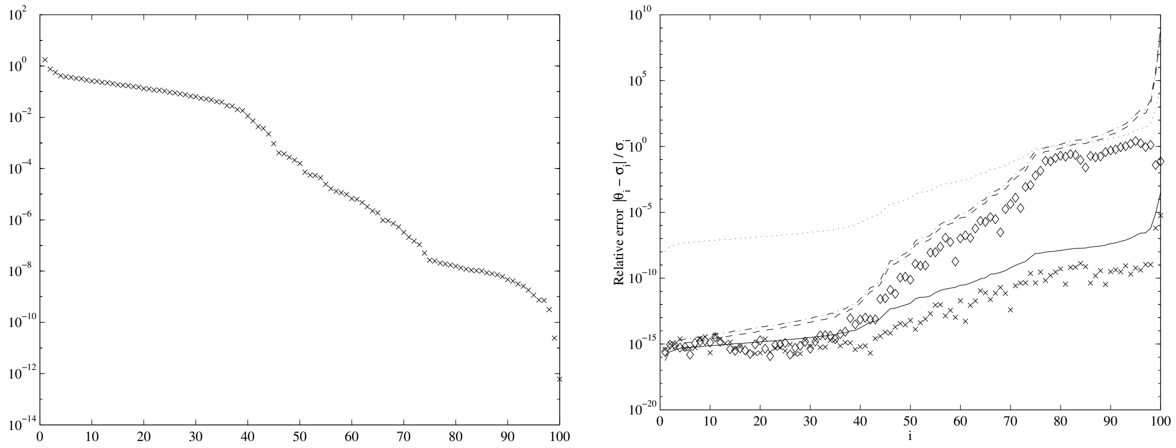


Figure 3: *Left panel: Singular values of the testmatrix HELIO212b. Right panel: Illustration of the error bounds. Diamonds show the relative error in singular values of test matrix HELIO212b (see Section 7.3) computed as the eigenvalues of A^TA using the* `eig` *routine in MATLAB, and crosses the singular values computed directly from A using the* `svd` *routine in MATLAB. The dotted, dashed and dot-dashed lines show the value predicted by error bounds in (3.10), (3.13) and (3.9) and with $m = 1$, and the solid line shows the bound in (3.14) with $p(m, n) = 1$.*

To get a bound for $\hat{\sigma}_i \leq \sigma_i$ we simply swap σ_i and $\hat{\sigma}_i$ in the inequality above (which corresponds to replacing x by $1/x$ in the figure) to get

$$\frac{|\hat{\sigma}_i - \sigma_i|}{\hat{\sigma}_i} \leq \frac{|\hat{\sigma}_i^2 - \sigma_i^2|}{2\hat{\sigma}_i^2} \leq 0.5 \cdot m \ \cdot \mathbf{u} \cdot \frac{\sigma_1^2}{\hat{\sigma}_i^2} , \qquad \text{for} \quad \hat{\sigma}_i \leq \sigma_i . \tag{3.12}$$

Now we can write the new bound in the compact form

$$\frac{|\hat{\sigma}_i - \sigma_i|}{\min(\hat{\sigma}_i, \sigma_i)} \leq 0.5 \cdot m \cdot \mathbf{u} \cdot \frac{\hat{\sigma}_1^2}{\min(\hat{\sigma}_i, \sigma_i)^2} . \tag{3.13}$$

The last inequality is not in the usual form of a forward error bound, since we bound the error relative to the size of the computed quantity rather than the true value, but that is the price we pay for the tighter bound. As can be seen in Figure 2, this is in fact the best linear bound we can hope for.

This should be compared with the error bound for a backward stable method, which nearly computes the exact SVD of a matrix $A + E$ where $\|E\|_2/\|A\|_2 \leq p(m, n)\mathbf{u}$, and $p(m, n)$ is some slowly growing function of m and n. In this situation we have that

$$\frac{|\hat{\sigma}_i - \sigma_i|}{\sigma_i} \leq p(m, n) \cdot \mathbf{u} \cdot \frac{\sigma_1}{\sigma_i} , \tag{3.14}$$

see e.g. [2, Section 4.7, 4.9]. It follows from the error analysis of Paige [48] that the symmetric Lanczos algorithm applied to C (and therefore also the LBD, as we shall see below) satisfy bounds on this form.

The bounds and actual errors in the computed singular values of one the test-matrices from Section 7.3 are illustrated in Figure 3.

3.3 The Lanczos algorithm

For a large and sparse matrix the Golub-Kahan algorithm is impractical. The algorithm starts out by applying a series of similarity transformations directly to A to reduce it to bidiagonal form. Therefore it requires the matrix to be stored explicitly, which may be impossible simply due to its size. Moreover, it may be difficult to take advantage of any structure or sparsity in A, since this is quickly destroyed by the transformations that are applied to the matrix.

3.3.1 Approximate SVD using Lanczos on matrix C

When A is large and sparse or structured a more efficient method for computing the SVD is to use the symmetric Lanczos process (see e.g. [15, 51]) applied to one of the equivalent symmetric systems, since then the matrix is only accesses via matrix-vector products with A and A^T. If for example we apply the symmetric Lanczos process with starting vector $q_1 = (u_1^T, 0)^T$ then after $2k$ steps we have reduced C to the special tridiagonal matrix

$$T_{2k} = \begin{pmatrix} 0 & \alpha_1 & & & \\ \alpha_1 & 0 & \beta_2 & & \\ & \beta_2 & 0 & \ddots & \\ & & \ddots & \ddots & \alpha_k \\ & & & \alpha_k & 0 \end{pmatrix} . \tag{3.15}$$

The corresponding Lanczos vectors $q_i \in \mathbb{R}^{m+n}$, $i = 1, \ldots, 2k$ computed in the process alternate between two forms:

$$q_{2j-1} = (u_j^T, 0)^T \ , \quad q_{2j} = (0, v_j^T)^T \ , \quad j = 1, \ldots, k \ ,$$

where $u_j \in \mathbb{R}^m$, $v_j \in \mathbb{R}^n$, and in exact arithmetic α_j, β_j, u_j and v_j are identical to the corresponding quantities computed by the Lanczos bidiagonalization with starting vector u_1. Now the Lanczos tridiagonalization of C can be written in terms of the matrices $Q_{2k+1} = (q_1, q_2, \ldots, q_{2k})$ and T_{2k} as the following relation

$$C\,Q_{2k} = Q_{2k}T_{2k} + \beta_{k+1} q_{2k+1} e_{2k}^T \tag{3.16}$$

$$= Q_{2k}T_{2k} + \beta_{k+1} \begin{pmatrix} u_{k+1} \\ 0 \end{pmatrix} e_{2k}^T \ , \tag{3.17}$$

which is illustrated in Figure 4.

As the number of iterations is increased, the eigenvalues θ_i, $i = 1, \ldots, 2k$ of T_{2k}, which are the *Ritz values* of C with respect to the Krylov subspace span(Q_{2k}), will be increasingly accurate approximations to the eigenvalues of C, and according to Theorem 1 the k non-negative ones will be approximations to the largest singular values of A. The Ritz values are found by computing the Schur decomposition of T_{2k}:

$$S_{2k}^T T_{2k} S_{2k} = \mathrm{diag}(\theta_1, \ldots, \theta_{2k}) \ , \tag{3.18}$$

This can be done by using, e.g., the symmetric QR algorithm (see [29, 2], LAPACK routine _STEQR), or bisection based on Sturm sequences followed by inverse iteration (see [3, 2], LAPACK routines _STEBZ and _STEIN). While the QR algorithm is usually the fastest for computing all eigenvalues of a tridiagonal matrix, the latter approach has the advantage that

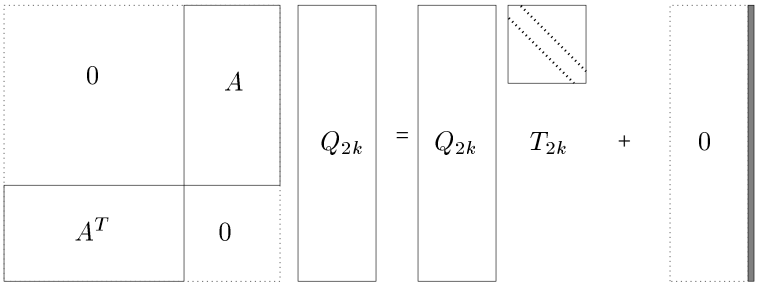


Figure 4: Lanczos tridiagonalization of C. The shaded vector on the right-hand side is $\beta_{k+1}q_{2k+1}$.

each eigenpair is computed independently, and therefore it is only necessary to compute the part of the spectrum that is actually needed – in this case the positive eigenvalues. This saves half the work and half the storage needed for the eigenvectors of T_{2k+1}, and also makes the eigenvalue calculations easy to parallelize (see e.g. [18]).

The accuracy of the computed Ritz values may be estimated using the following result, which is originally due to Paige [48]:

Theorem 2 *Suppose $2k$ steps of the symmetric Lanczos process on C have been performed and that the Schur decomposition of the tridiagonal matrix T_{2k} is given by (3.18). Let the matrix of Ritz vectors be*

$$Y_{2k} = (y_1, \ldots, y_{2k}) = Q_{2k} S_{2k} \in \mathbb{R}^{(m+n)\times 2k}$$

then for $i = 1, \ldots, 2k$ we have

$$\|Cy_i - \theta_i y_i\|_2 = |\beta_{k+1}|\,|s_{2k,i}| \equiv \xi_{ki} \ , \tag{3.19}$$

where $S_{2k} = (s_{pq})$.

Proof. Postmultiply (3.17) by S_{2k} to obtain

$$CY_{2k} = Y_{2k}\,\mathrm{diag}(\theta_1, \ldots, \theta_{2k}) + \beta_{k+1}\begin{pmatrix} u_{k+1} \\ 0 \end{pmatrix} e_{2k}^T S_{2k} \ .$$

Postmultiplying once more by e_i we obtain

$$Cy_i - \theta_i y_i = \beta_{k+1}\begin{pmatrix} u_{k+1} \\ 0 \end{pmatrix} e_{2k}^T S_{2k} e_i \ .$$

The theorem follows by taking norms and noting that $\|u_{k+1}\|_2 = 1$. $\square$

Theorem 2 gives the following error bounds for the eigenvalue approximations

$$\min_{\mu\in\lambda(C)} |\mu - \theta_i| \leq \xi_{ki} \ , \tag{3.20}$$

(see, e.g. [29, Chapter 8]).

The *Ritz vectors* y_i, $i = 1, \ldots, 2k$ are approximations to the eigenvectors of C. Let $\tilde{u}_i \in \mathbb{R}^m$ and $\tilde{v}_i \in \mathbb{R}^n$ be defined such that

$$\begin{pmatrix} \tilde{u}_i \\ \tilde{v}_i \end{pmatrix} = \sqrt{2}\, y_i \ , \quad i = 1, \ldots, k \ ,$$

then according to Theorem 1 $\tilde{u}_i$ and $\tilde{v}_i$ are the Ritz vectors of A with respect to the subspaces $\mathcal{K}_j(AA^T, u_1)$ and $\mathcal{K}_j(A^TA, v_1)$, and thus provides us with approximations to the singular vectors of A. For completeness, we list the perturbation result corresponding to Theorem 2 for the Ritz vectors:

Theorem 3 *Let y_i be a Ritz vector with corresponding Ritz value θ_i as defined in Theorem 2 and let $\hat{\lambda}$ be the eigenvalue of C closest to θ_i, and let $\hat{v}$ be its normalized eigenvector. Define the gap*

$$\gamma_i \equiv \min_{\lambda \neq \hat{\lambda}} |\lambda - \theta_i| \tag{3.21}$$

then

$$\sin \angle(y_i, \hat{v}) \leq \frac{\|Cy_i - \theta_i y_i\|_2}{\gamma_i} \leq \frac{\xi_{ki}}{\gamma_i} \ . \tag{3.22}$$

Proof. See [51, Chapter 11]. □

The theorem shows that the convergence of a given Ritz vector is essentially like that of the corresponding Ritz value, except when the latter is an approximation to an eigenvalue which is in a cluster (this has nothing to do with the Lanczos algorithm being used, but simply reflects a well-known result about the conditioning of eigenvectors in the symmetric eigenvalue problem). We also mention that the gap-structure can be used to sharpen the error estimates in Theorem 2 to

$$\min\left(\frac{\xi_{ki}^2}{\gamma_i}, \xi_{ki}\right) \ ,$$

see, e.g., [51, Section 13.2] and [53]. In our implementation of the BPRO algorithm described below, we use these improved error estimates to reduce the necessary number of steps in the Lanczos bidiagonalization. In the following, however, we shall not go further into these technicalities, since they are essentially irrelevant to the issues discussed in this paper.

An important consequence of the discussion in the preceding paragraphs is that error bounds can be computed cheaply by using Equation (3.20): The constant β_{k+1} can be found by taking one extra step in the Lanczos process and $s_{2k\,i}$ is the last row in S_{2k}, which can be found without generating the whole of S_{2k}. Thus one can simply save the Lanczos vectors along the way and only when the convergence criterion has been fulfilled, does the entire S_{2k} need to be computed in order to calculate the Ritz vectors. This fact is mentioned in several texts [5, 55, 51, 53] and involves a small computational trick, which is crucial to get an efficient implementation: If the QR algorithm is used for computing θ_i, then S_{2k} a product of the series of orthogonal transformations G_1, $G_2, \ldots$ (plane rotations, or Householder reflectors) used to diagonalize T_{2k}:

$$G_p^T \cdots G_2^T G_1^T \, T_{2k} \, G_1 G_2 \cdots G_p = \mathrm{diag}(\theta_1, \ldots, \theta_{2k}) \ , \qquad S_{2k} = G_1 G_2 \cdots G_p \ .$$

In the process, S_{2k+1} is normally accumulated by starting with I_{2k} and postmultiplying by G_1, G_2 etc. as they are generated. If instead we start with e_{2k}^T, we end up with $e_{2k}^T G_1 G_2 \cdots G_p =$

$e_{2k}^T S_{2k} = (s_{2k\,1}, \ldots, s_{2k\,2k})$ which is exactly the last row of S_{2k}. In the LANSO package, for example, a modified version of subroutine TQL1 from EISPACK is used for this purpose. If bisection is used to compute the Ritz values then the last row of S_{2k} may be computed after the desired Ritz values have been found either by inverse iteration, or by using the following formula from Paige [48]

$$s_{2k\,i}^2 = \chi_{2k-1}(\theta_i)/\chi'_{2k}(\theta_i) \ ,$$

where $\chi_{2k}(\lambda)$ is the characteristic polynomial of T_{2k} (Beware: The latter approach is unstable if χ and χ' are evaluated using the simple three-term recurrences; a stable implementation is described by Parlett and Nour-Omid in [53, p. 206]). If the Lanczos bidiagonalization is used, error estimates may be generated as a byproduct when calculating the singular values of B_k (see below), e.g. using the bidiagonal SVD routine _BDSQR from LAPACK. In any case, the error bounds can be computed in $O(k^2)$ operations, which means that the convergence of the Ritz values can be monitored during the iteration without too much overhead. For further discussions see [53, 15].

We have not yet discussed *how* the Ritz values converge. This is a rather complicated matter, which depends on the properties of the spectrum of A. Kaniel [39] and Paige [48] were the first to give bounds on the rate of convergence of the Ritz values. These bounds, which have later been improved by Saad [58], show that while we can expect rapid convergence of the Ritz values approximating the extreme (the algebraically smallest and largest) eigenvalues of C, the interior eigenvalues will in general converge more slowly. In [58, Theorem 2] Saad proves the following result:

Theorem 4 *Let B be a symmetric $n \times n$ matrix with eigenvalues $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_n$ and corresponding orthonormal eigenvectors $z_1, z_2, \ldots, z_n$. If $\theta_1 > \theta_2 > \cdots > \theta_k$ are the eigenvalues of T_k obtained after k steps of the Lanczos iteration with starting vector q_1, then*

$$0 \leq \lambda_i - \theta_i \leq (\lambda_i - \lambda_n) \left(\frac{K_i^{(k)}}{P_{k-i}(\gamma_i)} \tan \angle(z_i, q_1) \right)^2 \ , \tag{3.23}$$

and the angle between z_i and the Krylov subspace $\mathcal{K}_k = (q_1, Bq_1, \ldots, B^{k-1}q_1)$ satisfies

$$\tan \angle(z_i, \mathcal{K}_k) \leq \frac{K_i^{(k)}}{P_{k-i}(\gamma_i)} \tan \angle(z_i, q_1) \ , \tag{3.24}$$

where P_{k-i} is the $(k-i)$th Chebyshev polynomial and

$$\begin{aligned} K_i^{(k)} &= \prod_{j=1}^{i-1} \frac{\theta_j - \lambda_n}{\theta_j - \lambda_i} \ , \qquad \text{if } i \neq 1 \ , \\ K_1^{(k)} &= 1 \ , \\ \gamma_i &= 1 + \frac{2(\lambda_i - \lambda_{i+1})}{\lambda_{i+1} - \lambda_n} \ . \end{aligned}$$

The Chebyshev polynomial P_j, which satisfies the recurrence $P_j(\gamma) = 2\gamma P_{j-1}(\gamma) - P_{j-2}(\gamma)$, is bounded by unity on the interval $[-1, 1]$, but grows exponentially in j outside. Thus provided that $|\gamma_i| > 1$ the largest eigenvalues (where $k - i$ is large) will converge rapidly. The rate with which $|\lambda_i - \theta_i|$ decays to zeros is also determined by the value of γ_i, which in turn depends on relative gap structure of the spectrum of B; if λ_i and λ_{i+1} are well separated, i.e. if $(\lambda_i - \lambda_{i+1})/(\lambda_{i+1} - \lambda_n)$ is not too small, then we can expect θ_i to converge rapidly, while close eigenvalues will converge more slowly. By applying the theorem to $-B$ we see that the algebraically smallest eigenvalues will also converge rapidly, while we cannot obtain any good bound on the eigenvalues in the interior part of the spectrum of B. For our matrix C from (3.1), this means that the Ritz values approximating the large singular values of A will converge rapidly, while the small singular values will converge more slowly, since they are in the middle of the spectrum of C. For further details including the corresponding convergence results for the Ritz vectors see, e.g., [58], [27] and [6, Section 7.6.4].

3.3.2 Connection to Lanczos bidiagonalization

After having reviewed the theoretical background for the symmetric Lanczos process and its application in SVD calculations, let us return to discuss its connection with the Lanczos bidiagonalization. If we perform an odd-even permutation of the rows and columns of

$$T_{2k+1} = \begin{pmatrix} 0 & \alpha_1 & & & & & \\ \alpha_1 & 0 & \beta_2 & & & & \\ & \beta_2 & 0 & \ddots & & & \\ & & \ddots & \ddots & \alpha_k & & \\ & & & \alpha_k & 0 & \beta_{k+1} \\ & & & & \beta_{k+1} & 0 \end{pmatrix} ,$$

generated after $2k+1$ Lanczos steps on C, we obtain the matrix

$$\begin{pmatrix} 0 & B_k \\ B_k^T & 0 \end{pmatrix} .$$

Permuting the columns of Q_{2k+1} accordingly, equation (3.17) can be written in the form

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} U_{k+1} & 0 \\ 0 & V_k \end{pmatrix} = \begin{pmatrix} U_{k+1} & 0 \\ 0 & V_k \end{pmatrix} \begin{pmatrix} 0 & B_k \\ B_k^T & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ \alpha_{k+1} v_{k+1} e_{k+1}^T & 0 \end{pmatrix} . \tag{3.25}$$

By equating the off-diagonal blocks on either side of the equation we arrive at (2.6) and (2.7), which describe the LBD process. In other words, the LBD is simply a compact form of the symmetric Lanczos process applied to C. Remembering the result from Theorem 1, this also tells us that the eigenvalues of T_{2k+1} are simply $\pm\sigma(B_k) \cup \{0\}$. The Ritz values and vectors can be found cheaply by computing the SVD of the bidiagonal matrix B_k:

$$\begin{aligned} \bar{P}^T B_k \bar{Q} &= \mathrm{diag}(\theta_1, \theta_2, \ldots, \theta_k) , \\ \tilde{u}_i = U_k \bar{p}_i , &\qquad \tilde{v}_i = V_k \bar{q}_i . \end{aligned} \tag{3.26}$$

The theorems describing error bounds and rates of convergence apply with minor modifications to the LBD, and in particular the error bounds in (3.20) become

$$\min_{\mu \in \sigma(A)} |\mu - \theta_i| \le |\alpha_{k+1}| \, |\bar{q}_{k+1\,i}| , \tag{3.27}$$

where $\bar{q}_{k+1\,i}$ are the elements in the last row of $\bar{Q}$.

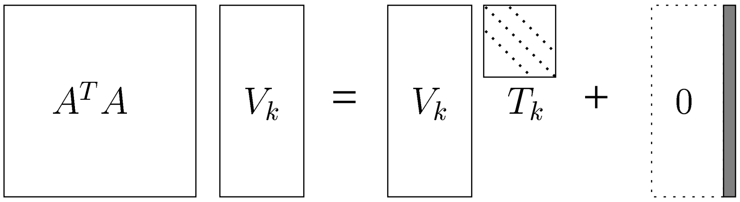


Figure 5: Lanczos tridiagonalization of A^TA. The shaded vector on the right-hand side is $\alpha_{k+1}\beta_{k+1}v_{k+1}$.

The LBD is also closely related to the symmetric Lanczos process applied to A^TA with starting vector $q_1 = A^Tu_1$, since this leads to a Lanczos tridiagonalization on the form

$$(A^TA)V_k = V_k\hat{T}_k + \alpha_{k+1}\beta_{k+1}v_{k+1}e_k^T \ ,$$

which is illustrated in Figure 5. In exact arithmetic we have that

$$\hat{T}_k = B_k^T B_k \ ,$$

and if the Schur decomposition of $\hat{T}_k$ is

$$\hat{S}_k^T \hat{T}_k \hat{S}_k = \mathrm{diag}(\hat{\theta}_1, \ldots, \hat{\theta}_k) \ ,$$

then $(\hat{\theta}_i)^{1/2}$, $i = 1, \ldots, k$ will be approximations to the k largest singular values of A and the Ritz vectors $V_k\hat{s}_i$, $i = 1, \ldots, k$ will be approximations to the corresponding right singular vectors v_i of A. If this approach is used, the left singular vectors u_i can subsequently be computed from the identity

$$Av_i = \sigma_i u_i \ .$$

It should be obvious from the discussion above that a sparse SVD program, which applies the symmetric Lanczos algorithm as a "black box" procedure to find the eigenvalues of C wastes a lot of resources manipulating the extra zeroes in the Lanczos vectors, while the LBD takes advantage of the special structure of C and the Lanczos vectors q_i, i.e. *i*) avoids to store the zero-parts of q_i, and *ii*) exploits that even- and odd-numbered Lanczos vectors are exactly orthogonal. This is especially important when using full or partial reorthogonalization, because here all the saved Lanczos vectors have to be manipulated in each step. In Table 2 we have listed the amount of storage used for the Lanczos vectors and the number of operations required by the reorthogonalization when computing approximations to the k largest singular values by means of either LBD with full reorthogonalization (BFRO) or the symmetric Lanczos algorithm with full reorthogonalization (FRO) on an equivalent symmetric matrix (we return to discuss reorthogonalization in Section 5.2). From the table one would expect

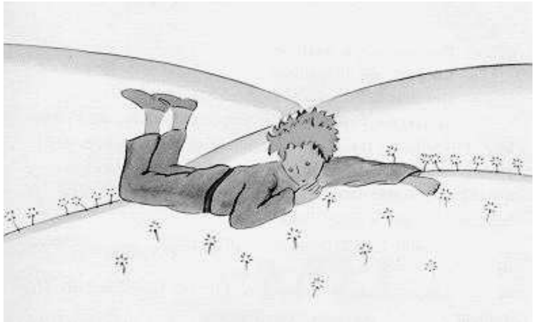
Table 2: *Words of storage and number of floating point operations used by different Lanczos algorithms with full reorthogonalization.*

	Storage	Work in reorth.
FRO(A^TA)	kn	$2k^2n$
FRO(C)	$2k(m+n)$	$8k^2(m+n)$
BFRO(A)	$k(m+n)$	$2k^2(m+n)$.

FRO(A^TA) to be twice as fast as BFRO(A) which would in turn be four times as fast as

FRO(C) when $m = n$. As we shall see in Section 7.2, this picture is changed when partial reorthogonalization is used, since in that case BPRO(A) and PRO(A^TA) do approximately the same amount of work when $m = n$ and both algorithms are about 3-4 times faster than PRO(C). This phenomenon, which is confirmed by our experiments with testmatrices from the Harwell-Boeing collection, is a result of the squaring of the singular values that occur when forming A^TA, which makes it necessary to reorthogonalize more frequently.

4 Sparse least squares



The cloud-capp'd towers, the gorgeous palaces,
The solemn temples, the great globe itself,
Ye all which it inherit, shall dissolve
And, like this insubstantial pageant faded,
Leave not a rack behind. We are such stuff
As dreams are made on, and our little life
Is rounded with a sleep.
— W. Shakespeare, from "The Tempest"

The Lanczos bidiagonalization described in Section 2 can also be applied to solving sparse linear least squares problems

$$\min \|A\,x - b\|_2 \ , \quad A \in \mathbb{R}^{m\times n} \ , \quad x \in \mathbb{R}^n \ , \quad b \in \mathbb{R}^m \ , \tag{4.1}$$

where $m > n$. Below we review the fundamental relations used to implement some frequently used methods, including the LSQR algorithm by Paige and Saunders [49, 50] and the hybrid algorithm based on Lanczos bidiagonalization which was discovered independently by Björck [5] and O'Leary and Simmons [46]. The latter is especially used in connection with regularization of ill-posed systems and for solving systems with many right-hand sides. The LSQR algorithm is usually carried out without reorthogonalization, but the convergence can be slowed down significantly by the loss of orthogonality. In the hybrid algorithm it is necessary to reorthogonalize to maintain stability when solving systems with many right-hand sides or when *generalized cross-validation* [25] is used for choosing the amount of regularization in connection with the solution of ill-posed problems. Both the simple GCV formula used in, e.g., [5], and the more advanced scheme based on implicit restarts proposed in [8] require that V_k and U_{k+1} are kept orthogonal. However, the algorithm from a more recent paper by Golub and von Matt [30], computes an approximation to the minimizer of the GCV function using LBD with no reorthogonalization. Golub and von Matt's algorithm uses the fact that the GCV function can be written as a matrix-moment and applies the theory of matrix-moments and Gauss-quadrature to compute upper and lower bounds on the GCV function. The algorithm terminates whenever the minimizer of the upper and lower bounds coincide within a preselected tolerance. However, as for LSQR, the convergence could be speeded up by the use of (partial) reorthogonalization.

Returning to the case when reorthogonalization is needed, the question arises how to implement it efficiently. In order to use, e.g., the symmetric Lanczos algorithm with partial reorthogonalization (PRO), one would have to work in terms of an equivalent symmetric system. This can be done by solving the augmented system

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix} ,$$

which unfortunately requires unnecessary work and storage since Lanczos vectors of length $m+n$ should be stored and orthogonalized. The alternative is to resort to solving the normal equations

$$A^T A\, x = A^T b \ ,$$

but that would greatly reduce the accuracy in the solution when the system is ill-posed as demonstrated in [49, 7]. In contrast, the BPRO algorithm described in Section 6 can be incorporated directly into the algorithms mentioned above while preserving the compact form of the LBD recurrences and without having to sacrifice numerical stability by resorting to the normal equations.

4.1 Least squares solvers based on LBD

We will now describe how the linear least squares problem in Equation (4.1) may be solved using LBD (our presentation is based on section 7.6.2 in [6]). If we take the starting vector p_0 equal to b then the recurrence relations (2.6) and (2.7) can be written

$$\left.\begin{array}{c} U_{k+1}(\beta_1 e_1) = b \ , \\ A\, V_k = U_{k+1}\, B_k \ , \qquad A^T\, U_{k+1} = V_k\, B_k^T + \alpha_{k+1} v_{k+1} e_{k+1}^T \ . \end{array}\right\} \tag{4.2}$$

We seek the vector x_k in $\mathcal{K}_k(A^T A, A^T b) = \mathrm{span}(V_k)$ that minimizes $\|A\,x - b\|_2$. The solution can be written in the form

$$x_k = V_k\, y_k \ , \qquad y_k \in \mathbb{R}^k \ ,$$

and substituting this into (4.1) and applying (4.2) we obtain

$$\begin{aligned} \min_{x\in\mathcal{K}_k} \|A\,x - b\|_2 &= \min_{y\in\mathbb{R}^k} \|A\,V_k\,y - b\|_2 \\ &= \min_{y\in\mathbb{R}^k} \|U_{k+1}(B_k\,y - \beta_1 e_1)\|_2 \end{aligned} \tag{4.3}$$

Using the orthogonality of U_{k+1} it follows that $\|A\,x_k - b\|_2$ is minimized in $\mathcal{K}_k$ by choosing y_k to be the solution to the least squares problem

$$\min_{y\in\mathbb{R}^k} \|B_k\,y - \beta_1 e_1\|_2 \ .$$

The LSQR algorithm by Paige and Saunders [49] is based on the relations described above, and is constructed in such a way that x_k is updated recursively from x_{k-1} without ever forming y_k. Most importantly, this makes the storage requirements of the algorithm small since it is no longer necessary to save the LAnczos vectors in V_k and U_k. In finite precision arithmetic the relations in (4.2) continue to hold with good accuracy, and the problem of minimizing $\|A\,x - b\|_2$ over $\mathrm{span}(V_k)$ is still almost equivalent to minimizing $\|B_k\,y_k - \beta_1 e_1\|_2$. However, the rate of convergence can be slowed down quite significantly due to the unavoidable loss of orthogonality, see [36, Section 6.4] for a detailed explanation. The stability of LSQR hinges on the special choice of starting vector $p_0 = b$ that ensures the relation $U_{k+1}\beta_1 e_1 = b$ to hold.

If a different starting vector is used we are forced to compute the solution as $x_k = V_k\, y_k$, where y_k is found by solving the least squares problem

$$\min_{y \in \mathbf{R}^k} \|B_k y - U_{k+1}^T\, b\|_2 \ .$$

The solution computed this way will no longer posses the minimizing property in Equation (4.3) and will furthermore be contaminated by round-off errors. This problem is solved by using reorthogonalization, which also speeds up convergence since no iterations are wasted on deflating components corresponding to extra copies of the Ritz values from the residual (see the discussion of "doppelgänger" singular values in Section 5).

4.2 Algorithms for problems with multiple right-hand sides

In a number of applications one cannot live with the loss of orthogonality, and some form of reorthogonalization must be applied. One example is when LBD is used to solve sparse linear least squares problems with multiple right-hand sides

$$\min \|A\, x^{(i)} - b^{(i)}\|_2\, , \quad i = 1, \ldots, N \ , \tag{4.4}$$

which arise, e.g., when implementing the so-called SOLA mollifier method used in helioseismic inversion (see [56, 42]). A straightforward solution would be to apply the LSQR algorithm to each system independently, but often the following procedure (which was analyzed by Saad [59] in the context of (square) linear systems) is more efficient:

1. Solve the first system by computing k steps of the LBD with starting vector $b^{(1)}$.

2. Project the remaining systems onto the generated Krylov subspaces to form the approximate solutions
$$x_k^{(i)} = V_k\, y_k^{(i)} \ , \quad i = 2, \ldots, N \tag{4.5}$$
where $y_k^{(i)}$ solves the least squares problem
$$\min_{y \in \mathbf{R}^k} \|B_k y - U_{k+1}^T\, b^{(i)}\|_2 \ . \tag{4.6}$$

After the approximate solutions have been computed, one can select the residual of one of the unconverged systems as a new starting vector for the LBD and repeat steps 1. and 2., and repeat this process until all systems have converged. Another possibility is to terminate this process after a few repetitions and use the approximate solutions as starting guesses to an ordinary iterative least squares solver such as LSQR, which is then applied independently (and possibly in parallel) to each of the unconverged systems. Notice that once the LBD has been computed, the approximate solutions can be found in step 2. without accessing the original matrix A.

When reorthogonalization is used $x_k^{(i)}$ minimizes the residual of the ith system over the generated Krylov subspace $\mathcal{K}_k(A^T A, A^T b^{(1)})$, so $x_k^{(1)}$ is almost identical to the solution obtained after k steps of the LSQR algorithm in exact arithmetic. The accuracy of the approximate solutions $x_k^{(i)}$, $i =, 2, \ldots, N$ computed via (4.5) and (4.6) depends on the distance from $b^{(i)}$ to $\mathrm{span}(U_{k+1})$. If we decompose $b^{(i)}$ into orthogonal components

$$b^{\|} = U_{k+1}\, U_{k+1}^T\, b^{(i)} \in \mathrm{span}(U_{k+1}) \quad \text{and} \quad b^{\perp} = (b^{(i)} - b^{\|}) \perp \mathrm{span}(U_{k+1}) \ ,$$

then it follows that

$$\|A\,x_k^{(i)} - b^{(i)}\|_2^2 = \|A\,x_k^{(i)} - b^{||}\|_2^2 + \|b^{\perp}\|_2^2 \qquad (4.7)$$

since we know from (4.2) and (4.5) that $A\,x_k^{(i)} = U_{k+1}B_k y_k^{(i)} \in \text{span}(U_{k+1})$. Now the speed of convergence depends on the size of the two term. Let us follow Saad [59] and first consider the two extreme cases. If $b^{(i)}$ is in $\text{span}(U_{k+1})$ then the second term on the right-hand side in (4.7) vanishes, and the method provides an accurate approximation. Typically, the size of the first term is "small", i.e. of the same order as the residual norm obtained after k steps of LSQR with starting vector $b^{(i)}$. In the other extreme when $b^{(i)}$ is orthogonal to $\text{span}(U_{k+1})$ (and therefore orthogonal to $b^{(1)}$) then it follows from (4.7) that $x^{(i)} = 0$, i.e. the projection did not improve the solution of system i.

In practice $\|b^{\perp}\|_2$ will be small compared to $\|A\,x_k^{(i)} - b^{||}\|_2$ as long as the remaining right-hand sides $b^{(i)}$ are not too different from $b^{(1)}$ and consequently $x_k^{(i)}$ will converge almost as rapidly as $x_k^{(1)}$. This will for instance often be the case for discrete ill-posed problems when the right-hand sides satisfy the discrete Picard criterion (see [35]). In this case the vectors $b^{(i)}$ will be dominated by components lying in a subspace spanned by the singular vectors corresponding to the largest singular values. In many applications the dimensions of the dominant subspace (often called the *signal subspace*) is usually small compared to the dimension of the problem. In this case, the Lanczos bidiagonalization with starting vector $b^{(1)}$ will generate Krylov subspaces that are also close to optimal for the systems corresponding to the remaining right-hand sides. Hence, even $\|b^{\perp}\|_2$ is not negligible it will often be dominated by noise components which should be removed by the regularization anyway.

When the right-hand sides are not too different, the approach just described is much cheaper in terms of matrix-vector multiplications than naively applying LSQR to the p systems one at a time. However, as already mentioned, it requires that V_k and U_{k+1} are kept orthogonal if the solutions computed using (4.5) and (4.6) are to have the minimizing property. The relative cost of reorthogonalization and matrix-vector multiplication determines which approach is the more efficient in practice. A more quantitative description of the convergence properties including bounds on the residual norms $\|A\,x_k^{(i)} - b^{(i)}\|_2$, $i = 1, \ldots, N$ after k steps can be derived from the results in [59].

There are a number of alternative ways to implement the bidiagonalization-projection process outlined above. In a future paper we intend to investigate the possibility of generalizing a number of the algorithms developed for solving symmetric systems of equations with multiple right-hand sides to the least squares case. These methods include the modified Lanczos process discovered by Parlett in [52] and analyzed by Saad in [59].

One such algorithm, which we have started investigating, is a generalization of the Lanczos-Galerkin projection method described in [13, 66]. This algorithm, which we will refer to as MCG (Multi-CG) in the following, applies the conjugate gradient algorithm to construct a sequence of Krylov subspaces. The same idea can be applied to the CGLS algorithm (e.g. [6, Section 7.4]) to obtain a stable Lanczos-Galerkin projection method for solving least squares problems with multiple right-hand sides. The resulting algorithm, which we call MCGLS (or Multi-CGLS) has the advantage that it does *not* require any reorthogonalization. The algorithm generates a pair of Krylov subspaces from the set of (search) direction vectors obtained by solving the system corresponding to $b^{(1)}$, called the *seed system*, using the CGLS algorithm. Then it projects the residuals $r^{(i)} = b^{(i)} - A\,x^{(i)}$, $i = 2, \ldots, N$ of the other systems orthogonally onto the generated Krylov subspaces to get the approximate solutions. The projections can be computed cheaply and involve only a single inner product and two vector updates per right-hand-side per iteration. The operations on each of the remaining right-hand sides can furthermore be executed completely in parallel.

The MCGLS algorithm takes the following form:

for $k = 1, 2, \ldots, N$ **do**
  $r_0^{(k)} = b^{(k)} - Ax_0^{(k)}$
end
for $k = 1, 2, \ldots, N$
  Select the kth system as seed:
  $p_0 = s_0 = A^T r_0^{(k)}$
  $\gamma_0 = \|s_0\|_2^2$
  for $i = 0, 1, 2, \ldots$ **while** $\gamma_i >$ `tol` **do**
    Take a CGLS step:
    $q_i = Ap_i$
    $\alpha_i = \gamma_i / \|q_i\|_2^2$
    $x_{i+1}^{(k)} = x_i^{(k)} + \alpha_i p_i$
    $r_{i+1}^{(k)} = r_i^{(k)} - \alpha_i q_i$
    $s_{i+1} = A^T r_{i+1}^{(k)}$
    $\gamma_{i+1} = \|s_{i+1}\|_2^2$
    $\beta_i = \gamma_{i+1} / \gamma_i$
    $p_{i+1} = s_{i+1} + \beta_i p_i$
    Perform Galerkin projection:
    for $j = k+1, k+2, \ldots, N$ **do** (each remaining unsolved RHS)
      $\eta_j = \alpha_i / \gamma_i (p_i^T r_0^{(j)})$
      $x_0^{(j)} = x_0^{(j)} + \eta_j p_i$
      $r_0^{(j)} = r_0^{(j)} - \eta_j q_i$
    end
  end
end

Notice, that as in the CGLS steps executed for the seed system, the true residuals for the non-seed systems are also recurred directly, and hence no serious loss of information (see the discussion in [7]) occurs in the projection phase. Since this is the key to the stability of CGLS we suspect the algorithm to be able to be solve all the systems just as accurately as by applying CGLS to each system independently. This will also be the topic of future investigations.

The MCGLS algorithm also has a block (multi-seed) generalization, based on the block conjugate gradient algorithm by O'Leary [45] applied to the normal equations. Chan and Wan discuss a similar block-MCG algorithm, and show that it can be superior to the single-seed version in terms of convergence rate when solving linear systems with multiple right-hand sides. However, as the original block-CG algorithm it is liable to so-called *breakdowns* (see, e.g., [10]), and convergence cannot be guaranteed even when the matrix is well-conditioned (something which we have also observed on a few occasions for block-MCGLS).

With respect to solving ill-posed problems, it is interesting (but not surprising) that by applying Lemma 3.2 from the paper by Chan and Wan [13] to the normal equations, it can

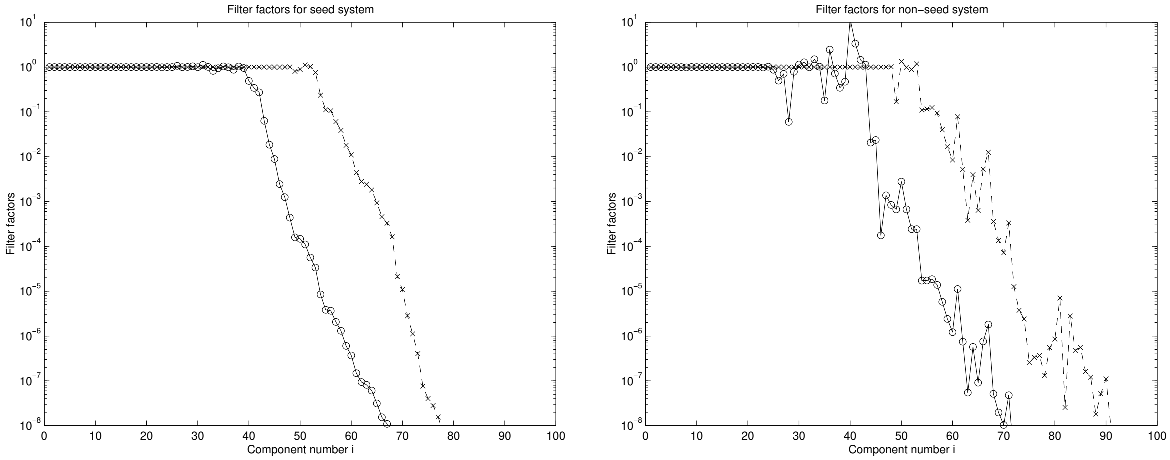


Figure 6: Filter factors in the MCGLS algorithm. Left panel shows the filter factors for the seed system with (crosses) and without (circles) reorthogonalization. The right panel shows the filter factors for a non-seed system.

be shown that the approximate solutions to the non-seed systems are essentially spanned by the singular vectors corresponding to the largest singular values. Hence the method has an intrinsic regularizing effect, which is also confirmed by our preliminary investigations. This is illustrated in Figure 6 where we have executed the MCGLS algorithm shown above for 50 iteration on a least-squares problem with multiple right-hand sides that arises in the so-called SOLA method used in helioseismic inversion (see [42]); the coefficient matrix is in fact the testmatrix HELIO212b used elsewhere in this report. The figure illustrates the so-called *filter factors* f_i for the seed system (left panel) and the first non-seed system, and they show how the different SVD-components contribute to the solution:

$$x_k^{(i)} = \sum_{i=1}^{n} f_i \frac{u_i^T b^{(i)}}{\sigma_i} v_i \ ,$$

see [36] for a further discussions of filter factors and the regularizing effect of the CGLS algorithm. In the figure, circles show the filter factors for MCGLS without reorthogonalization and crosses show filter factors when reorthogonalization is used; in CGLS the reorthogonalization can be implemented by keeping the residual vectors, s_i, of the normal equations orthogonal using one of the methods discussed in Section 5.2 (it should be straightforward to derive a partial reorthogonalization scheme, which is tailored specifically for (M)CGLS, using techniques similar to those in Section 6 – more future work!). Without reorthogonalization, the behavior is a little erratic and it remains to be seen if the regularized solutions for the non-seed systems computed thus are acceptable in general. We have made a few experiments with systems arising in large-scale 2-dimensional helioseismic rotational inversions (see e.g. [61, 41]), which seem to indicate that the computed solutions are quite acceptable, but many extra iterations are required for systems where the right-hand side looks very different from $b^{(1)}$ when no reorthogonalization is used. This is also what one would expect from the discussion above.

It is our general impression that while methods for solving linear systems with multiple right-hand sides have received some attention in (cf. [63, 59, 52, 66, 13]), the questions discussed in this section in relation to the solution of sparse linear least squares problems with multiple right-hand sides have received little attention so far, maybe except for a few remarks in [5, 46].

5 Lanczos bidiagonalization in finite precision arithmetic

Knabe sprach: Ich breche dich,
Röslein auf der Heiden!
Röslein sprach: Ich steche dich,
Daß du ewig denkst an mich,
Und ich will's nicht leiden.
Röslein, Röslein, Röslein rot,
Röslein auf der Heiden.

When the Lanczos bidiagonalization is carried out in finite precision arithmetic Equations (2.4) and (2.5), which describe the central steps in the computation, become

$$\alpha_j v_j = A^T u_j - \beta_j v_{j-1} + f_j \ , \tag{5.1}$$
$$\beta_{j+1} u_{j+1} = A v_j - \alpha_j u_j + g_j \ , \tag{5.2}$$

where $f_j \in \mathbb{R}^n$ and $g_j \in \mathbb{R}^m$ are error vectors accounting for the rounding errors at the jth step (from now on u, v, α, β and so on, will refer to the computed quantities). Usually the rounding terms are small, so after k steps (2.6) and (2.7) still hold to almost machine accuracy. In contrast, the orthogonality among the left and right Lanczos vectors is gradually lost such that equations (2.2) and (2.3) no longer hold.

The loss of orthogonality goes hand in hand with the convergence of Ritz pairs in a very systematic way: A now famous result by Paige [48] shows that after $2k$ steps, say, of the symmetric Lanczos process, the newly generated Lanczos vector q_{2k+1} satisfies the following relation (using the notation of Theorem 2):

$$|q_{2k+1}^T y_i| \approx \frac{\mathbf{u}\|C\|_2}{|\beta_{k+1}|\,|s_{2k,i}|} \ .$$

If we recall Equation (3.19)

$$\|Cy_i - \theta_i y_i\|_2 = |\beta_{k+1}||s_{2k,i}| \ ,$$

and Equation (3.20), which describe the error in the Ritz value approximation corresponding to y_i, this can be interpreted in the following way: As a result of rounding errors the generated Lanczos vectors tend to have unwanted large components in the direction of any converged Ritz vector.

Another remarkable feature of the Lanczos algorithm is that the accuracy of the converged Ritz values is not affected by the loss of orthogonality, but while the large singular values of B_k are still accurate approximations to the large singular values of A, the spectrum of B_k will in

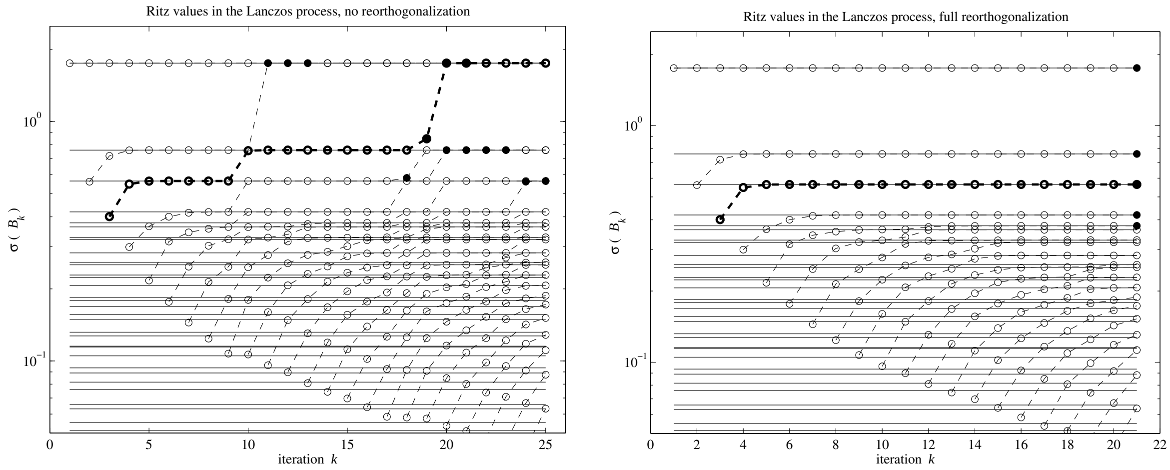


Figure 7: *Illustration of the convergence of Ritz values in the Lanczos bidiagonalization process in finite precision arithmetic. Horizontal lines indicate the true values of the singular values and circles show the Ritz values* $\theta_i^{(k)} = \sigma_i(B_k)$. *The lower and upper panel show the behavior with and without reorthogonalization respectively.*

addition contain false multiple copies of converged Ritz values, which we will call *doppelgänger* singular values in the following; this happens even if the corresponding true singular value of A is isolated. Moreover, spurious singular values, so-called *ghosts*, periodically appear between the already converged Ritz values.

To illustrate the effect of finite precision arithmetic on the convergence properties, we have used the Lanczos bidiagonalization to compute approximations to the 5 largest singular values of testmatrix HELIO212b with a relative error of 10^{-12} (error estimates were computed using (3.20)). The results are illustrated in Figure 7, where horizontal lines indicate the value of the true singular values and circles show the Ritz value approximation obtained after each iteration. Circles marking the same Ritz value $\theta_i^{(k)}$ at different iterations $k = 1, 2, \ldots$ have been connected by dashed lines to show how doppelgänger and ghost values that lie between the true singular values occur during the iteration. The ghosts, marked by a filled circle, were detected using to the criterion mentioned in Section 5.1 below.

In the left panel, which shows the behavior without reorthogonalization, we see the interesting phenomenon that some of the converged Ritz values suddenly "jump" to become a ghost and then converge to the next larger singular value after a few iterations. One example is θ_3 (the fat line), which first converges to σ_3 at $k \approx 8$. Then at $k = 10$ it suddenly jumps to become an approximation to σ_2, becomes spurious again at $k = 19$ and eventually settles as the second doppelgänger of σ_1 at $k = 22$. In this process, many iterations are wasted on computing false copies of the extreme singular values and for this reason 26 iterations are needed to achieve convergence while in the right panel, where full reorthogonalization was used, only 21 iterations was needed. With reorthogonalization the convergence behavior is also much simpler; it can be shown that only one Ritz value converges to any given singular value of A and no ghosts appear. Notice in both panels how the overall pattern shows the well separated extreme Ritz values converging most rapidly as predicted by Theorem 4. We refer to [15, 55, 51, 64], where effects of finite precision arithmetic are discussed in great detail.

5.1 Lanczos algorithms with no reorthogonalization

Du Doppelgänger, du bleicher Geselle!
Was äffst du nach mein Liebesleid,
das mich gequält auf dieser Stelle
so manche Nacht, in alter Zeit?

— Heinrich Heine

There exist two different “schools” with respect to what should be done to obtain a robust method in finite precision arithmetic. One approach which has been advocated by Paige, Cullum and Willoughby, among others, is to apply the simple Lanczos process as it is, and subsequently use some criterion to weed out the ghost and doppelgänger eigenvalues. The criterion used in [15] for the symmetric Lanczos process, is based on the curious fact that the spurious eigenvalues of T_{2k+1} are nearly eigenvalues of $\hat{T}$, which is the matrix T_{2k+1} with the first row and the first column removed. The procedure is to remove eigenvalues λ_i where

$$\min_{\mu \in \lambda(\hat{T})} |\lambda_i - \mu| < \texttt{tol}$$

For the Lanczos bidiagonalization this corresponds to removing those those singular values of B_k which are close to a singular value of the upper bidiagonal matrix

$$\hat{B} = \begin{pmatrix} \beta_2 & \alpha_2 & & \\ & \beta_3 & \ddots & \\ & & \ddots & \alpha_k \\ & & & \beta_{k+1} \end{pmatrix},$$

which is simply B_k with the first row removed.

The advantage of this approach is that it completely avoids the extra work associated with the reorthogonalization schemes described below, and the storage requirements are very low since only a few of the latest Lanczos vectors have to be remembered. The disadvantage is that many iterations are wasted on simply generating multiple copies of the large Ritz values, or as Parlett puts it in [51, pp. 262]: “Its fault is in not quitting while it is ahead because it continues to compute many redundant copies of each Ritz pair”. The number of extra iterations required compared to executing the Lanczos process in exact arithmetic can be very large. Up to six times the original number has been reported in e.g. [54]. This is in agreement with our own experiences from solving highly ill-conditioned problems in inverse helioseismology [41] using algorithms based on Lanczos bidiagonalization. Here many iterations are also wasted generating multiple copies of a few large isolated singular values, and a reduction in the number of iterations between a factor of 5 and 10 is typically observed when using reorthogonalization. Another disadvantage is that the criterion mentioned above can be rather difficult to implement, and its success depends on the correct choice of various tolerance parameters involved.

N	k (no reorth.)	k (full reorth.)
5	26	21
10	40	30
20	61	38
40	113	46
45	223	51

The large number of extra iterations required is illustrated in the table above, where we have listed the number of iterations k required to compute N singular values of HELIO212b with and without reorthogonalization. As long as only a few of the extreme singular values are required, the overhead is not very large. But the larger N the larger the overhead: As can be seen in Figure 3, the singular values start to drop off quickly at $i \approx 40$, and it turns out that the number of "wasted" iteration increases dramatically when N is increased beyond this point as is clearly seen in the table above; with reorthogonalization this behavior is not seen.

5.2 Stabilizing Lanczos using reorthogonalization

A different way of "stabilizing" the simple Lanczos bidiagonalization process is to enforce orthogonality among the Lanczos vectors by applying some *reorthogonalization* scheme. The simplest is to use full reorthogonalization (FRO) where each new Lanczos vector u_{j+1} is orthogonalized against all previous u_i, $i = 1, \ldots, j$ using, e.g., the Modified Gram-Schmidt algorithm (and similarly for the right Lanczos vector v_j). With this modification, the LBD algorithm becomes:

1. Choose a starting vector $p_0 \in \mathbb{R}^m$, and let
   $\beta_1 = \|p_0\|_2$, $u_1 = p_0/\beta_1$ and $v_0 \equiv 0$
2. **for** $j = 1, 2, \ldots$ **do**
   - $r_j = A^T u_j - \beta_j v_{j-1}$
   - **for** $i = 1, \ldots, j-1$ **do**
     - $r_j = r_j - (v_i^T r_j) v_i$
   - $\alpha_j = \|r_j\|_2$
   - $v_j = r_j / \alpha_j$
   - $p_j = A v_j - \alpha_j u_j$
   - **for** $i = 1, \ldots, j$ **do**
     - $p_j = p_j - (u_i^T p_j) u_i$
   - $\beta_{j+1} = \|p_j\|_2$
   - $u_{j+1} = p_j / \beta_{j+1}$

   end

This is usually considered too expensive for large problems where the additional $O(4(m+n)k^2)$ operations required by the reorthogonalization quickly dominate the execution time, unless the necessary number of iterations k is very small compared to the dimensions of the problem. The storage requirements of FRO may also be a limiting factor, since all the generated Lanczos vectors have to be saved. Moreover, when new Lanczos vectors are reorthogonalized the program sweeps through all the previous ones. When solving large problems, the Lanczos vectors typically require too much storage to fit in fast (cache-) memory and must be recalled from either main memory or secondary storage in every iteration. This will cause the performance of FRO to be poor on machines with a memory hierarchy.

An alternative implementation of reorthogonalization was developed by Golub, Underwood and Wilkinson [28], who suggested that U_{k+1} and V_k be represented as a product of Householder reflectors. If the reflectors are accumulated as so-called block reflectors (see [60]), this is probably the most efficient way of implementing full reorthogonalization. However, this representation is not possible in connection with the partial reorthogonalization schemes treated in the remaining part of this paper, and we will not discuss the Householder approach any further.

For the symmetric Lanczos method, a number of different schemes for reducing the work associated with keeping the Lanczos vectors orthogonal have been developed by B. N. Parlett and co-workers at the University of California at Berkeley, see e.g. [55, 63]. The goal of these methods is to cut down the number of orthogonalizations yet obtain a Lanczos algorithm that computes a result, which is close to what would have been computed in the absence of rounding errors.

If we define the level of orthogonality κ_j^u (κ_j^v) among the left (right) Lanczos vectors at the jth step as

$$\kappa_j^u = \max_{1\le i\le j-1} |u_j^T u_i| \ , \qquad \kappa_j^v = \max_{1\le i\le j-1} |v_j^T v_i| \tag{5.3}$$

then full reorthogonalization keeps κ_j^u and κ_j^v at the level of the machine round-off $\mathbf{u}$. All this work is not necessary. It was found in [55, 51, 64] that it is sufficient to maintain *semiorthogonality*, that is to keep κ_j^u and κ_j^v below $\sqrt{\mathbf{u}/k}$, to obtain accurate approximations to the singular values and avoid ghosts and doppelgängers from appearing.

Theorem 5 *Let B_k be the bidiagonal matrix computed after k steps of the Lanczos bidiagonalization algorithm where the Lanczos vectors are kept semiorthogonal. If*

$$\kappa_j^u,\ \kappa_j^v \le \sqrt{\frac{\mathbf{u}}{k}}, \qquad for \quad 1 \le j \le k \ ,$$

and the columns of $\hat{U}_{k+1}$ form an orthonormal basis for $\mathrm{span}(U_{k+1})$*, and the columns of $\hat{V}_k$ form an orthonormal basis for* $\mathrm{span}(V_k)$*, then*

$$\hat{U}_{k+1}^T A \hat{V}_k = B_k + E_k \tag{5.4}$$

where the elements of E_k are of order $O(\mathbf{u}\|A\|_2)$.

Proof. Since the Lanczos bidiagonalization is equivalent to applying the symmetric Lanczos algorithm to the symmetric matrix C in (3.1), the result follows directly from [64, Theorem 4]. □

The theorem says that if the Lanczos vectors are just kept semiorthogonal, then the computed B_k is up to roundoff the Ritz-Galerkin projection of A on the subspaces $\mathrm{span}(U_{k+1})$ and $\mathrm{span}(V_k)$. The subspaces are different from the optimal ones computed in exact arithmetic, but in practice that has not been observed to affect the convergence or final accuracy of the Ritz-values.

6 A Partial Reorthogonalization algorithm for LBD

Und der wilde Knabe brach
's Röslein auf der Heiden;
Röslein wehrte sich und stach,
Half ihm doch kein Weh und Ach,
Mußt' es eben leiden.
Röslein, Röslein, Röslein rot,
Röslein auf der Heiden.

The central idea in partial reorthogonalization is that the level of orthogonality among the Lanczos vectors satisfies a recurrence relation that can be derived from the recurrence used to generate the vectors themselves. This was already shown by Paige in his pioneering thesis [48]. It was Simon [63, 64], however, who realized that these recurrences can be used as a practical tool for computing estimates of the level of orthogonality in an efficient way, and devised scheme by which this information can be used to decide when to reorthogonalize, and which Lanczos vectors it is necessary to include in the reorthogonalization.

Below we present a result, which introduces the LBD equivalent of the ω-recurrence derived by Simon in [63] for the symmetric Lanczos process:

Theorem 6 *Let $\mu_{ji} \equiv u_j^T u_i$ and $\nu_{ji} \equiv v_j^T v_i$. Then μ_{ji} and ν_{ji} satisfy the following coupled recurrences:*

$$\begin{aligned} \mu_{ii} &= 1 , \qquad \textit{for} \quad 1 \le i \le j+1 , \\ \beta_{j+1}\mu_{j+1,i} &= \alpha_i \nu_{ji} + \beta_i \nu_{j,i-1} - \alpha_j \mu_{ji} + u_i^T g_j - v_j^T f_i, \end{aligned} \tag{6.1}$$

for $1 \le i \le j$, and $\mu_{i0} \equiv 0$,

$$\begin{aligned} \nu_{ii} &= 1 , \qquad \textit{for} \quad 1 \le i \le j , \\ \alpha_j \nu_{ji} &= \beta_{i+1}\mu_{j,i+1} + \alpha_i \mu_{ji} - \beta_j \nu_{j-1,i} - u_j^T g_i + v_i^T f_j \end{aligned} \tag{6.2}$$

for $1 \le i \le j-1$, and $\nu_{i0} \equiv 0$.

Proof. Write (5.2) and (5.1) for j and i to obtain

$$\beta_{j+1} u_{j+1} = A v_j - \alpha_j u_j + g_j \tag{6.3}$$

$$\beta_{i+1} u_{i+1} = A v_i - \alpha_i u_i + g_i \ . \tag{6.4}$$

and

$$\alpha_j v_j = A^T u_j - \beta_j v_{j-1} + f_j \tag{6.5}$$

$$\alpha_i v_i = A^T u_i - \beta_i v_{i-1} + f_i \ , \tag{6.6}$$

Premultiply (6.4) by u_j^T, subtract this from (6.5) premultiplied by v_i^T, and simplify to obtain (6.2). Similarly, subtract (6.6) premultiplied by v_j^T from (6.3) premultiplied by u_i^T, and simplify to obtain (6.1). $\square$

We now give a general outline of BPRO. In light of Theorem 5 it will be sufficient to reorthogonalize the Lanczos vectors whenever κ^u_{j+1} or κ^v_j exceeds $\sqrt{\mathbf{u}/k}$. We cannot check this condition directly without forming inner products of the latest Lanczos vector with all previous ones, and this involves almost the same amount of work as full reorthogonalization. However, Theorem 6 says that the inner products $u_i^T u_{j+1}$ and $v_i^T v_j$ are simply linear combinations of inner products from the previous Lanczos step, and therefore updating estimates $\mu_{j+1,i}$ and ν_{ji} from one step to the next only involves manipulating two vectors of length $j+1$ and j respectively. These estimates can then be used to determine when and against which of the previous Lanczos vectors to reorthogonalize. The resulting algorithm BPRO takes the following form:

1. Choose a starting vector $p_0 \in \mathbb{R}^m$, and let
   $\beta_1 = \|p_0\|_2, \; u_1 = p_0/\beta_1$ and $v_0 \equiv 0$
2. **for** $j = 1, 2, \ldots, k$
   - $r_j = A^T u_j - \beta_j v_{j-1}$
   - 2.1a Update $\nu_{j-1,i} \rightarrow \nu_{ji}$, $\quad i = 1, \ldots, j-1$
   - 2.1b Determine a set of indices $\mathcal{L}^\nu_j \subseteq \{i \,|\, 1 \le i \le j-1\}$
     for $i \in \mathcal{L}^\nu_j$
       $r_j = r_j - (v_i^T r_j)\, v_i$
       Reset $\nu_{j,i}$ to $O(\mathbf{u})$
     end
   - $\alpha_j = \|r_j\|_2$
   - $v_j = r_j/\alpha_j$
   - $p_j = A v_j - \alpha_j u_j$
   - 2.2a Update $\mu_{j,i} \rightarrow \mu_{j+1,i}$, $\quad i = 1, \ldots, j$
   - 2.2b Determine a set of indices $\mathcal{L}^\mu_{j+1} \subseteq \{i \,|\, 1 \le i \le j\}$
     for $i \in \mathcal{L}^\mu_{j+1}$
       $p_j = p_j - (u_i^T p_j)\, u_i$
       Reset $\mu_{j+1,i}$ to $O(\mathbf{u})$
     end
   - $\beta_{j+1} = \|p_j\|_2$
   - $u_{j+1} = p_j/\beta_{j+1}$

   end

The outline above is still far from a full algorithmic description of the method. In the two paragraphs below we discuss the details involved in implementing steps 2.1a, 2.1b, 2.2a and 2.2b, which are crucial to turn BPRO into a robust and efficient algorithm.

6.1 Tracking the loss of orthogonality

In steps 2.1a and 2.2a, we intend to use equations (6.1) and (6.2) to update the estimates $\mu_{j+1,i}$ and ν_{ji} from their values in the previous Lanczos step (in the following μ and ν will refer to the estimates computed using the updating rules above and not their Platonic images used in Theorem 6). To do this we need an estimate of the size of the round-off terms on the form $u_i^T g_j - v_j^T f_i$. In practice f_i and g_j are unknown, but we can give an upper bound on their size based on the forward error bound for the matrix-vector multiplication. From (5.2) and (5.1) we see that the elements in f_i (g_j) mainly come from the rounding errors that occur when the inner product of a column (row) of A with u_i (v_j) is accumulated. This gives us the following simple bounds: $\|f_i\|_2 \leq m\,\mathbf{u}\,\|A\|_2$, $\|g_j\|_2 \leq n\,\mathbf{u}\,\|A\|_2$, which are quite pessimistic in practice. Instead we use the rule of thumb that the mean rounding error is proportional to the square root of the number of arithmetic operations. This is justified by regarding the rounding errors as independent random variables and applying the central limit theorem[2]. Hence we use the estimate:

$$|u_i^T g_j - v_j^T f_i| \approx \max(m,n)^{1/2} \cdot \mathbf{u} \cdot \|A\|_2 \equiv \epsilon_1 \|A\|_2 \ .$$

This gives the following updating rules for μ_{j+1i} and ν_{ji}:

$$\left.\begin{array}{rcl} \mu'_{j+1,i} & = & \alpha_i \nu_{ji} + \beta_i \nu_{j,i-1} - \alpha_j \mu_{ji} \\ \mu_{j+1,i} & = & (\mu'_{ij} + \text{sign}(\mu'_{ij})\epsilon_1)/\beta_{j+1} \end{array}\right\} \quad \text{for} \quad 1 \leq i \leq j \ , \tag{6.7}$$

and $\mu_{j+1,j+1} = 1$,

$$\left.\begin{array}{rcl} \nu'_{ij} & = & \beta_{i+1}\mu_{j,i+1} + \alpha_i \mu_{ji} - \beta_j \nu_{j-1,i} \\ \nu_{ij} & = & (\nu'_{ij} + \text{sign}(\nu'_{ij})\epsilon_1)/\alpha_j \end{array}\right\} \quad \text{for} \quad 1 \leq i \leq j-1 \ , \tag{6.8}$$

and $\nu_{jj} = 1$. Here $\nu_{i0} \equiv 0$ and $\mu_{i0} \equiv 0$.

In Figure 8 we have shown the result computed with the updating rules for two of the testproblems from Section 7.3. We notice that the computed estimates describe the qualitative behavior of the true level of orthogonality quite well. Even for the smaller test problem HELIO212b, where the size of the rounding errors is clearly overestimated, μ_{ij} reaches the limit $\sqrt{\mathbf{u}/k}$ only one step too early. By overestimating the level of orthogonality the method will perform more reorthogonalizations than strictly necessary. On the other hand, if the level of orthogonality is underestimated by using too optimistic updating rules, the semiorthogonality requirement could be violated and cause spurious singular values to be computed. As reported in Section 7 we have used the resulting method on a variety of problems, and have seen no examples of failure to keep the Lanczos vectors semiorthogonal when the conservative rules above are used; the method seems to be robust.

As an experiment we tried to use a more optimistic updating rule with $\epsilon_1 = \mathbf{u}$ and the recurrences still produced slight overestimates of $\mu_{j+1,i}$ and ν_{ji}, except for the two large test matrices QH1484 and MHD4800a (see Table 6). The amount of work in the reorthogonalization was reduced from 12% (HELIO212b) up to 38% (RBS480a) by using the optimistic rule, and the largest savings occurred in well-conditioned problems where the reorthogonalizations were reduced to a fairly small number. However, until we have a better understanding of this behavior, we stick to the conservative bound above, which was also used for the experiments reported in Section 7. It should be emphasized that even with the conservative rule, the reduction in the amount of work compared to full reorthogonalization is quite significant.

If additional information about the matrix is available this should be used to get tighter bounds on the size of the round-off terms, and thus reduce the amount of reorthogonalization. In the case of a sparse matrix, such information could be the maximum number of non-zero

[2]Keeping in mind, yet ignoring (which can be achieved using the technique commonly known as Doublethink, see the appendix of [47]) the fundamental improbability (or doubleplusungodness) of such probabilistic error analyzes, as pointed out in the very thought-provoking notes by W. Kahan [38], we shall nonetheless proceed to use this estimate.

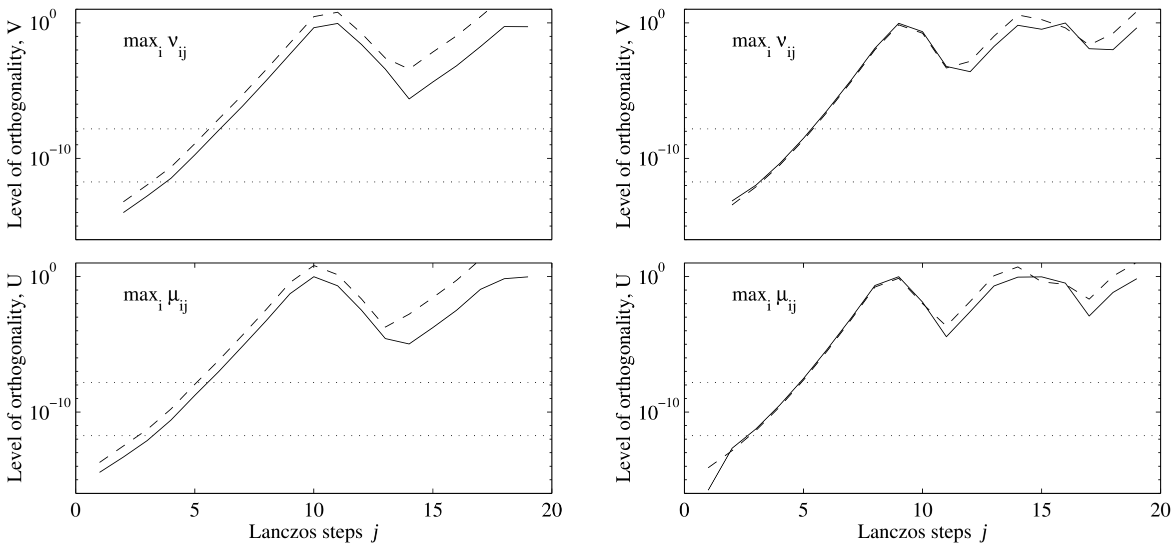


Figure 8: *Illustration of the recurrence in Theorem 6. The estimated (dashed) and true (solid) levels of orthogonality as a function of the number of iterations j. The left figure shows the results for the small testproblem HELIO212b where the estimates are rather pessimistic (high). The right figure shows the same thing for the large and very ill-conditioned testproblem MHD4800a, where the estimates predict the true level of orthogonality more accurately. The upper and lower dotted horizontal line marks $\sqrt{\mathbf{u}/k} \simeq 10^{-8}$ and $\eta = \mathbf{u}^{3/4} \simeq 10^{-12}$.*

elements per row and column. Other examples where the bounds could be reduced include applications where the matrix-vector product is computed using the fast Fourier transform [23, 34] (when A is Toeplitz or circulant) or involve Kronecker products [41, 24].

Another point of great importance for the stability and efficiency of the algorithm is the proper estimation of $\|A\|_2$, which enter into the bounds on the round-off terms. If the norm is underestimated it may lead to wrong results being computed as explained above, and if $\|A\|_2$ is overestimated unnecessary work may be spent in reorthogonalization. In the applications we consider here, the matrix is usually not directly available, but is supplied in the form of subroutines that calculate the matrix-vector products Av and $A^T u$. Fortunately, the norm of A can be estimated cheaply from the quantities generated during the execution of the LBD algorithm. We know that the largest singular value $\sigma_1(A) = \|A\|_2$ usually converges in just a few iterations, and therefore we can simply use $\|B_k\|_2$ as an estimate of $\|A\|_2$. In the algorithm we will maintain an estimate of $\|B_k\|_2$ in the variable `anorm` which is updated every iteration. This can be done in a number of ways:

1. If we use that $\|B_k\|_2 = \|T_{2k+1}\|_2$ and apply the well known result

$$\|T_{2k+1}\|_2 \leq \sqrt{\|T_{2k+1}\|_1 \|T_{2k+1}\|_\infty} = \|T_{2k+1}\|_1,$$

we get the following simple updating rules for `anorm`:

$$\texttt{anorm} = \begin{cases} \alpha_1 & \text{if } j = 1 \ , \\ \max\left(\texttt{anorm}, \alpha_j + \beta_j\right) & \text{if } j \geq 2 \ , \end{cases}$$

which should be executed just before step 2.1a in the algorithm on page 30, and

$$\texttt{anorm} = \max\left(\texttt{anorm}, \alpha_j + \beta_{j+1}\right) \ ,$$

which should be executed just before step 2.2a.

2. A tighter bound can usually be obtained by using that

$$\|B_k\|_2^2 = \|B_k^T B_k\|_2 \leq \|B_k^T B_k\|_1 \ .$$

From this we get the following updating rules for `anorm`:

$$\texttt{anorm} = \begin{cases} \alpha_1 & \text{if } j = 1 \ , \\ \max(\texttt{anorm}, (\alpha_1^2 + \beta_2^2 + \alpha_2\beta_2)^{1/2}) & \text{if } j = 2 \ , \\ \max(\texttt{anorm}, (\alpha_{j-1}^2 + \beta_j^2 + \alpha_{j-1}\beta_{j-1} + \alpha_j\beta_j)^{1/2}) & \text{if } j \geq 3 \ , \end{cases}$$

which should be executed just before step 2.1a, and

$$\texttt{anorm} = \begin{cases} \max(\texttt{anorm}, (\alpha_1^2 + \beta_2^2)^{1/2}) & \text{if } j = 1 \ , \\ \max(\texttt{anorm}, (\alpha_j^2 + \beta_{j+1}^2 + \alpha_j\beta_j)^{1/2}) & \text{if } j \geq 2 \ , \end{cases}$$

which should be executed just before step 2.2a. Needless to say, care should be taken when implementing this rule to avoid overflow.

3. After a few (k=5, say) Lanczos steps, a very accurate estimate of $\|A\|_2$ can usually be found by computing the largest Ritz value $\sigma_1(B_k)$.

The simpler updating rules in 1 are used in most symmetric Lanczos codes (including lanso), but since this might increase the amount of work in partial reorthogonalization, we have decided to use the tighter bounds produced by 2 and 3.

6.2 Computing the reorthogonalization

To complete our description of BPRO, we need to provide the details of steps 2.1b and 2.2b, where the reorthogonalization takes place. An important question is which of the previous Lanczos vectors should be included when reorthogonalizing v_j or u_{j+1}. The simplest approach would be to choose $\mathcal{L}_j^\nu = \{1, 2, \ldots, j-1\}$, whenever κ_j^v exceeds the limit. This corresponds to performing one step of full reorthogonalization, and usually represents more work than necessary. To see what might be done to save some of the work, let us take a closer look at what is happening during the iteration. In Figure 9 we have plotted a series of snapshot from the Lanczos bidiagonalization showing for each iteration j the estimates $|\nu_{ji}|$, $i = 1, \ldots, j-1$ and $|\mu_{j+1,i}|$, $i = 1, \ldots, j$ (dotted lines) and the true value of the inner products $|u_i^T u_{j+1}|$, $i = 1, \ldots, j$ and $|v_i^T v_j|$, $i = 1, \ldots, j-1$ (solid lines) of the new Lanczos vectors with all the previous ones. It is very instructive to study how the levels of orthogonality behave as the iteration proceeds:

After 10 steps $\mu_{11,1}$ reaches the limit, but while u_{11} has large components along the first few Lanczos vectors, the remaining $\mu_{11,i}$ are several orders of magnitude below the limit. This suggests that we can save work if we only reorthogonalize against the vectors where $\mu_{11,i}$ is larger than some constant η, $0 \leq \eta < \sqrt{\mathbf{u}/k}$. In the figure we see what happens when this is done: After the reorthogonalization of v_{11} and u_{11}, the level of orthogonality (dash-dotted line) is reduced below η and in the following steps it grows steadily and reaches the limit once more in step 14. Again, the first few Lanczos vectors make the largest contribution to the deviation from orthogonality. The components in $\mu_{11,i}$ and $\nu_{11,i}$, $i > 6$ corresponding

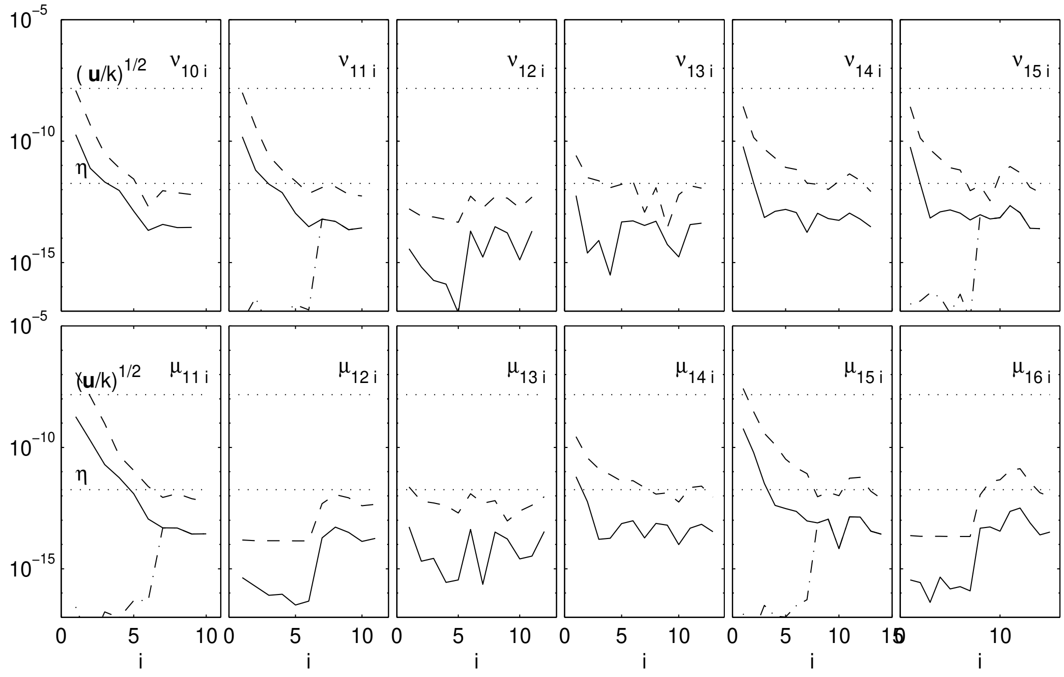


Figure 9: *Illustration of the partial reorthogonalization procedure, on testproblem HELIO212b. Each panel shows the level of orthogonality of v_j (upper panels) and u_j (lower panels) against the previous vectors after a Lanczos step. Solid lines show the true level of orthogonality $|u_i^T u_{j+1}|$, $i = 1, \ldots, j$ and $|v_i^T v_j|$, $i = 1, \ldots, j-1$, and dashed lines show the estimated level of orthogonality $|\nu_{ji}|$, $i = 1, \ldots, j-1$ and $|\mu_{j+1,i}|$, $i = 1, \ldots, j$, while dot-dashed lines show the true level of orthogonality after reorthogonalization was performed in steps 10 and 15.*

to vectors that were not reorthogonalized at step 10 have hardly risen above their original level, and consequently it would have been a complete waste to have included them in the reorthogonalization. The observed behavior is explained in the language of Paige's theorem by noting that the matrix in question has a large isolated singular value that converges within the first few iterations. After it has converged large round-off components are generated in every iteration along the corresponding Ritz vector, which lies in the span of the first few Lanczos vectors and this in turn causes $\mu_{j,i}$ and $\nu_{j,i}$ (for $i = 1, 2, 3$ or so) to grow rapidly in magnitude. For this reason it suffices to reorthogonalize against the first few Lanczos vector – a strategy which is automatically effected by using the η-criterion.

Paige's theorem also gives a clue to the reason for our observation that large components in $\mu_{j+1,i}$ (and ν_{ji}) tend to peak around some i's, such that components $\mu_{j+1,i-r} \ldots \mu_{j+1,i} \ldots \mu_{j+1,i+s}$ in a region around the $\mu_{j+1,i}$ actually exceeding the limit can also be expected to be large. This is because the Lanczos vectors generated in the iterations preceeding the convergence of a Ritz value usually all contain large components along the corresponding Ritz vector, which contributes to the round-off following the convergence. A few isolated components that exceed η, such as $\mu_{15,11}$ and $\mu_{15,12}$ in the second panel from the right in the lower part of Figure 9, usually grow slowly and are therefore quite harmless.

Based on these observations we choose the vectors to be included in a reorthogonalization as the union over all $\mu_{j+1,i}$ exceeding $\sqrt{\mathbf{u}/k}$ and their neighbors exceeding η. Thus the index sets $\mathcal{L}_j^\nu$ and $\mathcal{L}_{j+1}^\mu$ are described by the formulas

$$\mathcal{L}_j^\nu = \bigcup_{\nu_{ji}>\delta} \{k \,|\, 1 \le i-r \le k \le i+s \le j-1,\ \nu_{jk} > \eta\} \tag{6.9}$$

$$\mathcal{L}_{j+1}^\mu = \bigcup_{\mu_{j+1,i}>\delta} \{k \,|\, 1 \le i-r \le k \le i+s \le j,\ \mu_{j+1,k} > \eta\}\ , \tag{6.10}$$

where $\delta \equiv \sqrt{\mathbf{u}/k}$ is the desired level of orthogonality among the Lanczos vectors. Notice that when no estimates exceed the limit the corresponding $\mathcal{L}$ will be the empty set.

The idea of using the parameter η was introduced by Simon in [63] as part of the original PRO algorithm, and he demonstrated that the strategy could significantly reduce the amount of work without affecting the accuracy of the final results. Experimentally he found that $\eta = \mathbf{u}^{3/4}$ was the value that minimized the amount of work, and this is also the value we have used in Figure 9. The effect of changing η is illustrated for BPRO in Figure 10. In the left panel a value of $\eta = 0$ (orthogonalize against all previous vectors) caused 92 inner products to be used in the reorthogonalization. In the right panel, where a value of $\eta = (2\mathbf{u})^{3/4} \simeq 2\cdot 10^{-12}$ was used, this number was reduced to 76, without changing the number of iterations or when the reorthogonalizations occurred. The results in Table 3 show how changing the value of η affects the amount of work in BPRO. Increasing the value of η means that the total number of reorthogonalizations N_r increases because the number of steps from one reorthogonalization to the next goes down. This is however more than compensated by the fact that fewer μ's and ν's will actually have exceeded η, and thus the number of inner products N_{dot} performed in the reorthogonalization (a good measure of the total amount of work) is decreased. From the

Table 3: *The number of reorthogonalizations N_r and inner products N_d performed by BPRO for different values of η on testproblem* HELIO212b. *The left part shows the amount of work when the estimated level of orthogonality is use. The right part shows the amount of work used when the true level of orthogonality is used to decide which vectors to reorthogonalize against.*

	Work, Estimated levels		Work, True levels	
η	N_r	N_{dot}	N_r	N_{dot}
$10^3\mathbf{u}^{3/4}$	31 + 30	431 + 386	36 + 35	383 + 355
$10^2\mathbf{u}^{3/4}$	23 + 23	405 + 405	28 + 27	368 + 356
$10\mathbf{u}^{3/4}$	17 + 17	444 + 444	22 + 21	375 + 365
$\mathbf{u}^{3/4}$	16 + 16	474 + 474	16 + 16	390 + 390
$10^{-1}\mathbf{u}^{3/4}$	16 + 16	491 + 491	16 + 15	467 + 418
$10^{-2}\mathbf{u}^{3/4}$	16 + 16	491 + 491	16 + 15	494 + 449

table we see that the number of inner products is minimized for a value larger than the one recommended by Simon, namely $\eta \approx 100\mathbf{u}^{3/4}$. Despite the fact that the value recommended by Simon is not optimal for BPRO, we use it in the experiments reported below to allow consistent comparisons with the results obtained with PRO. Figures 9 and 10 also show the effect of using the pessimistic bound on the round-off terms, since estimates produced are approximately a factor of 100 too high – this seems to offer an explanation of the ratio of 100 between the value of η recommended by Simon and the value found to be optimal in

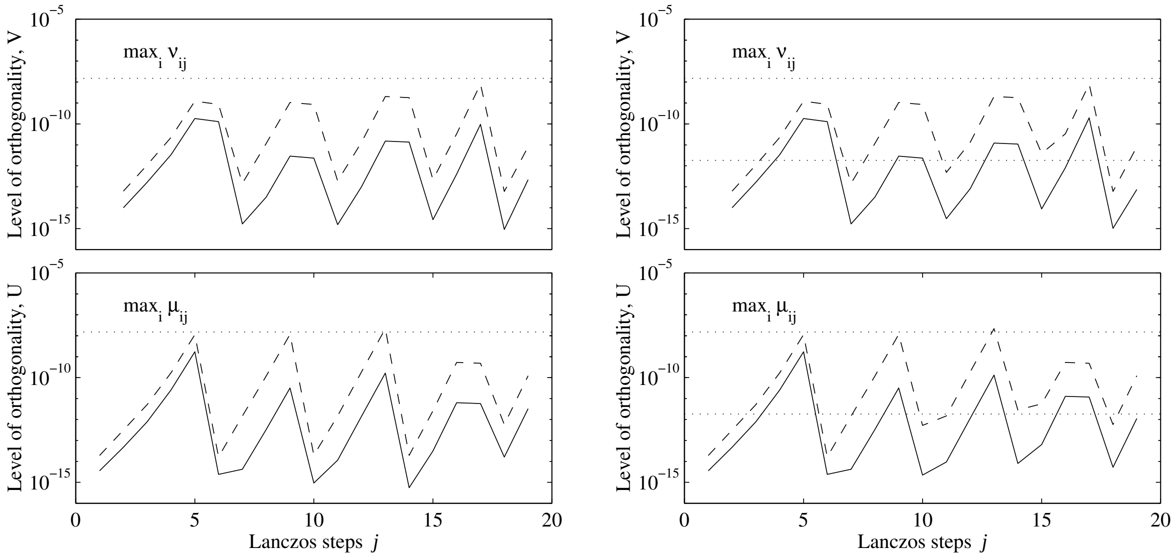


Figure 10: *Illustration of a typical BPRO run on testmatrix HELIO212b. The upper row shows* κ_j^v *(solid line) and* $\max_i \nu_{ji}$ *(dashed line), the lower panel* κ_j^u *(solid line) and* $\max_i \mu_{ji}$ *(dashed line). The upper dotted horizontal line marks the value of* $\sqrt{\mathbf{u}} \simeq 1.5 \cdot 10^{-8}$ *(all panels) and the lower marks the value of* η *which is* 0 *in the left plots and* $\mathbf{u}^{3/4} \simeq 1.8 \cdot 10^{-12}$ *on the right.*

the experiment reported in the table, but there may be other effects at play, and it is not obvious why a value of $\mathbf{u}^{3/4}$ should be the optimal. Had the true level of orthogonality been known only the first few vectors would have been reorthogonalized against, hence reducing the amount of work by approximately 20% as shown in Table 3.

From PRO we can borrow another important observation, namely that for a reorthogonalization at step j to be effective, another one should be carried out at step $j+1$. A similar thing proves to be essential for the efficiency of BPRO, as can be seen from the recurrences in Theorem 6: Assume that u_j was orthogonalized against $u_i,\ i = 1, \ldots, j-1$ at step $j-1$. Then at step j we have

$$\mu_{j+1,i} = \frac{1}{\beta_{j+1}}(\alpha_i \nu_{ji} + \beta_i \nu_{j,i-1} + O(\mathbf{u}))\ , \quad i = 1, \ldots, j\ .$$

But since u_j was reorthogonalized, one of $\mu_{ji},\ i = 1, \ldots, j-1$ must have been $O(\sqrt{\mathbf{u}})$, and therefore one or more of

$$\nu_{ji} = \frac{1}{\alpha_j}(\beta_{i+1}\mu_{j,i+1} + \alpha_i \mu_{ji} - \beta_j \nu_{j-1,i} + O(\mathbf{u}))\ , \quad i = 1, \ldots, j-1\ ,$$

is likely to be of order $\sqrt{\mathbf{u}}$. Because $\mu_{j+1,i}$ contains terms involving ν_{ji} it will immediately jump back up to level $O(\sqrt{\mathbf{u}})$ unless v_j is also reorthogonalized to reduce the large components $\nu_{ji},\ i = 1, \ldots, j-1$ to the round-off level. Needless to say, if this happens nothing will have been gained from the reorthogonalization.

This phenomenon is illustrated in Figure 11, where the positive effect of the reorthogonalization of u_{16} is annihilated in the following step by the large values of $\nu_{16,i}$. This should be compared with Figure 9, where the coupling between μ and ν was taken into account. With

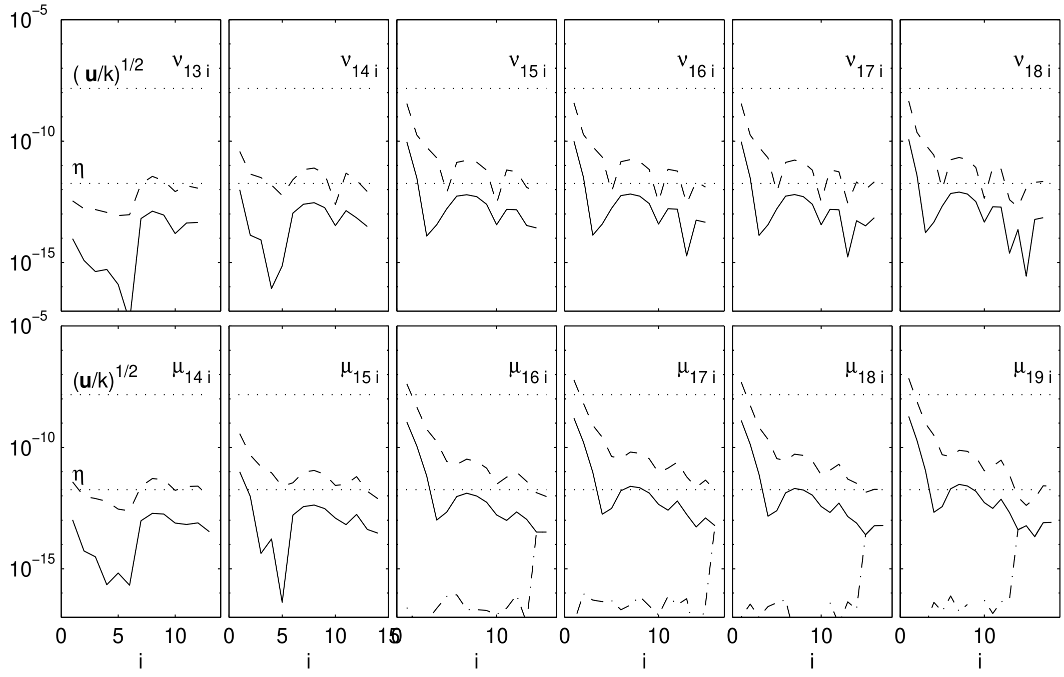


Figure 11: *The effect of ignoring the coupling between μ_{ij} and ν_{ij}. When the reorthogonalization of u_j is not immediately followed by a reorthogonalization of v_j, as is the case at step 16 in the figure, then the orthogonality of u_{j+1} is destroyed and thus triggers a new reorthogonalization of u_{j+1} in every iteration. Linestyles as in Figure 9.*

this modification, we are now in a position to write down the final form of steps 2.1b and 2.2b:

2.1b **if** $\mathcal{L}_j^\mu \neq \emptyset$ **then**
    $\mathcal{L}_j^\nu = \mathcal{L}_j^\mu$
  else
    Choose $\mathcal{L}_j^\nu$ according to (6.9)
  end

2.2b **if** $\mathcal{L}_j^\nu \neq \emptyset$ **then**
    $\mathcal{L}_{j+1}^\mu = \mathcal{L}_j^\nu$
  else
    Choose $\mathcal{L}_{j+1}^\mu$ according to (6.10)
  end

Finally we mention that in the actual implementation, the reorthogonalization is computed using either iterated classical Gram-Schmidt (CGS) or iterated modified Gram-Schmidt (MGS), since this is guaranteed to reduce the $|u_{j+1}^T u_i|$, $i \in \mathcal{L}_{j+1}^\mu$ and $|v_j^T v_i|$, $i \in \mathcal{L}_{j+1}^\nu$ to order $\mathbf{u}$; see the discussion in [6, pp. 68–69] and [17]. In practice even the CGS is very rarely iterated, unless one attempts to compute small singular values of an ill-conditioned matrix, so the cost of this safeguard is negligible. For implementation on parallel computers the iterated CGS, which can be implemented as two dense matrix-vector multiplications, often performs better than the inherently sequential MGS algorithm. As an example, iterated CGS is used in ARPACK for this reason (cf. [43, p. 50]).

6.3 Computing small singular values

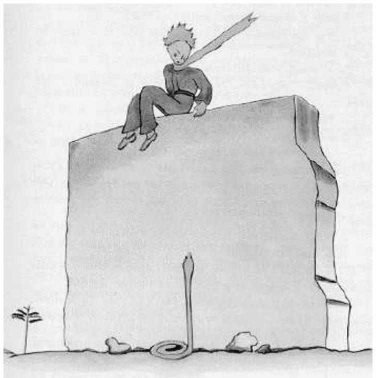
Before we go on to the evaluate the performance of the algorithm developed in this section, we wish to discuss a situation that requires special attention when a semiorthogonalization scheme such as BPRO is implemented. We know from Theorem 5 that if the columns of V_k and U_{k+1} are kept semiorthogonal, then the computed B_k corresponds to the matrix obtained from a Ritz-Galerkin projection of A where the elements have been perturbed by $O(\mathbf{u})$. This means we can expect the final accuracy of the converged Ritz values to be described by (3.14). However, if we attempt to compute the small singular values of A, the values of α_j and β_{j+1} can become smaller and smaller and the growth rate of $\mu_{j+1,i}$ and ν_{ji} will become larger and larger. From the form of the updating rules (6.7) and (6.8) we see that if eventually α_j and β_{j+1} become so small that

$$\frac{\epsilon_1\|A\|_2}{\beta_{j+1}} \geq \sqrt{\mathbf{u}/k} \qquad \text{or} \qquad \frac{\epsilon_1\|A\|_2}{\alpha_j} \geq \sqrt{\mathbf{u}/k}$$

then $\mu_{j+1,i}$ or ν_{ji} will reach the limit already in the Lanczos step following a reorthogonalization, and the algorithm has effectively switched to full reorthogonalization. There is nothing wrong in this, and the algorithm may continue many steps before having computed an invariant subspace – something which does not occur until α_j and β_{j+1} have decreased to $O(\mathbf{u}\|A\|_2)$. It tells us, however, that the semiorthogonalization is now useless. To secure a robust behavior of BPRO we have chosen to switch explicitly to full reorthogonalization if this situation arises. An example is shown in Figure 12, where BPRO switches to full reorthogonalization after 71 iterations on the HELIO212b testmatrix. After 100 iterations the algorithm terminates after having computed a full SVD of the matrix. As shown in the right panel of Figure 13 the computed singular values are comparable in accuracy to those computed by the LINPACK (cf. [21]) SVD algorithm used by MATLAB.

Although a sparse SVD algorithm is typically used to compute only a few of the largest (or smallest) singular values, we feel that a robust implementation should be able to handle problems, where singular values of highly differing magnitudes must be computed accurately. This situation occurs, e.g., when the matrix is numerically rank deficient.

6.3.1 A correction to the LANSO package



O rose thou art sick.
The invisible worm,
That flies in the night
In the howling storm:

Has found out thy bed
of crimson joy:
and his dark secret love
Does thy life destroy.

— William Blake

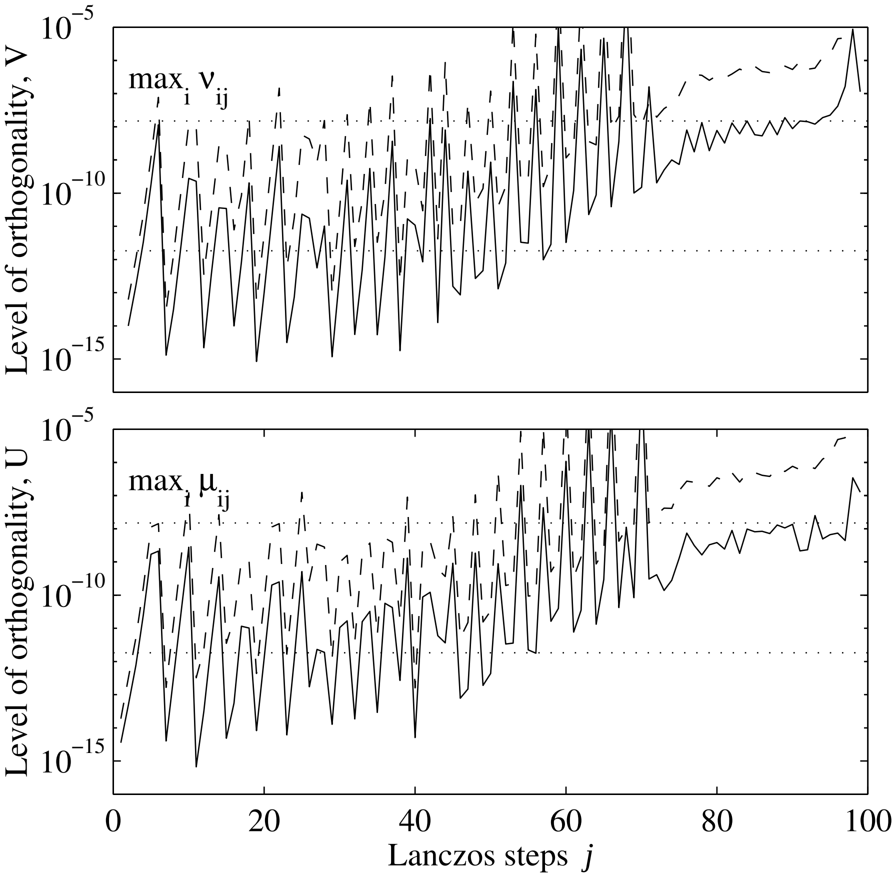


Figure 12: *Illustration of the level of orthogonality when computing all singular values for HELIO212b, which is ill-conditioned ($\sigma_1/\sigma_{212} \simeq 2.9 \cdot 10^{12}$). At iteration 71 semiorthogonality can no longer be maintained and the algorithm switches to full reorthogonalization. Linestyles as in Figure 8.*

In [23] Eldén and Sjöström analyze a class of rank deficient Toeplitz matrices that arise in signal analysis applications. They use the lanso subroutine found in SVDPACK, and compare the performance and accuracy with a number of other iterative methods for sparse SVD calculations including the ARPACK codes `ssaupd.f` and `sseupd.f`. On the largest problems the convergence of PRO is reported to fail. We have analyzed the FORTRAN code of the latest version of lanso, and found that this behavior is most likely due to a bug, which causes $\|A\|_2$ to be estimated incorrectly. In the subroutine an updating rule similar to the simple rule 1 from page 32 is used, but instead of setting $\texttt{anorm} = \max(\texttt{anorm}, \alpha_j + \beta_j)$ (using the notation of this paper) it is assigned the value $\texttt{anorm} = \alpha_j + \beta_j$. As long as only a few large singular values are computed this is not very important. However, once small singular values start to converge $\alpha_j + \beta_j$ becomes small, and this causes the round-off terms to be grossly underestimated leading to a breakdown of the semiorthogonality and appearance of spurious singular values. We have tested this hypothesis using our own MATLAB implementation of PRO, which is is modeled carefully after the abovementioned FORTRAN subroutine, and we observe the same failure as reported by Eldèn and Sjöström, when calculating the small singular values of the testproblem HELIO212b. After replacing the updating of `anorm` with one of the rules discussed in section 6.1 the method converges correctly, and even the smallest singular values are calculated with the accuracy expected from the error bound in (3.14) (results are shown in Section 7.1). This will be corrected in the next version of the LANSO package [68].

6.4 A hybrid method to improve performance on cache-based architectures

In this section we propose a small modification of BPRO that will help improving the performance on computers with a memory hierarchy. The improvement is not massive, but since the modification is very simple to implement, we feel that it is worthwhile considering.

A problem with any Lanczos algorithm employing reorthogonalization, is the need to store Lanczos vectors and recall them from secondary storage when required. As discussed earlier this can lead to poor performance on computers with a memory hierarchy, involving e.g. one or more levels of cache. The partial reorthogonalization idea only reduces the number of times vectors need to be recalled, and the number of vectors needed in a given recall. The *selective reorthogonalization* (SO) algorithm by Parlett and Scott, on the other hand, works directly from Paige's theorem by explicitly computing and reorthogonalizing against the converged Ritz vectors. This is potentially superior to the simpler approach of PRO, because the number of converged Ritz values can be much smaller than the total number of Lanczos vectors. We do not use the SO algorithm directly, but propose a modification of BPRO that is built on the same basic principle.

We argued in the previous section that it is often the same few vectors with large components along the converged Ritz vectors that make the main contribution to the round-off, and hence are included in the reorthogonalization over and over again. A simple modification to the basic BPRO scheme would be keep these few vectors in the fast cache memory and reorthogonalize against them in every iteration. This would keep the fastest growing components in μ and ν at the $O(\mathbf{u})$ level, and would increase the number of iterations from one reorthogonalization to the next. Although this is likely to increase total amount of work, the number of reorthogonalizations where vectors have to be recalled from secondary storage will decrease. If the ratio $t_{\mathrm{mem}}/t_{\mathrm{cache}}$ of the time to execute a floating point operation when the operands reside in secondary storage and in the cache is large, then the actual execution time may decrease by a significant amount, since most of the of the work is performed on vectors in the cache.

In Table 4 we have estimated the speedup for 50 iterations of BPRO on the HELIO212b testproblem resulting from keeping the first p Lanczos vectors in the cache. The speedup is calculated from the formula

$$S = \frac{N_{\mathrm{dot}}(p=0)}{(t_{\mathrm{mem}}/t_{\mathrm{cache}})^{-1} F_c N_{\mathrm{dot}}(p) + (1-F_c) N_{\mathrm{dot}}(p)} \ ,$$

which is the well-known Amdahl law (cf. [1]) in disguise. In the table we have further assumed that $t_{\mathrm{mem}}/t_{\mathrm{cache}} = 10$, which is quite reasonable (on the SGI PowerChallenge, for example, this ratio is around 15-20). The table shows that this simple modification of the basic algorithm (which is also used in the LANSO package), could give a non-trivial reduction in the execution time. As the number p of vectors that are kept in the cache and reorthogonalized against in every iteration increases, more and more inner products are computed, and at some point the

Table 4: *Effect of enforcing orthogonality against the first pairs* $u_i, v_i, i = 1, \ldots, p$ *of Lanczos vectors. In the table is given the number of pairs of vectors* p *that fit in the cache, the number of reorthogonalizations* N_r*, the number of inner products* N_{dot}*, the fraction* F_c *of the inner products that are computed with the operands in the cache, and the resulting speedup* S*. Numbers are for* $k = 50$ *iterations on testproblem HELIO212b.*

p	N_r	N_{dot}	F_c	S
0	17 + 16	512 + 462	0.00	1.00
1	14 + 14	502 + 501	0.10	1.07
2	13 + 13	509 + 507	0.19	1.16
3	12 + 11	546 + 496	0.28	1.25
5	11 + 10	608 + 558	0.41	1.33
10	10 + 10	778 + 768	0.58	1.34
15	9 + 8	987 + 937	0.67	1.28
20	8 + 7	1169 + 1119	0.70	1.17

extra work is no longer compensated by the faster execution times of operations on vectors in the cache. With the given assumptions the example shown here has a break-even point around $p = 10$, but already at $p = 3$ we see a speedup of 25%.

We suspect that this simple minded approach may be improved if the set of vectors to keep in the cache is chosen more carefully: From Figure 9 it is obvious that we should choose the vectors where the corresponding μ or ν has the fastest rate of growth. The following argument suggests how we may estimate the growth rates directly from the elements of B_k: By manipulating the recurrences in Theorem 6 we get the following expression for ν_{ji}:

$$\nu_{ji} = \frac{\alpha_i^2 + \beta_i^2}{\alpha_j \beta_j} \nu_{j-1,i} + (\text{terms in } \nu_{j,i\pm1},\ \mu_{ji},\ \mu_{j,i+1}) + O(\mathbf{u})$$

from which we can (crudely) estimate the exponential growth rate of ν_{ji} as $(\alpha_i^2 + \beta_i^2)/(\alpha_j \beta_j)$. A possible scheme would be to keep those Lanczos vectors v_i and u_i for which $\alpha_i^2 + \beta_i^2$ is largest in the cache. We have not fully implemented this modification, but experiments in MATLAB confirm that the vectors with the largest growth rates are usually those corresponding to the largest values of $\alpha_i^2 + \beta_i^2$. Our main motivation for discussing this criterion is that it is very simple to calculate – there is no need to compute the Ritz values and Ritz vectors as in SO. This is to be tested in future experiments.

7 Numerical experiments

In this section we report the results from a number of numerical experiments performed to test the efficiency and accuracy of the BPRO algorithm. To do this, we have implemented the algorithm in the MATLAB language. A short description of the resulting PROPACK package and the MATLAB source code is found in Appendix B; PROPACK is available from the author upon request. To compare BPRO with the original PRO algorithm for the symmetric Lanczos process, PROPACK includes a MATLAB version of PRO. This implementation is modeled after the FORTRAN 77 code of the latest version of the PLANSO package, which we have downloaded from the address listed in Table 1. We have used the simple forms of PRO and BPRO, without the modifications discussed in Section 6.4, since these aim at improving cache performance – an issue that we have not addressed in the experiments because they were carried out in MATLAB. As mentioned in the introduction, we have also included routines from a recent software package for large sparse eigenvalue problems and singular value decomposition called ARPACK in the comparison. The ARPACK subroutines are built on the implicitly restarted Lanczos algorithm described in [43, 12]. A subset of ARPACK has been included in MATLAB version 5.1 as the function eigs. A sparse SVD routine called svds that calls eigs to compute the eigenvalues of C in the way described in Section 3, is also available. In the experiments below we have set up matrix C and computed the eigenvalues by calling eigs directly, since this proved to be slightly more efficient than calling svds. It should also be mentioned that we used a new version of eigs in which a number of bugs in the interface routine have been removed [44] – no significant changes, which might have affected the results in Section 7.3, were made to the main computational routines.

All experiments were carried out in MATLAB 5.1 on an SGI Octane SI workstation with a 175 Mhz R10000 CPU, 256 Mbytes of memory, 32 Kbytes of primary and 1 Mbyte of secondary cache. The operating system was IRIX version 6.4. We used IEEE double precision arithmetic with unit round-off $\mathbf{u} = 2^{-53} \simeq 1.11 \cdot 10^{-16}$. If nothing else is mentioned in the text, the starting vector used for BPRO was $p_0 \in \mathbb{R}^m$ with random entries distributed uniformly on the interval $[-0.5 : 0.5]$, PRO(C) was started with $q_1 = (p_0^T, 0, \ldots, 0)^T \in \mathbb{R}^{m+n}$, and PRO($A^TA$) was started with $q_1 = A^T p_0$.

7.1 Experiment 1: Accuracy of the computed singular values

In the first experiment we have analyzed the accuracy of the singular values computed by BPRO and PRO applied to A^TA and C. This was done for matrices HELIO212a and HELIO212b, and for comparison we also analyzed the output from the svd routine in MATLAB, which uses the standard Golub-Kahan algorithm implemented in the LINPACK routine _SVDC. The results in the lower panel were obtained for a full matrix $\hat{A}$ constructed by first computing the SVD of HELIO212b

$$A = U \Sigma V^T ,$$

computed using svd, and then setting $\hat{A}$ equal to $U * \Sigma * V^T$. Subsequently all methods (including svd) were used to compute the singular values of $\hat{A}$ and these were subsequently compared to Σ. The results are shown in the right-hand panel of Figure 13. We notice that the accuracy of BPRO, PRO(C) and the standard SVD algorithm is comparable, and that the results look similar to Figure 3 shown in connection with our discussion of the error bounds. In fact, we cannot expect the accuracy to be any better, since the singular values of $\hat{A}$ are the singular values of $A+E$, $\|E\|_2 \approx \mathbf{u}\|A\|_2$, where E accounts for the round-off errors committed in setting up $\hat{A}$.

To see how far we can push the accuracy of the partial reorthogonalization procedures, we applied the Lanczos algorithms to a simple matrix, where the singular values were known exactly. The results displayed in the left-hand panel were computed for the matrix HELIO212a generated by permuting randomly the rows and columns of the diagonal matrix Σ obtained from the SVD of HELIO212b. Mathematically, the Lanczos algorithm is invariant with respect to any similarity transformation of the matrix. This means that in exact arithmetic PRO and BPRO would compute exactly the same Ritz values for HELIO212a as for HELIO212b. In

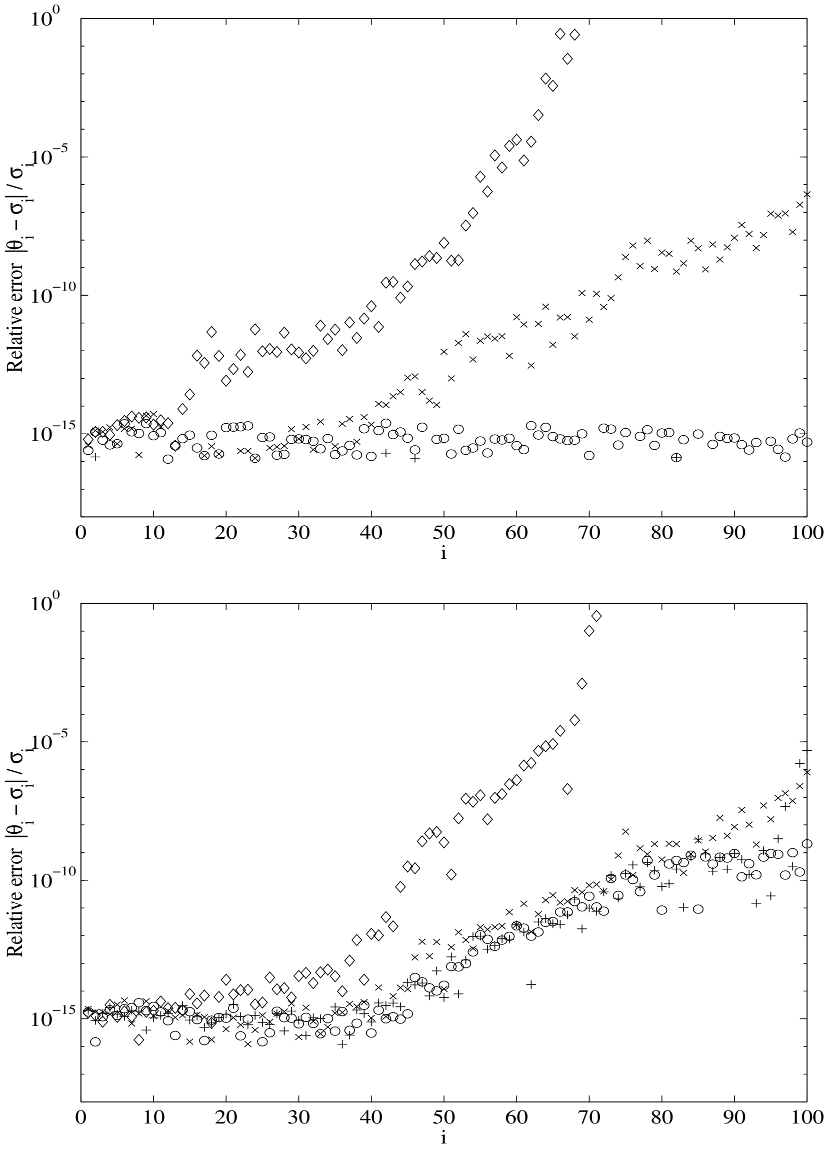


Figure 13: *Relative error in the computed singular values of HELIO212a (upper panel) and HELIO212b (lower panel). Diamonds correspond to PRO(A^TA), crosses to PRO($\hat{C}$), circles to BPRO(A) and pluses to the built-in* svd *command in MATLAB. The plusses are (almost!?) missing in the upper panel because the* svd *command computes the singular values of a permuted diagonal matrix exactly.*

finite precision arithmetic the situation is quite different, since when multiplying a vector by HELIO212b the round-off errors committed in accumulating the inner products will perturb the singular values by $\mathbf{u}\|A\|_2$, which was what we observed in the right-hand panel above. For the matrix HELIO212a, however, the matrix-vector products are computed almost exactly, because they just consist in scaling and permuting the entries in the vector, and we should therefore expect the computed singular values to be almost exact. As can be seen in the upper panel in the figure, this also seems the case for BPRO while PRO is apparently not able to take advantage of HELIO212a being (apart from a permutation) diagonal. This seems a bit odd, and the explanation is that the difference has nothing to do with the two different Lanczos algorithms used! The entries $\alpha_1, \ldots, \alpha_k$ and $\beta_2, \ldots, \beta_k$ in T_{2k} and B_k computed by PRO(C) and BPRO are the same to within round-off. Instead, the difference is in the algorithms that are used to calculate the eigenvalues of T_{2k} and the singular values of B_k: The bidiagonal SVD algorithm (see [19]) can compute even the smallest singular values with guaranteed high relative accuracy; the same is not true for the QL/QR algorithm used for computing the eigenvalues of a symmetric tridiagonal. As a test we used the α's and β's in T_k to construct an equivalent bidiagonal matrix B'_k and used the bidiagonal SVD algorithm to compute its singular values. It turns out that these are equal to the true singular values within a few units of round-off, exactly like those of the bidiagonal computed by BPRO – another warning to be careful when computing singular values via an equivalent symmetric system!

In both panels we also notice the destructive effect of the rounding errors when working with A^TA that make it impossible to determine the small singular values. For this testmatrix PRO(A^TA) terminates after $j = 71$ iterations with a numerically invariant subspace. This is due the squaring of the singular values, which causes the 29 smallest eigenvalues of A^TA to be numerically zero, i.e. smaller $\mathbf{u}\|A^TA\|_2$.

7.2 Experiment 2: Efficiency of the reorthogonalization methods

The next experiment compares the efficiency of the various reorthogonalization methods. To do this we generated Krylov subspaces of different dimensions for HELIO212a, and recorded the number of reorthogonalizations and inner products computed in the Gram-Schmidt procedure. The results for PRO(A^TA), PRO(C), BPRO, and the corresponding Lanczos algorithms with full reorthogonalization are listed in Table 5, along with the actual number of floating point operations used. The latter was measured using the `flops` command in MATLAB, which counts the number of arithmetic operations actually performed by the program. All methods perform two matrix-vector multiplications per iteration, one with A and one with A^T, and we deliberately chose the diagonal matrix HELIO212a to minimize the contribution from the matrix-vector multiplications to the flop count. It should be noticed that the PRO method applied to A^TA switches to full reorthogonalization after $j = 46$ iterations (see Section 6.3) and both PRO and FRO(A^TA) terminates at $j = 71$ after having found a numerically invariant subspace. The PRO method applied to C and BPRO switch to full reorthogonalization after 142 and 72 iterations respectively.

The table shows that for $k \leq 40$ the three algorithms based on partial reorthogonalization (PRO(A^TA), PRO(C), BPRO) are successful in reducing the amount of work, both in terms of the number of reorthogonalizations performed and in terms of the number of flops. Compared to its FRO counterpart, the amount of work is approximately halved by using PRO(A^TA) while PRO(C) and BPRO reduce it by almost a factor of 3 compared to FRO(C) and BFRO(A). With $k > 40$ the smaller Ritz values start to converge, forcing the algorithms to perform more reorthogonalization. At 71 iterations the advantage of PRO over FRO is reduced to 30% for C and a mere 9% for A^TA. The advantage of BPRO over BFRO has also decreased, but is still a factor of 1.8.

We recall from Table 2 shown in Section 3.3, that the expected ratio between the amount of work in FRO(A^TA) and BFRO is approximately $(n + m)/n = 3.12$ and approximately 4 between BFRO and FRO(C). This is confirmed by numbers in the lower part of the table. If we use the numbers in the "Mops" columns to calculate the same ratios for the three algo-

Table 5: *Work used by the various algorithms to generate an orthogonal basis for the Krylov subspaces* $\mathcal{K}_k(A^TA, u_1)$ *and* $\mathcal{K}_k(AA^T, v_1)$, *where* A *is the testmatrix HELIO212a. In the table is given the number of reorthogonalizations* N_r, *the number of inner products* N_{dot} *and the number of floating point operations* Mops *(in millions). Entries on the form* $N_r = p + q$ *for BPRO/BFRO indicates that* p *reorthogonalization were performed for* u_{j+1} *and* q *for* v_j.

k	N_r	N_{dot}	Mops	N_r	N_r	Mops	N_r	N_{dot}	Mops
	PRO(A^TA)			PRO(C)			BPRO(A)		
10	2	10	0.019	4	54	0.16	1 + 1	6 + 6	0.038
20	8	84	0.066	10	174	0.40	4 + 4	42 + 42	0.12
40	23	505	0.28	28	1072	1.73	10 + 10	209 + 209	0.40
60	43	1506	0.71	65	4798	6.65	23 + 23	859 + 859	1.30
71	54	2232	1.05	87	7681	10.40	31 + 30	1316 + 1316	1.92
100	—	—	—	145	17887	23.44	60 + 59	3810 + 3781	5.14
	FRO(A^TA)			FRO(C)			BFRO(A)		
10	9	54	0.039	20	230	0.39	10 + 9	55 + 45	0.11
20	19	209	0.12	40	860	1.28	20 + 19	210 + 190	0.35
40	39	819	0.40	80	3320	4.56	40 + 39	820 + 780	1.20
60	59	1829	0.83	120	7380	9.82	60 + 59	1830 + 1770	2.54
71	70	2555	1.14	142	10295	13.58	71 + 70	2556 + 2485	3.50
100	—	—	—	200	20300	25.34	100 + 99	5050 + 5050	6.30

rithms based on partial reorthogonalization we obtain the following:

k	10	20	40	60	71	100
BPRO/PRO(A^TA)	2.0	1.8	1.4	1.8	1.8	—
PRO(C)/ BPRO	4.2	3.3	4.3	5.1	5.4	4.6

We notice that while PRO(C)/ BPRO is slightly higher than the expected value of 4, the ratio between BPRO and PRO(A^TA) is only about half of the expected. We can find an explanations for the relatively poor performance of PRO(A^TA) by comparing the number of reorthogonalizations N_r in the upper and lower part of the table. We notice that while N_r for FRO(A^TA) is half the value of N_r for BFRO, this quantity is approximately the same for PRO(A^TA) and BPRO. PRO(A^TA) reorthogonalizes twice as often as BPRO.

The problem with PRO(A^TA) is, exactly as we saw in the error analysis, that the eigenvalues of A^TA are the square of the singular values of A, which means that the growth rates (see Section 6.4) of $\mu_{j,1}$ and ν_{j1} are doubled. For this particular experiment we can explain the results by noticing that the matrix HELIO212b has a large isolated singular value that converges almost in the first two steps. Following the convergence the growth rate of the dominating components $\mu_{j,1}$ and ν_{j1} in BPRO are approximately given by the ratio

$$((\beta_2^2 + \alpha_2^2)/(\beta_3^2 + \alpha_3^2))^{1/2} \approx \theta_1/\theta_2 \simeq 1.7542/0.7599 \simeq 2.3 \ .$$

Thus the number of steps l it takes for these components to rise from $\eta \simeq 1.8 \cdot 10^{-12}$ to $\sqrt{\mathbf{u}} \simeq 1.5 \cdot 10^{-8}$ is determined by

$$\left(\frac{\theta_1}{\theta_2}\right)^l \cdot \eta \geq \sqrt{\mathbf{u}} \qquad \Rightarrow \qquad l = \left\lceil \frac{\log\theta_1 - \log\theta_2}{\log\sqrt{\mathbf{u}} - \log\eta} \right\rceil = 11$$

which is indeed equal to the number of steps before the first reorthogonalization occurs in the experiment. The same kind of analysis applies to the ω-recurrences in PRO(A^TA) from which we get

$$\left(\frac{\theta_1^2}{\theta_2^2}\right)^l \cdot \eta = \left(\frac{\theta_1}{\theta_2}\right)^{2l} \cdot \eta \geq \sqrt{\mathbf{u}} \qquad \Rightarrow \qquad l = \left\lceil \frac{1}{2} \frac{\log\theta_1 - \log\theta_2}{\log\sqrt{\mathbf{u}} - \log\eta} \right\rceil = 6 \ .$$

Here the exponent ($2l$) is doubled compared to BPRO and therefore only half as many Lanczos steps can be taken before it becomes necessary to reorthogonalize the Lanczos vectors. A similar behavior is observed at later stages in the iteration process.

The result is that the number of reorthogonalizations in PRO(A^TA) will always be about twice the number used by BPRO, and as a consequence the expected ratio between the number of operations in the two methods will be $(n+m)/(2n)$ rather than $(n+m)/n$. As an example, this means that we can expect BPRO to be just as efficient as PRO(A^TA) when the matrix is square. This result is confirmed by our experiments with testmatrices from the Harwell-Boeing collection described in the next section.

7.3 Experiment 3: Comparison of sparse SVD algorithms

To make sure that our algorithms have not simply been tuned to perform well on one "toy problem", we have submitted the implementations of PRO and BPRO to an experiment involving a set of matrices from a variety of different application ranging from Geodetic surveying to the calculation of Alfvèn spectra in magnetohydrodynamics. The suite of testmatrices represents a wide variety of sparsity patterns, sizes and numerical properties. In light of the discussion in Section 6.3 regarding the challenge of computing small singular values and the effect of the matrix structure on the estimates of the round-off terms that occur in the updating of $\mu_{j+1,i}$ and ν_{ji}, it is important to include matrices with different densities, and the testsuite should include both well- and ill-conditioned problems. To achieve this we have used a number of matrices from the Harwell-Boeing collection, which contains a large set of matrices from a variety real-world applications and is widely used for testing numerical linear algebra software. The matrices together with MATLAB, C or FORTRAN subroutines for input and output, can be downloaded from the web-site called "Matrix Marked" [9]. The characteristics of the testmatrices are given in Table 6, and the 10 largest singular values of each matrix are listed in Appendix A.

From the Harwell-Boeing collection we have selected two sparse least squares problems: ABB313 which is small and well-conditioned, and ILLC1850 which is larger and ill-conditioned and was originally used for testing the LSQR algorithm. In addition, we have chosen 4 square non-symmetric matrices (originally non-symmetric eigenvalue problem): WEST0479, RBS480a, QH1484 and MHD4800a. The matrices WEST0479 and RBS480a are both fairly small, and while WEST0479 is very sparse and ill-conditioned RBS480a is denser and fairly well-conditioned. In particular RBS480a has a cluster of large singular values, which makes the Ritz values converge quite slowly. This is the reason for the large value of k in Table 6, and explains why PRO and BPRO are so effective for this particular matrix since the growth rates of μ and ν become small. The matrices QH1484 and MHD4800a have been included to test the methods on large and very ill-conditioned problems. The matrix AF23560 was included for the simple reason that it is one of the largest matrices in the collection.

In this experiment we have computed the 10 largest singular values of the nine testmatrices. The number of floating point operations and the execution time for the different

Table 6: *Characteristics of the test problems. The column k lists the dimension of the Krylov subspaces necessary to obtain a relative error* $|\theta_i - \sigma_i|/\sigma_i < 100\mathbf{u}$, $i = 1, \ldots, 10$ *in the experiment reported in Table 7. The criterion was checked by comparing the computed singular values* $\hat{\sigma}_i$ *with* σ_i *computed by the built-in* `svd` *routine in MATLAB.*

Matrix	m	n	nnz	Density	$\kappa_2(A)$	k	Description
HELIO212a	212	100	100	0.0047	$2.9 \cdot 10^{12}$	29	Helioseismic inversion
HELIO212b	212	100	21200	1.0	$2.9 \cdot 10^{12}$	29	Helioseismic inversion
ABB313	313	175	1557	0.028	$1.5 \cdot 10^{1}$	40	Sudan Survey
ILLC1850	1850	712	8758	0.0066	$1.4 \cdot 10^{3}$	61	Gravity meter survey
WEST0479	479	479	1887	0.0082	$1.4 \cdot 10^{12}$	17	Chemical engineering
RBS480a	480	480	17088	0.074	$1.3 \cdot 10^{5}$	75	Robotics
QH1484	1484	1484	6110	0.0028	$5.6 \cdot 10^{17}$	28	Power systems simulation
MHD4800a	4800	4800	102252	0.0044	$3.2 \cdot 10^{47}$	25	Magnetohydrodynamics
AF23560	23560	23560	460598	0.0008	Unknown	70	Comp. fluid dynamics

algorithms is listed in Table 7. The number of flops listed for BPRO include the cost of computing the SVD of the bidiagonal matrix B_k by means of the `svd` routine in MATLAB, and the number of flops listed for PRO include the cost of computing Schur decomposition of the tridiagonal matrix T_k by means of the `eig` routine in MATLAB. The † indicates that it was not possible to make the `eigs` function converge for QH1484 (not even after increasing the size of the Lanczos basis and the maximal number of iterations.) One way in which QH1484 differs significantly from the other matrices, is that it has a very large norm $\|A\|_2 = 1.26 \cdot 10^{16}$, and that could be part of the explanation. Neither `eigs` nor the built-in `svd` routine could be used with the largest testproblems MHD4800a and AF23560 due to lack of memory. The column "error" lists the quantity $\lceil \log_2 |\theta_i - \sigma_i|/(\mathbf{u}\sigma_i) \rceil$, i.e. the number of "wrong bits" in the binary representation of θ_i compared to the values σ_i calculated by the `svd` routine. For problems MHD4800a and AF23560 where the full matrix was too large to fit in memory, the singular values calculated using LBD with full reorthogonalization (BFRO) were used as the reference values σ_i to estimate the error. For HELIO212a the singular values were known exactly, as explained in Section 7.1. Also notice that HELIO212b is a full matrix and the large contribution to the flop count from the matrix-vector multiplications reduces the difference in performance between the different algorithms. Finally, we should point out that `svd` is a built-in command in MATLAB implemented by calling a FORTRAN library routine. This is the reason behind the smaller execution time of `svd` even for problems like ABB313, where it performs many times more floating points operations than the Lanczos based methods.

In Table 7 we find the same ratio between the number of floating point operations used by BPRO and PRO(A^TA) observed in the previous experiment, namely that BPRO performs approximately $(m+n)/(2n)$ times as many operations as PRO(A^TA). This relation seems to hold independent of the condition number or other properties of the matrix. Moreover, BPRO and PRO(A^TA) are 3 to 4 times faster than PRO(C), and the latter is even slower than BFRO for most problems. The improvement in performance of BPRO over BFRO can be seen to vary quite dramatically, ranging from a factor of 1.5 for MHD4800a to 5 for ILLC1850. This value depends on the gap-structure of the spectrum of singular values for the matrix involved. If the singular values are densely clustered, the growth rate of μ and ν will be small and a lot of work can be saved by BPRO compared to full reorthogonalization. If on the other hand

Table 7: *Comparison of the various methods using testmatrices from the Harwell-Boeing collection. The column "Mops" is the number of floating point operations used (in millions), "time" is the execution time in seconds and "error" is the relative error given as the number of erroneous bits in the result.*

	Mops	time	error	Mops	time	error	Mops	time	error
Matrix	HELIO212a			HELIO212b			ABB313		
svd(full(A))	1.88	0.05	0	7.30	0.17	5	29.06	0.60	0
eigs(C)	4.12	1.37	4	18.41	7.30	5	19.82	6.30	6
PRO(A^TA)	0.15	0.22	5	2.60	1.11	7	0.48	0.35	6
PRO(C)	0.91	0.50	5	5.90	1.77	5	2.32	0.87	6
BPRO(A)	0.27	0.30	4	2.71	1.20	5	0.65	0.46	6
BFRO(A)	0.69	0.46	4	3.14	1.41	6	2.20	0.91	5
Matrix	ILLC1850			WEST0479			RBS480a		
svd(full(A))	3272.38	71.10	0	293.04	6.54	0	299.48	7.12	0
eigs(C)	133.07	32.72	6	23.01	5.88	5	79.58	20.73	7
PRO(A^TA)	3.65	1.54	5	0.44	0.24	12	6.58	2.52	6
PRO(C)	21.56	4.41	5	1.87	0.55	5	20.16	4.88	6
BPRO(A)	4.74	1.83	6	0.47	0.26	4	7.02	2.75	6
BFRO(A)	23.80	4.83	6	0.91	0.31	4	17.37	5.15	6
Matrix	QH1484			MHD4800a			AF23560		
svd(full(A))	8599.10	184.99	0	—	—	—	—	—	—
eigs(C)	75.22	20.74	35†	—	—	—	—	—	—
PRO(A^TA)	3.30	0.93	5	16.71	5.21	12	200.16	60.06	6
PRO(C)	12.58	2.57	6	48.95	9.60	6	641.12	177.36	6
BPRO(A)	2.76	0.91	6	16.90	5.44	2	206.97	63.15	4
BFRO(A)	6.46	1.41	8	25.39	6.65	0	628.52	126.40	0

the matrix has large isolated singular values, then as soon as these begin to converge μ and ν will start growing rapidly with the result that many reorthogonalizations will be necessary to keep the Lanczos vectors semiorthogonal. From the table in Appendix A we see that for MHD4800a the initial growth rate of μ and ν is approximately $\sigma_1/\sigma_2 = 2.76$, while it is only $\sigma_1/\sigma_2 = 1.02$ for ILLC1850. This explains the large variations in the relative performance of BPRO compared to BFRO.

When we compare the Lanczos based algorithms with the eigs routine that is based on the implicitly restarted Lanczos algorithm, the difference in performance is quite striking. The execution time for BPRO is from 4.6 (HELIO212a) to 22.8 (QH1484) times smaller than for eigs applied to the same problem. This large difference cannot be explained by inefficient MATLAB programming alone, because the actual number of operations is also reduced between 6.8 (HELIO212b) and 50 (WEST0479) times. One may speculate why the implicitly restarted Lanczos algorithm in eigs is so inefficient for solving the equivalent symmetric eigenproblems, but since this may simply be a problem with the MATLAB implementation, we postpone this discussion until we have made a comparison of a FORTRAN implementation of BPRO with the original routines in ARPACK[3].

[3]The Mathworks (the creator of MATLAB) are aware that eigs and svds are very slow, as can be seen in their reply to a question from a MATLAB user regarding this issue. The reply can be found in the document `http://www.mathworks.com/support/solutions/v5/7788.shtml` on their website. A number of bugs found

Finally we regard the accuracy of the computed singular values. The singular values computed by the majority of the Lanczos based methods differ from the singular values computed by `svd` in the last 5-6 bits, and as suggested by the result for HELIO212b this is indeed the accuracy that can be expected from `svd` itself. Only PRO(A^TA) exhibits larger errors for some problems To understand this we need to consider the effective condition number σ_1/σ_{10} for computing the 10 largest singular values. In the table below this number is listed for the testproblems used in the experiment:

Matrix	HELIO212(a,b)	ABB313	ILLC1850	WEST0479
σ_1/σ_{10}	6.8	1.4	1.1	75.2
Matrix	RBS480a	QH1484	MHD4800a	AF23560
σ_1/σ_{10}	1.2	1.0	16.3	1.2

The table indicates that HELIO212(a,b), WEST0479 and MHD4800a are the problems where we might expect problems for PRO(A^TA). Indeed, if one calculates from (3.9) and (3.14) the additional error caused by the squaring of the condition number this predicts the differences in the number of correct bits in Table 7 quite well. The fact that this effect is important when calculating the 10 largest singular values is a fair warning that an unstable method working explicitly with A^TA should be used with care if the effective condition number of the problem to be solved is not known in advance.

8 Conclusion

Compared to sparse SVD algorithms based on the symmetric Lanczos algorithm with partial reorthogonalization the BPRO algorithm developed here has the advantage that it is both computationally efficient and capable of computing the singular values with good relative accuracy. In comparison with the two variants of PRO used in SVDPACK, BPRO achieves the same (or higher) accuracy as PRO(C) and at the same time is just as fast as PRO(A^TA) for square matrices and only a factor of $(m+n)/(2n)$ slower for rectangular matrices as opposed to $(m+n)/(n)$ as one might aspect from comparing the corresponding algorithms with full reorthogonalization. In particular for problems where the singular values must be computed very accurately, or where they are of highly varying magnitudes, e.g. when the matrix is almost numerically rank deficient, BPRO represents a substantial improvement, because it reduces the amount of work by a factor of 3–4 compared to PRO(C).

Our experiments with matrices from the Harwell-Boeing collection have shown that the BPRO implementation is efficient and robust when applied to problems of different size and with a wide variety of numerical properties. We believe that BPRO is well suited as the computational kernel in software for large-scale sparse or structured SVD calculations, as well as other methods based on the Lanczos bidiagonalization. It will be useful for solving large sparse or structured linear least squares problems with multiple right-hand sides, and also provides the basis for an efficient implementation of the LSQR algorithm where the convergence is not slowed down by the loss of orthogonality.

A number of questions regarding the BPRO algorithm still need to be considered. First of all we believe that there is room for improvement in computing better estimates of the round-off terms that enter into the recurrences to monitor the loss of orthogonality. Our experiments have shown that especially for problems where the loss of orthogonality is slow,

in these routines will be corrected in the next version of MATLAB [44].

the simple estimates used in the current implementation can be up to 100 times above the true values, and up to 40% of the work done in the reorthogonalization is unnecessary. Choosing a larger value for η will solve this problem for some matrices, but since we have observed the factor by which μ and ν are overestimated to be very problem dependent, this is probably not a very general solution. As discussed earlier we believe the inclusion of problem specific information, such as the number non-zeros per row and column in the sparse matrix, to be part of the answer, but one might also consider adaptive schemes that try to collect statistical information about the rounding errors during the iteration.

Another interesting point is the hybrid schemes discussed in Section 6.4, where ideas from selective reorthogonalization are used to suggest a way of improving the performance of BPRO on machines with a memory hierarchy. Finally we wish to mention that an efficient parallel implementation of the LANSO package has recently appeared. The structure of PRO and BPRO is very similar, and we see no reason why the good parallel performance reported in [67] should not be attainable by a parallel implementation of BPRO.

It should be emphasized that our experiments have been carried out in MATLAB, and it is therefore difficult to compare the computational efficiency of our methods with the routines from, e.g., ARPACK. Using the `flops` counter in MATLAB usually gives a good estimate of the amount of work involved in various algorithms, but it is obvious that implementation details can have a substantial impact on the performance of the program on a given machine. Therefore we look forward to comparing a parallel FORTRAN implementation of BPRO with the routines from (P)ARPACK, SVDPACK and (P)LANSO.

References

[1] G. M. Amdahl, *Validity of the single processor approach to achieving large-scale computing capabilities*, in AFIPS Conference Proceedings, **30**, AFIPS Press, Montvale, NJ, 1967, 483–485.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, O. Ostrouchow and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[3] W. Bath, R. S. Martin and J. H. Wilkinson, *Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection*, Numerische Mathematik, **9** (1967), 386–393.

[4] M. W. Berry, *Large-scale sparse singular value computations*, Int. J. Supercomputer Appl., **6** (1992), no. 1, 1–21.

[5] Å. Björck, *A bidiagonalization algorithm for solving large and sparse ill-posed systems of linear equations*, BIT, **28** (1988), 659–670.

[6] Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996.

[7] Å. Björck, T. Elfving and Z. Strakoš. *Stability of conjugate gradient and Lanczos methods for linear least squares problems*, SIAM J. Matrix Anal. Appl., **19** (1998), no. 3, 720–736.

[8] Å. Björck, E. Grimme and P. Van Dooren, *An implicit shift bidiagonalization algorithm for ill-posed systems*, BIT, **34** (1994), no. 4, 510–534.

[9] R. Boisvert, R. Pozo, K. Remington, R. Barrett, J. Dongarra, *Matrix Market : a web resource for test matrix collections, The Quality of Numerical Software: Assessment and Enhancement*, (R. Boisvert, ed.), Chapman and Hall, London, 1997, pp. 125–137. URL: `http://math.nist.gov/MatrixMarket`

[10] C. G. Broyden, *Look-ahead block-CG algorithms*, in "Algorithms for large scale linear algebraic systems" (Gran Canaria 1996), eds. G. A. Althaus & E. Spedicato, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci., **508**, 197–215, Kluwer Acad. Publ., Dordrecht, 1998.

[11] D. Calvetti, G. H. Golub, L. Reichel, *Estimation of the L-Curve via Lanczos Bidiagonalization*, Tech. Rep. SCCM-97-12, Stanford University, 1997.

[12] D. Calvetti, L. Reichel and D. C. Sorensen, *An Implicitly Restarted Lanczos Method for Large Symmetric Eigenvalue Problems*, Elec. Trans. Num. Anal., **2** (1994), 1–21.

[13] T. Chan and W. L. Wan, *Analysis of Projection Methods for solving Linear Systems with Multiple Right-hand Sides*, SIAM J. Sci. Comput., **18** (1997), no. 6, 1698–1721.

[14] J. Christensen-Dalsgaard, J. Schou and M. J. Thompson, *A comparison of methods for inverting helioseismic data*, Mon. Not. R. Astr. Soc., **242** (1990), 353–369.

[15] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations. Vol. I Theory*, Birkhäuser, Boston, 1985.

[16] J. K. Cullum, R. A. Willoughby and M. Lake, *A Lanczos algorithm for computing singular values and vectors of large matrices*, SIAM J. Sci. Statist. Comput., **4** (1983), no. 2, 197–215.

[17] J. W. Daniel, W. B. Gragg, L. Kaufman and G. W. Stewart, *Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization*, Math. Comp., **30** (1976), no. 136, 772–795.

[18] J. W. Demmel, M. T. Heath, H. A. van der Vorst, *Parallel Numerical Linear Algebra*, Acta Numerica 1993, 111–197.

[19] J. W. Demmel and W. Kahan, *Accurate singular values of bidiagonal matrices*, SIAM J. Sci. Statist. Comput., **11** (1990), 712–719.

[20] J. J. Dongarra, *Improving the accuracy of computed singular values*, SIAM J. Sci. Statist. Comput., **4** (1983), no. 4, 712–719.

[21] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

[22] L. Eldén, *Algorithms for the regularization of ill-conditioned least-squares problems*, BIT, **17** (1977), 134–145.

[23] L. Eldén and E. Sjöström, *Fast computation of the principal singular values of Toeplitz matrices arising in exponential data fitting*, Signal Processing, **50** (1996), 151–164.

[24] D. W. Fausett, C. T. Fulton, *Large least squares problems involving Kronecker products*, SIAM J. Matrix Anal. Appl., **15** (1994), 219-227.

[25] G. H. Golub, M. Heath and G. Wahba, *Generalized Cross-Validation as a Method for Choosing a Good Ridge Parameter*, Technometrics, **21** (1979), 215–223.

[26] G. H. Golub and W. Kahan, *Calculating the singular values and pseudo-inverse of a matrix*, J. Soc. Indust. Appl. Math. Ser. B Numer. Anal., **2** (1965), 205–224.

[27] G. H. Golub, F. T. Luk and M. L. Overton, *A block Lanczos method for computing the singular values and corresponding vectors of a matrix*, ACM Trans. Math. Software, **7** (1981), 149–169.

[28] G. H. Golub, R. Underwood and J. Wilkinson, *The Lanczos Algorithm for the $Ax = \lambda Bx$ Problem*, Report STAN-CS-72-270, Department of Computer Science, Stanford University, Stanford, California, 1972.

[29] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2. Ed., Johns Hopkins University Press, Baltimore, 1989.

[30] G. H. Golub, U. von Matt, *Generalized Cross-Validation for Large Scale Problems*, Journal of Computational and Graphical Statistics, **6** (1997), 1–34.

[31] G. H. Golub, U. von Matt, *Tikhonov Regularization for Large Scale Problems*, in Workshop on Scientific Computing, eds. G. H. Golub, S. H. Lui, F. Luk. and R. Plemmons, Springer, New York, 1997.

[32] A. Greenbaum and J. J. Dongarra, *Experiments with QL/QR methods for the symmetric tridiagonal eigenproblem*, Computer Science Dept. Technical Report CS-89-92, University of Tennessee, Knoxville, 1989. (LAPACK Working Note 17).

[33] M. Hanke and P. C. Hansen, *Regularization methods for large-scale problems*, Surv. Math. Ind., **3** (1993), 253–315.

[34] M. Hanke, J. Nagy and R. Plemmons, *Preconditioned iterative regularization for ill-posed problems*, Numerical linear algebra (Kent, OH, 1992), 141–163, de Gruyter, Berlin, 1993.

[35] P. C. Hansen, *The discrete Picard condition for discrete ill-posed problems*, BIT, **30** (1990), 658–672.

[36] P. C. Hansen, *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*, SIAM, Philadelphia, 1998.

[37] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.

[38] W. Kahan, *The Improbability of Probabilistic Error Analyses for Numerical Computations*, Prepared for the UCB Statistics Colloquium, February 1996, URL: `http://http.cs.berkeley.edu/~wkahan/improber.ps`.

[39] S. Kaniel, *Estimates for Some Computational Techniques in Linear Algebra*, Math. Comp., **20** (1966), 369–78.

[40] C. Lanczos, *An Iteration methods Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators*, J. of Res. Nat. Bur. Stand. **45** (1950), 255–282.

[41] R. M. Larsen, *Iterative algorithms for two-dimensional helioseismic inversion*, in Proc. of the Interdisciplinary Inversion Workshop 5 (Ed. B. H Jacobsen), Dept. Earth Sciences, Aarhus University (1997), 123–138.

[42] R. M. Larsen and P. C. Hansen, *Efficient Implementations of the SOLA Mollifier Method*, Astron. Astrophys. Suppl., **121** (1997), 587–598.

[43] R. B. Lehoucq, D. C Sorensen and C. Yang, *ARPACK Users's Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, SIAM, Philadelphia, 1998, URL: `http://www.caam.rice.edu/software/ARPACK`.

[44] The Mathworks, Private communication, 1998.

[45] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra Appl., **29** (1980), 292–322.

[46] D. P. O'Leary and J. A. Simmons, *A bidiagonalization-regularization procedure for large scale discretizations of ill-posed problems*, SIAM J. Sci. Statist. Comput., **2** (1981), 474–489.

[47] G. Orwell, *1984*, Copyright © 1949 Harcourt Brace Jovanovich, Inc.

[48] C. C. Paige, *The Computation of Eigenvalues and Eigenvectors of Very Large Sparse Matrices*, Ph. D. thesis, University of London, United Kingdom, 1971.

[49] C. C. Paige and M. A. Saunders, *LSQR: an algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Software, **8** (1982), 43–71.

[50] C. C. Paige and M. A. Saunders, *Algorithm 583. LSQR: sparse linear equations and least squares problems*, ACM Trans. Math. Software, **8** (1982), 195–209.

[51] B. N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[52] B. N. Parlett, *A new look at the Lanczos algorithm for solving symmetric systems of linear equations*, Linear Algebra Appl., **29** (1980), 323–346.

[53] B. N. Parlett and B. Nour-Omid, *The use of a refined error bound when updating the eigenvalues of tridiagonals*, Lin. Alg. Appl., **68** (1985), 179–219.

[54] B. N. Parlett, J. K. Reid, *Tracking the progress of the Lanczos algorithm for large symmetric eigenproblems*, IMA J. Numer. Anal., **1** (1981), no. 2, 135–155.

[55] B. N Parlett and D. S. Scott *The Lanczos algorithm with selective orthogonalization*, Math. Comp., **33** (1979), no. 145, 217–238.

[56] F. P. Pijpers and M. J. Thompson, *Faster formulation of the optimally localized averages method for helioseismic inversions*, Astr. Astrophys. **262** (1992), L33–L36.

[57] A. Ruhe, *Numerical aspects of Gram-Schmidt orthogonalization of vectors*, Lin. Alg. Appl., **52/53** (1983), 591–601.

[58] Y. Saad, *On the rates of convergence of the Lanczos and the block-Lanczos Methods*, SIAM J. Numer. Anal., **17** (1980), no. 5, 687–706

[59] Y. Saad, *On the Lanczos Method for Solving Symmetric Linear Systems with Several Right-Hand Sides*, Math. Comp., **48** (1987), no. 178, 651–662.

[60] R. Schreiber and C. Van Loan, *A storage-efficient WY representation for products of Householder transformations*, SIAM J. Sci. Statist. Comput. **10** (1989), no. 1, 53–57.

[61] J. Schou, J. Christensen-Dalsgaard, and M. J. Thompson, *On Comparing Helioseismic 2-dimensional Inversion Methods*, Astrophys. J., **433** (1994), 389.

[62] Science special issue on helioseismology, Science, **272** (1996), 1281–1309.

[63] H. D. Simon, *The Lanczos algorithm with partial reorthogonalization*, Math. Comp., **42** (1984), no. 165, 115–142.

[64] H. D. Simon, *Analysis of the symmetric Lanczos algorithm with reorthogonalization methods*, Linear Algebra Appl., **61** (1984), 101–131.

[65] D. C. Sorensen, *Implicitly restarted Arnoldi/Lanczos methods for large scale eigenvalue calculations*, Dept. Comp. Appl. Math., Rice University, Houston, Texas, 1995.

[66] H. A. van der Vorst, *An iterative solution method for solving $f(A)x = b$, using Krylov subspace information obtained for the symmetric positive definite matrix A*, J. Comput. Appl. Math., **18** (1987), no. 2, 249–263.

[67] K. Wu and H. D. Simon, *A Parallel Lanczos Method for Symmetric Generalized Eigenvalue Problems*, Tech. Report., NERSC, 1997,
URL: `http://www.nersc.gov/research/SIMON/planso.html`.

[68] K. Wu and H. D. Simon, Private communication, 1998.

A Singular values of the testmatrices

In the table below we have listed the 10 largest singular values of the matrices used for the numerical experiments.

i	HELIO212	ABB313	ILLC1850	WEST0479
1	1.75416885208775	8.624571506285145	2.123342642739714	318951.7598051425
2	0.75994788932135	7.837679685693903	2.079293601886760	317252.8998362914
3	0.56464846522408	7.522436441654018	2.070148692246095	316948.9798008894
4	0.41932269628941	7.504555680595632	2.055344464000141	316847.7370186802
5	0.37725028038295	7.115057823411112	2.034954713061991	316687.7890987259
6	0.36297882213116	6.846632032905399	2.026870406060143	30383.15433419206
7	0.32834791912147	6.743829397959233	1.973716978288877	14669.17025840166
8	0.32150825430870	6.518455343678970	1.939631441087475	5277.606250923692
9	0.28286240653325	6.439690716473062	1.909188260790087	4575.849920006961
10	0.25850995888747	6.198118832238062	1.874764369104710	4244.119958839099
i	RBS480a	QH1484	MHD4800a	AF23560
1	857.482872013056	1.256792690959426e+16	362870.1170020910	645.740014579336
2	795.544455546689	1.256792690959425e+16	132097.1357652045	645.714884295259
3	778.061225081411	1.256611401591208e+16	81449.70955249084	593.345456826059
4	770.514550295336	1.256611401591189e+16	58951.80607480534	593.324126772549
5	753.309670903894	1.256608510545386e+16	46211.49691522065	582.210949117382
6	746.680238728886	1.256608510545368e+16	38005.99628606661	582.172081847832
7	740.515959817913	1.256604051085655e+16	32277.98581214598	543.273745152562
8	721.852134169162	1.256604051085654e+16	28051.84097285097	543.257878163888
9	720.643353502449	1.256603988609464e+16	24805.00862837231	532.626669256042
10	712.450713136385	1.256603988609461e+16	22232.26491274030	532.568016147591

B PROPACK: Sparse SVD and eigenvalue routines in MATLAB

B.1 Introduction

A MATLAB package PROPACK has been written to test the algorithms discussed in this paper, and to provide robust implementations of PRO and BPRO to be used as building blocks in other applications. PROPACK contains a number routines that implement the symmetric Lanczos algorithm with partial reorthogonalization (PRO) and the Lanczos bidiagonalization algorithm with partial bidiagonalization (BPRO). In addition, two driver routines laneig and lansvd are available. The function laneig is modeled after the FORTRAN 77 subroutine lanso from the latest version of the PLANSO package by Parlett et al. and uses PRO to solve the symmetric eigenvalue problem. The function lansvd is similar to laneig but uses the BPRO algorithms to compute singular values and (if requested) singular vectors of a sparse matrix. We have tried to make the interface and calling sequences to both laneig and lansvd as close as possible to those of the MATLAB 5 functions eigs and svds.

B.2 On-line documentation

In this section we have listed the on-line documentation for the four main routines in PROPACK: lanbpro, lanpro, lansvd and laneig. The documentation for the remaining routines can be found in the source code listings in the following section.

B.2.1 lansvd

```
LANSVD  Compute a few singular values and singular vectors.
   LANSVD computes singular triplets (u,v,sigma) such that
   A*u = sigma*v and  A'*v = sigma*u  using the Lanczos
   bidiagonalization algorithm with partial reorthogonalization (BPRO).

   S = LANSVD(A)
   S = LANSVD('Afun','Atransfun',M,N)

   The first input argument is either a real matrix, or a string
   containing the name of an M-file which applies a linear operator
   to the columns of a given matrix. In the latter case, the second
   input must be the name of an M-file which applies the transpose of
   the same linear operator to the columns of a given matrix,
   and the third and fourth arguments must be M and N, the dimensions
   of the problem.

   The full calling sequence is

   [U,S,V] = LANSVD(A,K,OPTIONS)
   [U,S,V] = LANSVD('Afun','Atransfun',M,N,K,OPTIONS)

   where K is the number of singular values desired.
   The OPTIONS structure specifies certain parameters in the algorithm.

    Field name      Parameter                              Default
```

```
  OPTIONS.tol     Convergence tolerance                 16*eps
  OPTIONS.lanmax  Maximal dimension of the Lanczos
                  basis.
  OPTIONS.v0      Starting vector for the Lanczos       rand(n,1)-0.5
                  iteration.
  OPTIONS.delta   Level of orthogonality among the      sqrt(eps/K)
                  Lanczos vectors.
  OPTIONS.eta     Level of orthogonality after          10*eps^(3/4)
                  reorthogonalization.
  OPTIONS.cgs     reorthogonalization method used       0
                  '0' : iterated modified Gram-Schmidt
                  '1' : iterated classical Gram-Schmidt
  OPTIONS.elr     If equal to 1 then extended local     1
                  reorthogonalization is enforced. i.e.
                  v_{j} is orthogonalized against v_{j-1}
                  and  u_{j+1} is orthogonalized against
                  u_j in every step.

 See also LANBPRO, SVDS, SVD.

References:
 R. M. Larsen, ``Lanczos bidiagonalization with partial
 reorthogonalization'',Tech. Report, Dept. of Computer Science,
 Aarhus University, 1998.

 B. N. Parlett, ``The Symmetric Eigenvalue Problem'',
 Prentice-Hall, Englewood Cliffs, NJ, 1980.

 H. D. Simon, ``The Lanczos algorithm with partial reorthogonalization'',
 Math. Comp. 42 (1984), no. 165, 115--142.
```

B.2.2 lanbpro

```
LANBPRO Lanczos bidiagonalization with partial reorthogonalization.
   LANPRO computes the Lanczos bidiagonalization of a real
   matrix using the  with partial reorthogonalization.

   [U_k,B_k,V_k,R,ierr,work] = lanbpro(A,K,R0,OPTIONS,U_old,B_old,V_old)
   [U_k,B_k,V_k,R,ierr,work] = lanbpro('Afun','Atransfun',M,N,K,R0, ...
                                       OPTIONS,U_old,B_old,V_old)

   Computes K steps of the Lanczos bidiagonalization algorithm with partial
   reorthogonalization (BPRO) with M-by-1 starting vector R0, producing a
   lower bidiagonal K-by-K matrix B_k, an N-by-K matrix V_k, an M-by-K
   matrix U_k and a M-by-1 vector such that
        A*V_k = U_k*B_k + R
   Partial reorthogonalization is used to keep the columns of V_K and U_k
   semiorthogonal:
         MAX(DIAG((EYE(K) - V_K'*V_K))) <= OPTIONS.delta
   and
         MAX(DIAG((EYE(K) - U_K'*U_K))) <= OPTIONS.delta.

   If an invariant subspace for range(A) or range(A') is found before K
```

```
  steps have been computed, LANBPRO attempts to restart the iteration with
  a random starting vector orthogonal to the invariant subspace already
  computed.

  B_k = LANBPRO(...) returns the bidiagonal matrix only.

  The first input argument is either a real matrix, or a string
  containing the name of an M-file which applies a linear operator
  to the columns of a given matrix. In the latter case, the second
  input must be the name of an M-file which applies the transpose of
  the same linear operator to the columns of a given matrix,
  and the third and fourth arguments must be M and N, the dimensions
  of then problem.

  The OPTIONS structure is used to control the reorthogonalization:
    OPTIONS.delta:  Desired level of orthogonality
                    (default = sqrt(eps/K)).
    OPTIONS.eta  :  Level of orthogonality after reorthogonalization
                    (default = 10*eps^(3/4)).
    OPTIONS.cgs  :  Flag for switching between different reorthogonalization
                    algorithms:
                     0 = iterated modified Gram-Schmidt  (default)
                     1 = iterated classical Gram-Schmidt
    OPTIONS.elr  :  If OPTIONS.elr = 1 (default) then extended local
                    reorthogonalization is enforced.

  If both R0, U_old, B_old, and V_old are provided, they must
  contain a partial Lanczos bidiagonalization of A on the form

       A V_old = U_old B_old + R0 .

  In this case the factorization is extended to dimension K x K by
  continuing the Lanczos bidiagonalization algorithm with R0 as a
  starting vector.

  The output array work contains information about the work used in
  reorthogonalizing the u- and v-vectors.
     work = [ RU  PU ]
            [ RV  PV ]
  where
     RU = Number of reorthogonalizations of U.
     PU = Number of inner products used in reorthogonalizing U.
     RV = Number of reorthogonalizations of V.
     PV = Number of inner products used in reorthogonalizing V.

  See also LANSVD, REORTH, COMPUTE_L.

References:
  R. M. Larsen, ``Lanczos bidiagonalization with partial
  reorthogonalization'',Tech. Report, Dept. of Computer Science,
  Aarhus University, 1998.

  G. H. Golub & C. F. Van Loan, "Matrix Computations",
```

```
   3. Ed., Johns Hopkins, 1996.  Section 9.3.4.

   B. N. Parlett, ``The Symmetric Eigenvalue Problem'',
   Prentice-Hall, Englewood Cliffs, NJ, 1980.

   H. D. Simon, ``The Lanczos algorithm with partial reorthogonalization'',
   Math. Comp. 42 (1984), no. 165, 115--142.
```

B.2.3 laneig

```
LANEIG  Compute a few eigenvalues and eigenvectors.
   LANEIG solves the eigenvalue problem A*v=lambda*v, when A is
   real and symmetric. Only a few eigenvalues and eigenvectors are
   computed using the symmetric Lanczos algorithm with partial
   reorthogonalization (PRO).

   [V,D] = LANEIG(A)
   [V,D] = LANEIG('Afun',N)

   The first input argument is either a real symmetric matrix, or a
   string containing the name of an M-file which applies a linear
   operator to the columns of a given matrix.  In the latter case,
   the second input argument must be N, the order of the problem.

   The full calling sequence is

   [V,D,ERR] = LANEIG(A,K,PART,OPTIONS)
   [V,D,ERR] = LANEIG('Afun',N,K,PART,OPTIONS)

   On exit ERR contains the computed error bounds.
   K is the number of eigenvalues desired and PART is a two letter
   string which specifies which part of the spectrum should be computed:

   PART            Specified eigenvalues

   'AL'            Algebraically Largest (default)
   'AS'            Algebraically Smallest
   'LM'            Largest Magnitude
   'BE'            Both Ends.  Computes k/2 eigenvalues
                   from each end of the spectrum (one more
                   from the high end if k is odd.)

   The OPTIONS structure specifies certain parameters in the algorithm.

    Field name      Parameter                              Default

    OPTIONS.tol     Convergence tolerance                  16*eps
    OPTIONS.lanmax  Maximal dimension of the Lanczos
                    basis.
    OPTIONS.v0      Starting vector for the Lanczos        rand(n,1)-0.5
                    iteration.
    OPTIONS.delta   Level of orthogonality among the       sqrt(eps/K)
                    Lanczos vectors.
```

```
    OPTIONS.eta     Level of orthogonality after          10*eps^(3/4)
                    reorthogonalization.
    OPTIONS.cgs     Reorthogonalization method used       0
                    '0' : iterated modified Gram-Schmidt
                    '1' : iterated classical Gram-Schmidt
    OPTIONS.elr     If equal to 1 then extended local     1
                    reorthogonalization is enforced, i.e.
                    q_{j+1} is orthogonalized against
                    q_{j-1} and q_j in every step.

   See also LANPRO, EIGS, EIG.

 References:
   R. M. Larsen, ``Lanczos bidiagonalization with partial
   reorthogonalization'',Tech. Report, Dept. of Computer Science,
   Aarhus University, 1998.

   B. N. Parlett, ``The Symmetric Eigenvalue Problem'',
   Prentice-Hall, Englewood Cliffs, NJ, 1980.

   H. D. Simon, ``The Lanczos algorithm with partial reorthogonalization'',
   Math. Comp. 42 (1984), no. 165, 115--142.
```

B.2.4 lanpro

```
LANPRO   Lanczos tridiagonalization with partial reorthogonalization
   LANPRO computes the Lanczos tridiagonalization of a real symmetric
   matrix using the symmetric Lanczos algorithm with partial
   reorthogonalization.

   [Q_K,T_K,R,ANORM,IERR,WORK] = LANPRO(A,K,R0,OPTIONS,Q_old,T_old)
   [Q_K,T_K,R,ANORM,IERR,WORK] = LANPRO('Afun',N,K,R0,OPTIONS,Q_old,T_old)

   Computes K steps of the Lanczos algorithm with starting vector R0,
   and returns the K x K tridiagonal T_K, the N x K matrix Q_K
   with semiorthonormal columns and the residual vector R such that

        A*Q_K = Q_K*T_K + R .

   Partial reorthogonalization is used to keep the columns of Q_K
   semiorthogonal:
        MAX(DIAG((eye(k) - Q_K'*Q_K))) <= OPTIONS.delta.

   If an invariant subspace is found before K steps have been computed,
   LANPRO attempts to restart the iteration with a random starting vector
   orthogonal to the invariant subspace already computed.

   The first input argument is either a real symmetric matrix, or a
   string containing the name of an M-file which applies a linear
   operator to the columns of a given matrix.  In the latter case,
   the second input argument must be N, the order of the problem.

   The OPTIONS structure is used to control the reorthogonalization:
```

```
    OPTIONS.delta:  Desired level of orthogonality
                    (default = sqrt(eps/K)).
    OPTIONS.eta  :  Level of orthogonality after reorthogonalization
                    (default = 10*eps^(3/4)).
    OPTIONS.cgs  :  Flag for switching between different reorthogonalization
                    algorithms:
                     0 = iterated modified Gram-Schmidt  (default)
                     1 = iterated classical Gram-Schmidt
    OPTIONS.elr  :  If OPTIONS.elr = 1 (default) then extended local
                    reorthogonalization is enforced.

  If both R0, Q_old and T_old are provided, they must contain
  a partial Lanczos tridiagonalization of A on the form

       A Q_old = Q_old T_old + R0 .

  In this case the factorization is extended to dimension K x K by
  continuing the Lanczos algorithm with R0 as starting vector.

  On exit ANORM contains an approximation to ||A||_2.
    IERR = 0  :  K steps were performed successfully.
    IERR > 0  :  K steps were performed successfully, but the algorithm
                 switched to full reorthogonalization after IERR steps.
    IERR < 0  :  Iteration was terminated after -IERR steps because an
                 invariant subspace was found.
  On exit WORK(1) contains the number of reorthogonalizations performed, and
  WORK(2) contains the number of inner products performed in the
  reorthogonalizations.

  See also LANEIG, REORTH, COMPUTE_L.

 References:
  R. M. Larsen, ``Lanczos bidiagonalization with partial
  reorthogonalization'',Tech. Report, Dept. of Computer Science,
  Aarhus University, 1998.

  G. H. Golub & C. F. Van Loan, "Matrix Computations",
  3. Ed., Johns Hopkins, 1996.  Chapter 9.

  B. N. Parlett, ``The Symmetric Eigenvalue Problem'',
  Prentice-Hall, Englewood Cliffs, NJ, 1980.

  H. D. Simon, ``The Lanczos algorithm with partial reorthogonalization'',
  Math. Comp. 42 (1984), no. 165, 115--142.
```

B.3 Source code

B.3.1 lansvd

This routine computes the singular values decomposition of a sparse matrix.

```
function [U,S,V,bnd] = lansvd(varargin)
```

```
% Rasmus Munk Larsen, DAIMI, 1998

%%%%%%%%%%%%%%%%%%%%%% Parse and check input arguments. %%%%%%%%%%%%%%%%%%%%%%
if nargin<1 | length(varargin)<1
  error('Not enough input arguments.');
end

A = varargin{1};
if ~isstr(A)
  if ~isreal(A)
    error('A must be real')
  end
  Aisfunc = 0;
  [m n] = size(A);
  if length(varargin) < 2, k=min(min(m,n),6); else k=varargin{2}; end
  if length(varargin) < 3, options = []; else options=varargin{3};end
else
  if length(varargin)<4
    error('Not enough input arguments.');
  end
  Aisfunc = 1;
  Atrans = varargin{2};
  if ~isstr(Atrans)
    error('Atransfunc must be the name of a function')
  end
  m = varargin{3};
  n = varargin{4};
  if length(varargin) < 5, k=min(min(m,n),6); else k=varargin{5}; end
  if length(varargin) < 6, options = []; else options=varargin{6}; end
end

if ~isnumeric(n) | real(abs(fix(n))) ~= n | ~isnumeric(m) | ...
      real(abs(fix(m))) ~= m | ~isnumeric(k) | real(abs(fix(k))) ~= k
  error('M, N and K must be positive integers.')
end


% Quick return for min(m,n) equal to 0 or 1 or for zero A.
if min(n,m) < 1 | k<1
  if nargout<3
    U = zeros(k,1);
  else
    U = eye(m,k); S = zeros(k,k);  V = eye(n,k);  bnd = zeros(k,1);
  end
  return
elseif min(n,m) == 1 & k>0
  if Aisfunc
    % Extract the single column or row of A
    if n==1
      A = feval(A,1);
    else
      A = feval(Atrans,1)';
    end
```

```
  end
  if nargout==1
    U = norm(A);
  else
    [U,S,V] = svd(full(A));
    bnd = zeros(k,1);
  end
  return
end

% A is the matrix of all zeros (not detectable if A is a string)
if ~Aisfunc
  if  nnz(A)==0
    if nargout<3
      U = zeros(k,1);
    else
      U = eye(m,k); S = zeros(k,k);  V = eye(n,k);  bnd = zeros(k,1);
    end
    return
  end
end

lanmax = min(m,n);
tol = 16*eps;
p = rand(m,1)-0.5;
% Parse options struct
if isstruct(options)
  c = fieldnames(options);
  for i=1:length(c)
    if any(strcmp(c(i),'p0')), p = getfield(options,'p0'); p=p(:); end
    if any(strcmp(c(i),'tol')), tol = getfield(options,'tol'); end
    if any(strcmp(c(i),'lanmax')), lanmax = getfield(options,'lanmax'); end
  end
end

% Protect against absurd options.
tol = max(tol,eps);
lanmax = min(lanmax,min(m,n));
if size(p,1)~=m
  error('p0 must be a vector of length m')
end

lanmax = min(lanmax,min(m,n));
if k>lanmax
  error('K must satisfy  K <= LANMAX <= MIN(M,N).');
end

%%%%%%%%%%%%%%%%%%%% Here begins the computation  %%%%%%%%%%%%%%%%%%%%%%
ksave = k;
neig = 0; nrestart=-1;
j = min(k+max(8,k),lanmax);
U = []; V = []; B = []; anorm = []; work = zeros(2,2);
```

```
while neig < k

  %%%%%%%%%%%%%%%%%%%%% Compute Lanczos bidiagonalization %%%%%%%%%%%%%%%%%
  if isnumeric(A)
    [U,B,V,p,ierr,w] = lanbpro(A,j,p,options,U,B,V,anorm);
  else
    [U,B,V,p,ierr,w] = lanbpro(A,Atrans,m,n,j,p,options,U,B,V,anorm);
  end
  work= work + w;
%printf('j=, work = %e, work(FRO) = %e',j,sum([m n]*work(:,2)),(m+n)*j*(j+1)/2)

  if ierr<0 % Invariant subspace of dimension -ierr found.
    j = -ierr;
  end

  %%%%%%%%%%%%%%%%%% Compute singular values and error bounds %%%%%%%%%%%%%%%%
  % Analyze B
  resnrm = norm(p);
  [P,S,Q] = svd(full([B;[zeros(1,j-1),resnrm]]),0);   S = diag(S);
  bot = min([P(end,1:j);Q(end,1:j)])';

  % Use Largest Ritz value to estimate ||A||_2. This might save some
  % reorth. in case of restart.
  anorm=S(1);

  % Set simple error bounds
  bnd = resnrm*abs(bot);

  % Examine gap structure and refine error bounds
  bnd = refinebounds(S.^2,bnd,n*eps*anorm);

  %%%%%%%%%%%%%%%%%%% Check convergence criterion %%%%%%%%%%%%%%%%%%%%
  jj = 1;  neig = 0;
  while jj<=min(j,k)
    if (bnd(jj) <= tol*abs(S(jj)))
      jj = jj+1;
      neig = neig + 1;
    else
      jj = k+1;
    end
  end

  %%%%%%%%%% Check whether to stop or to extend the Krylov basis? %%%%%%%%%%
  if ierr<0 % Invariant subspace found
    if j<k
      warning(['Invariant subspace of dimension ',num2str(j-1),' found.'])
    end
    break;
  end
  if j>=lanmax  % Maximal dimension of Krylov subspace reached. Bail out
    if neig<ksave
      warning(['Maximum dimension of Krylov subspace exceeded prior',...
            ' to convergence.']);
```

```
    end
    break;
  end

  % Increase dimension of Krylov subspace
  %
  if neig>1
    j = j + ceil(min(20,max(2,((j-1)*(k-neig+1))/(2*(neig+1)))));
  elseif neig<k
    j = j + ceil(min(20,max(8,(k-neig)/2)));
  end
  j = min(j,lanmax);
  nrestart = nrestart + 1;
end



%%%%%%%%%%%%%%%% Lanczos converged (or failed). Prepare output %%%%%%%%%%%%%%%
k = min(ksave,j);

if nargout>2
  j = size(B,2);
  % Compute eigenvectors
%  [P,S,Q] = svd([full(B);[zeros(1,j-1),resnrm]],0); S = diag(S);
  S = S(1:k);
  Q = Q(:,1:k);
  P = P(:,1:k);
  % Compute and normalize Ritz vectors (overwrites U and V to save memory).
  if resnrm~=0
    U = [U,p/resnrm]*P;
  else
    U = U*P(1:j,:);
  end
  V = V*Q;
  for i=1:k
    nq = norm(V(:,i));
    if isfinite(nq) & nq~=0 & nq~=1
      V(:,i) = V(:,i)/nq;
    end
    nq = norm(U(:,i));
    if isfinite(nq) & nq~=0 & nq~=1
      U(:,i) = U(:,i)/nq;
    end
  end
end

% Pick out desired part the spectrum
S = S(1:k);
bnd = bnd(1:k);

if nargout<3
  U = S;
  S = B; % Undocumented feature -  for checking B.
```

```
else
  S = diag(S);
end
```

B.3.2 lanbpro

This function implements the BPRO algorithm.

```
function [U,B_k,V,p,ierr,work] = lanbpro(varargin)

% Rasmus Munk Larsen, DAIMI, 1998.

% Check input arguments.
if nargin<1 | length(varargin)<2
  error('Not enough input arguments.');
end
narg=length(varargin);

A = varargin{1};
if ~isstr(A)
  if ~isreal(A)
    error('A must be real')
  end
  Aisfunc = 0;
  [m n] = size(A);
  k=varargin{2};
  if narg < 3, p = rand(m,1)-0.5; else p=varargin{3}; end
  if narg < 4, options = []; else options=varargin{4}; end
  if narg > 4
    if narg<7
      error('All or none of U_old, B_old and V_old must be provided.')
    else
      U = varargin{5}; B_k = varargin{6}; V = varargin{7};
    end
  else
    U = []; B_k = []; V = [];
  end
  if narg > 7, anorm=varargin{8}; else anorm = []; end
else
  if narg<5
    error('Not enough input arguments.');
  end
  Aisfunc = 1;
  Atrans = varargin{2};
  if ~isstr(Atrans)
    error('Atransfunc must be the name of a function')
  end
  m = varargin{3};
  n = varargin{4};
  if ~isreal(n) | abs(fix(n)) ~= n | ~isreal(m) | abs(fix(m)) ~= m
    error('M and N must be positive integers.')
  end
  k=varargin{5};
```

```
  if narg < 6, p = rand(m,1)-0.5; else p=varargin{6}; end
  if narg < 7, options = []; else options=varargin{7}; end
  if narg > 7
    if  narg < 10
      error('All or none of U_old, B_old and V_old must be provided.')
    else
      U = varargin{8}; B_k = varargin{9}; V = varargin{10};
    end
  else
    U = []; B_k = []; V=[];
  end
  if narg > 10, anorm=varargin{11}; else anorm = [];  end
end

% Quick return for min(m,n) equal to 0 or 1.
if min(m,n) == 0
   U = [];  B_k = [];  V = [];  p = [];  ierr = 0;  work = zeros(2,2);
   return
elseif  min(m,n) == 1
  if isnumeric(A)
    U = 1;  B_k = A;  V = 1;  p = 0; ierr = 0; work = zeros(2,2);
  else
    U = 1;  B_k = feval(A,1); V = 1; p = 0; ierr = 0; work = zeros(2,2);
  end
  if nargout<3
    U = B_k;
  end
  return
end

% Set options.
delta = sqrt(eps/k); % Desired level of orthogonality.
eta = 10*eps^(3/4);     % Level of orth. after reorthogonalization.
cgs = 0;                % Flag for switching between iterated MGS and CGS.
elr = 1;                % Flag for switching extended local
                        % reorthogonalization on and off.
LL = 0;                 % Number of initial Lanczos vectors to reorthogonalize
                        % against.

% Parse options struct
if ~isempty(options) & isstruct(options)
  c = fieldnames(options);
  for i=1:length(c)
    if strmatch(c(i),'delta'), delta = getfield(options,'delta');  end
    if strmatch(c(i),'eta'), eta = getfield(options,'eta'); end
    if strmatch(c(i),'cgs'), cgs = getfield(options,'cgs'); end
    if strmatch(c(i),'ll'), LL = getfield(options,'ll'); end
    if strmatch(c(i),'elr'), elr = getfield(options,'elr'); end
  end
end

if isempty(anorm)
  anorm = []; est_anorm=1;
```

```
else
  est_anorm=0;
end

%%%%%%%%%%%%%%%%%%%%% Here begins the computation  %%%%%%%%%%%%%%%%%%%%%%

% Conservative statistical estimate on the size of round-off terms.
% Notice that {\bf u} == eps/2.
eps1 = sqrt(max(m,n))*eps/2;

% Prepare for Lanczos iteration.
npu = 0; npv = 0; ierr = 0;
if isempty(U)
  V = zeros(n,k); U = zeros(m,k);
  beta = zeros(k+1,1); alpha = zeros(k,1);
  beta(1) = norm(p);
  % Initialize MU/NU-recurrences for monitoring loss of orthogonality.
  nu = zeros(k,1); mu = zeros(k,1);
  mu(1)=1; nu(1)=1;

  numax = zeros(k,1); mumax = zeros(k,1);
  force_reorth = 0;  nreorthu = 0; nreorthv = 0;
  j0 = 1;
else
  j = size(U,2); % Size of existing factorization
  alpha = zeros(k+1,1);  beta = zeros(k+1,1);
  alpha(1:j) = diag(B_k);  beta(2:j) = diag(B_k,-1);
  beta(j+1) = norm(p);
  % Reorthogonalize p.
  if j<k & beta(j+1)*delta < anorm*eps1,
    fro = 1;
    ierr = j;
  end
  int = 1:j;
  [p,beta(j+1),rr] = reorth(U,p,beta(j+1),int,0.5,cgs);
  npu =  rr*j;  nreorthu = 1;  force_reorth= 1;

  % Compute Gerscgorin bound on ||B_k||_2
  if est_anorm
    anorm = sqrt(norm(B_k'*B_k,1));
  end
  mu = eps1*ones(k,1); nu = ones(k,1);
  numax = zeros(k,1); mumax = zeros(k,1);
  force_reorth = 1;  nreorthu = 0; nreorthv = 0;
  j0 = j+1;
end

LLint = [];
if delta==0
  fro = 1; % The user has requested full reorthogonalization.
else
  fro = 0;
end
```

```
% Perform Lanczos bidiagonalization partial reorthogonalization.
for j=j0:k
  if beta(j) ~=0
    U(:,j) = p/beta(j);
  else
    U(:,j) = p;
  end
  %%%%%%%%%% Lanczos step to generate v_j. %%%%%%%%%%%%%

  if j==1
    if ~Aisfunc
      r = A'*U(:,1);
    else
      r = feval(Atrans,U(:,1));
    end
    alpha(1) = norm(r);
    if est_anorm
      anorm = alpha(1);
    end
  else
    if ~Aisfunc
      r = A'*U(:,j) - beta(j)*V(:,j-1);
    else
      r = feval(Atrans,U(:,j)) - beta(j)*V(:,j-1);
    end
    % Extended local reorthogonalization
    if elr
      for i=1:1
        t = V(:,j-1)'*r;
        r = r - V(:,j-1)*t;
        if beta(j) ~= 0
          beta(j) = beta(j) + t;
        end
      end
    end
    alpha(j) = norm(r);

    if est_anorm
      if j==2
        anorm = max(anorm,sqrt(alpha(1)^2+beta(2)^2+alpha(2)*beta(2)));
      else
        anorm = max(anorm,sqrt(alpha(j-1)^2+beta(j)^2+alpha(j-1)*beta(j-1)+ ...
            alpha(j)*beta(j)));
      end
    end
    % Possibly orthogonalize against first LL Lanczos vectors.
    if LL>0
      LLint = 1:min(j-1,LL);
      [r,alpha(j),rr] = reorth(V,r,alpha(j),LLint,0.5,cgs);
      npv = npv + rr*LL;
      nu(LLint) = eps1;  % Reset nu for orthogonalized vectors.
      % so they do not contribute to new nu-bounds.
```

```
    end

    if ~fro
      % Update estimates of the level of orthogonality for the
      %  columns 1 through j-1 in V.
      nu = update_nu(nu,mu,j,eps1,alpha,beta,anorm);
      nu(LLint) = eps1;  % Reset nu for orthogonalized vectors.
      numax(j) = max(abs(nu(1:j-1)));
    end

    % IF level of orthogonality is worse than delta THEN
    %    Reorthogonalize v_j against some previous  v_i's, 0<=i<j.
    if ( fro | numax(j) > delta | force_reorth )
      % Decide which vectors to orthogonalize against:
      if fro
        int = LL+1:j-1;
      elseif force_reorth==0
        int = compute_int(nu,j-1,delta,eta,LL,0,0);
      end
      % Else use int from last reorth. to avoid spillover from mu_{j-1}
      % to nu_j.

      % Reorthogonalize v_j using Modified Gramm-Schmidt.
      [r,alpha(j),rr] = reorth(V,r,alpha(j),int,0.5,cgs);
      npv = npv + rr*length(int); % number of inner products.
      nu(int) = eps1;  % Reset nu for orthogonalized vectors.
      if force_reorth==0
        force_reorth = 1; % Force reorthogonalization of u_{j+1} to avoid
        % spillover into nu_{j+1}
      else
        force_reorth = 0;
      end
      nreorthv = nreorthv + 1;
    end
  end


  % Check for convergence of invariant subspace or failure to
  % maintain semiorthogonality
  if alpha(j) < max(n,m)*anorm*eps  & j<k,
    % If alpha is "small" we deflate by setting it
    % to 0 and attempt to restart with a basis for a new
    % invariant subspace by replacing r with a random starting vector:
    alpha(j) = 0;
    bailout = 1;
    for attempt=1:3
      r = rand(m,1)-0.5;
      if ~Aisfunc
        r = A'*r;
      else
        r = feval(Atrans,r);
      end
      nrm=sqrt(r'*r); % not necessary to compute the norm accurately here.
```

```
      int = 1:j-1;
      [r,nrmnew,rr] = reorth(V,r,nrm,int,0.5,cgs);
      npv = npv + rr*length(int(:));        nreorthv = nreorthv + 1;
      if nrmnew > 0
        % A vector numerically orthogonal to span(Q_k(:,1:j)) was found.
        % Continue iteration.
        bailout=0;
        break;
      end
    end
    if bailout
      j = j-1;
      ierr = -j;
      break;
    else
      r=r/nrmnew; % Continue with new normalized r as starting vector.
      nu(1:j-1) = eps1;
      force_reorth = 1;
      if delta>0
        fro = 0;    % Turn off full reorthogonalization.
      end
    end
  elseif  j<k & ~fro & anorm*eps1 > delta*alpha(j)
    fro = 1;
    ierr = j;
  end

  if alpha(j) ~= 0
    V(:,j) = r/alpha(j);
  else
    V(:,j) = r;
  end

  %%%%%%%%%% Lanczos step to generate u_{j+1}. %%%%%%%%%%%%%
  if ~Aisfunc
    p = A*V(:,j) - alpha(j)*U(:,j);
  else
    p = feval(A,V(:,j)) - alpha(j)*U(:,j);
  end
  % Extended local reorthogonalization
  if elr
    for i=1:1
      t = U(:,j)'*p;
      p = p - U(:,j)*t;
      if alpha(j) ~= 0
        alpha(j) = alpha(j) + t;
      end
    end
  end
  beta(j+1) = norm(p);

  % Possibly orthogonalize against first LL Lanczos vectors.
  if LL>0
```

```
  LLint = 1:min(j,LL);
  [p,beta(j+1),rr] = reorth(U,p,beta(j+1),LLint,0.5,cgs);
  npu = npu + rr*LL;
  mu(LLint) = eps1; % Reset nu for vectors that have been orthogonalized.
  % so they do not contribute to new nu-bounds.
end

if est_anorm
  % We should update estimate of ||A||  before updating mu - especially
  % important in the first step for problems with large norm since alpha(1)
  % may be a severe underestimate!
  if j==1
    anorm = max(anorm,sqrt(alpha(1)^2+beta(2)^2));
  else
    anorm = max(anorm,sqrt(alpha(j)^2+beta(j+1)^2 + alpha(j)*beta(j)));
  end
end


if ~fro
  % Update estimates of the level of orthogonality for the columns of V.
  mu = update_mu(mu,nu,j,eps1,alpha,beta,anorm);
  mu(LLint) = eps1;
  mumax(j) = max(abs(mu(1:j)));
end

% IF level of orthogonality is worse than delta THEN
%    Reorthogonalize u_{j+1} against some previous  u_i's, 0<=i<=j.
if (fro | mumax(j) > delta | force_reorth)
  % Decide which vectors to orthogonalize against.
  if fro
    int = LL+1:j;
  elseif force_reorth==0
    int = compute_int(mu,j,delta,eta,LL,0,0);
  end
  % Else use int from last reorth. to avoid spillover from nu to mu.

  % Reorthogonalize u_{j+1} using Modified Gramm-Schmidt.
  [p,beta(j+1),rr] = reorth(U,p,beta(j+1),int,0.5,cgs);
  npu = npu + rr*length(int);  nreorthu = nreorthu + 1;

  % Reset mu to epsilon.
  mu(int) = eps1;
  if force_reorth==0
    force_reorth = 1; % Force reorthogonalization of v_{j+1}.
  else
    force_reorth = 0;
  end
end

% Check for convergence of invariant subspace or failure to
% maintain semiorthogonality
if beta(j+1) < max(m,n)*anorm*eps  & j<k,
```

```
    % If beta is "small" we deflate by setting it
    % to 0 and attempt to restart with a basis for a new
    % invariant subspace by replacing p with a random starting vector:
    beta(j+1) = 0;
    bailout = 1;
    for attempt=1:3
      p = rand(n,1)-0.5;
      if ~Aisfunc
        p = A*p;
      else
        p = feval(A,p);
      end
      nrm=sqrt(p'*p); % not necessary to compute the norm accurately here.
      int = 1:j;
      [p,nrmnew,rr] = reorth(U,p,nrm,int,0.5,cgs);
      npu = npu + rr*length(int(:));  nreorthu = nreorthu + 1;
      if nrmnew > 0
        % A vector numerically orthogonal to span(Q_k(:,1:j)) was found.
        % Continue iteration.
        bailout=0;
        break;
      end
    end
    if bailout
      ierr = -j;
      break;
    else
      p=p/nrmnew; % Continue with new normalized p as starting vector.
      mu(1:j) = eps1;
      force_reorth = 1;
      if delta>0
        fro = 0;    % Turn off full reorthogonalization.
      end
    end
  elseif  j<k & ~fro & anorm*eps1 > delta*beta(j+1)
    fro = 1;
    ierr = j;
  end
end

k=min(k,j);
B_k = spdiags([alpha(1:k) [beta(2:k);0]],[0 -1],k,k);
if nargout==1
  U = B_k;
else
  U = U(:,1:k);
  V = V(:,1:k);
end
if nargout>5
  work = [[nreorthu,npu];[nreorthv,npv]];
end
```

B.3.3 laneig

This routine computes the eigenvalues and eigenvectors of a sparse symmetric matrix.

```
function [V,D,bnd] = laneig(A,nin,k,part,options)

% Rasmus Munk Larsen, DAIMI, 1998

%%%%%%%%%%%%%%%%%%%%% Parse and check input arguments. %%%%%%%%%%%%%%%%%%%%%%

if ~isstr(A)
  if nargin<1
    error('Not enough input arguments.');
  end
  [m n] = size(A);
  if m~=n | ~isequal(A,A') | ~isreal(A)
    error('A must be real symmetric')
  end
  if nargin < 4 | isempty(part)
    options = [];
  else
    options = part;
  end
  if nargin < 3 | isempty(k), part = 'AL'; else, part = k; end
  if nargin < 2 | isempty(nin), k = min(min(lanmax,n),5); else, k = nin; end
else
  if nargin<2
    error('Not enough input arguments.');
  end
  n = nin;
  if nargin < 5 | isempty(options)
    options.tol = 16*eps;
    options.lanmax = n;
    options.v0 = rand(n,1)-0.5;
  end
  if nargin < 4 | isempty(part), part = 'AL'; end
  if nargin < 3 | isempty(k), k = min(n,5);  end
end
if ~isnumeric(k) | real(abs(fix(k)))~=k | ~isnumeric(n) | real(abs(fix(n)))~=n
  error('Input arguments N and K must be positive integers.')
end
if ~isstr(part)
  error('Input argument PART must be a string.')
end

% Quick return for n==1 or n==0
if n == 0
   V = [];
   D = [];
   bnd =[];
   return
end
if n == 1
  if isnumeric(A)
```

```
    D = A;
    V = 1;
    bnd = 0;
  else
    D = feval(A,1);
    V = 1;
    dnb = 0;
  end
  if nargout<2
    V=D;
  end
  return
end

lanmax = n;
tol = 16*eps;
r = rand(n,1)-0.5;
% Parse options struct
if ~isempty(options) & isstruct(options)
  c = fieldnames(options);
  for i=1:length(c)
    if strmatch(c(i),'v0'), r = getfield(options,'v0'); r=r(:); end
    if strmatch(c(i),'tol'), tol = getfield(options,'tol'); end
    if strmatch(c(i),'lanmax'), lanmax = getfield(options,'lanmax'); end
  end
end

% Protect against absurd arguments.
tol = max(tol,eps);
lanmax = min(lanmax,n);
if size(r,1)~=n
  error('v0 must be a vector of length n')
end
lanmax = min(lanmax,n);
if k>lanmax
  error('K must satisfy  K <= LANMAX <= N.');
end
ksave = k;
neig = 0; nrestart=-1;
if strcmp(part,'AL') | strcmp(part,'AS')
  j = min(2*k+2,lanmax);
else
  j = min(k+1,lanmax);
end


%%%%%%%%%%%%%%%%%%%%% Here begins the computation  %%%%%%%%%%%%%%%%%%%%%%
V = []; T = []; anorm = []; work = zeros(1,2); rnorm=0;
while neig < k
  %%%%%%%%%%%%%%%%%%%%% Compute Lanczos tridiagonalization %%%%%%%%%%%%%%%%%
  if rnorm > eps*anorm
    j = j-1;
  end
```

```
j = min(lanmax,j+mod(j,2));
% "Trick" to avoid unwanted zero eigenvalues when laneig is used for
% SVD calculations. (Nothing to if lanmax is odd, though.)

if isnumeric(A)
  [V,T,r,anorm,ierr,w] = lanpro(A,j,r,options,V,T,anorm);
else
  [V,T,r,anorm,ierr,w] = lanpro(A,n,j,r,options,V,T,anorm);
end
work= work + w;

if ierr<0 % Invariant subspace of dimension -ierr found.
  j = -ierr;
end

%%%%%%%%%%%%%%%%%%% Compute eigenvalues and error bounds %%%%%%%%%%%%%%%%%%
% Analyze T
rnorm = norm(r);
if rnorm > eps*anorm
  % Might as well use the extra information...
  [D,top,bot,err] = tqlb([full(diag(T));0],full([0;diag(T,1);rnorm]));
else
  [D,top,bot,err] = tqlb([full(diag(T))],full([0;diag(T,1)]));
end
[D,I] = sort(D);
bot = bot(I);
if err>0
  printf(['TQLB failed. Eigenvalue no. %i did not converge in 30', ...
        ' iterations'],err);
end

% Set simple error bounds
bnd = rnorm*abs(bot);

% Use ||T_k||_2 for accurate estimate of ||A||_2.
% In case of restart it usually saves some reorthogonalizations.
anorm = max(D);

% Estimate gap structure and refine error bounds
bnd = refinebounds(D,bnd,n*eps*anorm);

if rnorm > eps*anorm
  j = j+1;
end

%%%%%%%%%%%%%%%%%%%% Check convergence criterion %%%%%%%%%%%%%%%%%%%%
% Reorder eigenvalues according to PART
switch part
  case 'AS'
    IPART = 1:j;
  case 'AL'
    IPART = j:-1:1;
  case 'LM'
```

```
      [dummy,IPART] = sort(-abs(D));
    case 'BE'
      mid = ceil(k/2);
      IPART = [[1:min(j,mid-1)],[max(1,j-mid+1):j]];
  end
  D = D(IPART);  bnd = bnd(IPART);

  % Check if enough have converged.
  jj = 1;  neig = 0;
  while jj<=min(j,k)
    if (bnd(jj) <= tol*abs(D(jj)))
      jj = jj+1;
      neig = neig + 1;
    else
      jj = k+1;
    end
  end

  %%%%%%%%%%% Check whether to stop or to extend the Krylov basis? %%%%%%%%%%
  if ierr<0 % Invariant subspace found
    if j<k
      warning(['Invariant subspace of dimension ',num2str(j-1),' found.'])
    end
    break;
  end
  if j>=lanmax % Maximal dimension of Krylov subspace reached => Bail out!
    if neig<ksave
      warning(['Maximum dimension of Krylov subspace exceeded prior',...
            ' to convergence.']);
    end
    break;
  end
  % Increase dimension of Krylov subspace and try again.
  if neig>1
    j = j + ceil(min(20,max(2,((j-1)*(k-neig+1))/(2*(neig+1)))));
  elseif neig<k
    j = j + ceil(min(20,max(8,(k-neig)/2)));
  end
  j = min(j,lanmax);
  nrestart = nrestart + 1;
end

%%%%%%%%%%%%%%%% Lanczos converged (or failed). Prepare output %%%%%%%%%%%%%%%
k = min(ksave,j);
if nargout>1
  % Compute eigenvectors
  [Q,D] = eig(full(T)); D = diag(D);
  [D,I] = sort(D);
  D = D(IPART(1:k));
  Q = Q(:,I(IPART(1:k)));
  % Compute and normalize Ritz vectors (overwrite V to save memory).
  V = V*Q;
  for i=1:k
```

```
    nq = norm(V(:,i));
    if isfinite(nq) & nq~=0
      V(:,i) = V(:,i)/nq;
    end
  end
end

% Pick out desired part of the spectrum
D = D(1:k);
bnd = bnd(1:k);

if nargout<2
  V = D;
else
  D = diag(D);
end
```

B.3.4 lanpro

This routine implements the PRO algorithm.

```
function [Q_k,T_k,r,anorm,ierr,work] = lanpro(A,nin,kmax,r,options,...
    Q_k,T_k,anorm)


% Rasmus Munk Larsen, DAIMI, 1998

% Check input arguments.
if ~isstr(A)
  Aisfunc = 0;
  if nargin<1, error('Not enough input arguments.');  end
  [m n] = size(A);
  if m~=n | ~isequal(A,A') | ~isreal(A)
    error('A must be real symmetric')
  end
  if nargin<7 | isempty(T_k),
    anorm = []; est_anorm=1;
  else,
    anorm = T_k; est_anorm=0;
  end
  if nargin<6,  Q_k=[]; T_k=[]; else,  T_k = Q_k; Q_k = options; end
  if nargin<4 | isempty(r),  options = []; else,  options = r;  end
  if nargin<3 | isempty(kmax),
    r = rand(n,1)-0.5;
  else
    r = kmax;
  end
  if nargin<2 | isempty(nin);  kmax = max(10,n/10); else,  kmax = nin;  end
else
  Aisfunc = 1;
  if nargin<2
    error('Not enough input arguments.');
  end
```

```
  % Check input functions and parse to create an internal object
  % if an explicit expression is given.
  [A, msg] = fcnchk(A);
  if ~isempty(msg)
    error(msg);
  end
  n = nin;
  if nargin<8 | isempty(anorm), anorm = []; est_anorm=1; else est_anorm=0; end
  if nargin<7,  Q_k=[]; T_k=[]; end
  if nargin<5 | isempty(options),  options = [];          end
  if nargin<4 | isempty(r),  r = rand(n,1)-0.5;   end
  if nargin<3 | isempty(kmax);  kmax = max(10,n/10); end
end

% Set options.
delta = sqrt(eps/kmax); % Desired level of orthogonality.
eta = 10*eps^(3/4);     % Level of orth. after reorthogonalization.
cgs = 0;                % Flag for switching between iterated CGS and MGS.
elr = 1;                % Flag for switching extended local
                        % reorthogonalization on and off.

% Parse options struct
if ~isempty(options) & isstruct(options)
  c = fieldnames(options);
  for i=1:length(c)
    if strmatch(c(i),'delta'), delta = getfield(options,'delta');  end
    if strmatch(c(i),'eta'), eta = getfield(options,'eta'); end
    if strmatch(c(i),'cgs'), cgs = getfield(options,'cgs'); end
    if strmatch(c(i),'elr'), elr = getfield(options,'elr'); end
  end
end
np = 0;  nr = 0; ierr=0;

%%%%%%%%%%%%%%%%%%%%% Here begins the computation  %%%%%%%%%%%%%%%%%%%%%%

% Rule-of-thumb estimate on the size of round-off terms:
eps1 = sqrt(n)*eps/2; % Notice that {\bf u} == eps/2.

% Prepare Lanczos iteration
if isempty(Q_k) % New Lanczos tridiagonalization.
  % Allocate space
  alpha = zeros(kmax+1,1);  beta = zeros(kmax+1,1);
  Q_k = zeros(n,kmax);
  q = zeros(n,1); beta(1)=norm(r);
  omega = zeros(kmax,1); omega_max = omega;  omega_old = omega;
  omega(1) = 0;   force_reorth= 0;
  j0 = 1;
else            % Extending existing Lanczos tridiagonalization.
  j = size(Q_k,2); % Size of existing factorization
  % Allocate space
  Q_k = [Q_k zeros(n,kmax-j)];
  alpha = zeros(kmax+1,1);  beta = zeros(kmax+1,1);
  alpha(1:j) = diag(T_k);  beta(2:j) = diag(T_k,-1);
```

```
  q = Q_k(:,j);
  % Reorthogonalize r.
  beta(j+1) = norm(r);
  if j<kmax & beta(j+1)*delta < anorm*eps1,
    fro = 1;
    ierr = j;
  end
  int = 1:j;
  [r,beta(j+1),rr] = reorth(Q_k,r,beta(j+1),int,0.5,cgs);
  np = rr*j;    nr = 1;   force_reorth= 1;
  % Compute Gerscgorin bound on ||T_k||_2 as SQRT(||T_k'*T_k||_1)
  if est_anorm
    anorm = sqrt(norm(T_k'*T_k,1));
  end
  omega = eps1*ones(kmax,1); omega_max = omega;  omega_old = omega;
  j0 = j+1;
end

if delta==0
  fro = 1; % The user has requested full reorthogonalization.
else
  fro = 0;
end

for j=j0:kmax,
  % Lanczos Step:
  q_old = q;
  if beta(j)==0
    q = r;
  else
    q = r / beta(j);
  end
  Q_k(:,j) = q;
  if ~Aisfunc
    u = A*q;
  else
    u = feval(A,q);
  end
  r = u - beta(j)*q_old;
  alpha(j) = q'*r;
  r = r - alpha(j)*q;

  % Extended local reorthogonalization:
  if elr
    if  j==1
      t1=0;
      for i=1:2
        t = q'*r;
        r = r-q*t;
        t1 = t1+t;
      end
      alpha(j) = alpha(j) + t1;
    elseif j>1
```

```
    t1 = q_old'*r;
    t2 = q'*r;
    r = r  - (q_old*t1 + q*t2); % Add small terms together first to
    if beta(j)~=0               % reduce risk of cancellation.
      beta(j) = beta(j) + t1;
    end
    alpha(j) = alpha(j) + t2;
  end
end
beta(j+1) = sqrt(r'*r); % Quick and dirty estimate.

% Update Gersgorin estimate of ||T_k|| if required
if  est_anorm & beta(j+1)~=0
  anorm = update_gbound(anorm,alpha,beta,j);
end

% Update omega-recurrence
if j>1 & ~fro & beta(j+1)~=0
  [omega,omega_old] = update_omega(omega,omega_old,j,alpha,beta,...
      eps1,anorm);
  omega_max(j) = max(abs(omega));
end

% Reorthogonalize if required
if j>1 & (fro  | force_reorth | omega_max(j)>delta) & beta(j+1)~=0
  if fro
    int = 1:j;
  else
    if force_reorth == 0
      force_reorth= 1; % Do forced reorth to avoid spill-over from q_{j-1}.
      int = compute_int(omega,j,delta,eta,0,0,0);
    else
      force_reorth= 0;
    end
  end
  [r,beta(j+1),rr] = reorth(Q_k,r,beta(j+1),int,0.5,cgs);
  omega(int) = eps1;
  np = np + rr*length(int(:));    nr = nr + 1;
else
  beta(j+1) = norm(r); % compute norm accurately.
end

if  j<kmax & beta(j+1) < n*anorm*eps  ,
  % If beta is "small" we deflate by setting the off-diagonals of T_k
  % to 0 and attempt to restart with a basis for a new
  % invariant subspace by replacing r with a random starting vector:
  beta(j+1) = 0;
  bailout = 1;
  for attempt=1:3
    r = rand(n,1)-0.5;
    if ~Aisfunc
      r = A*r;
    else
```

```
        r = feval(A,r);
      end
      nrm=sqrt(r'*r); % not necessary to compute the norm accurately here.
      int = 1:j;
      [r,nrmnew,rr] = reorth(Q_k,r,nrm,int,0.5,cgs);
      np = np + rr*length(int(:));    nr = nr + 1;
      if nrmnew > 0
        % A vector numerically orthogonal to span(Q_k(:,1:j)) was found.
        % Continue iteration.
        bailout=0;
        break;
      end
    end
    if bailout
      ierr = -j;
      break;
    else
      r=r/nrmnew; % Continue with new normalized r as starting vector.
      force_reorth = 1;
      omega(:) = eps1;
      if delta>0
        fro = 0;    % Turn off full reorthogonalization.
      end
    end
  elseif j<kmax & ~fro & beta(j+1)*delta < anorm*eps1,
    % If anorm*eps1/beta(j+1) > delta then  omega(j+1) will
    % immediately exceed delta, and thus forcing a reorth. to occur at the
    % next step. The components of omega will mainly be determined
    % by the initial value and not the recurrence, and therefore we
    % cannot tell reliably which components exceed eta => we might
    % as well switch to full reorthogonalization to avoid trouble.
    % The user is probably trying to determine pathologically
    % small ( < sqrt(eps)*||A||_2 ) eigenvalues.
    fro = 1;
    ierr = j;
  end
end

% Set up tridiagonal T_k in sparse matrix data structure.
T_k = spdiags([[beta(2:j);0] alpha(1:j) beta(1:j)],-1:1,j,j);
if nargout<2
  Q_k = T_k;
else
  Q_k = Q_k(:,1:j);
  work = [nr np];
end
```

B.3.5 reorth

This routine implements iterated classical or iterated modified Gram-Schmidt reorthogonalization.

```
function [r,normr,nre] = reorth(Q,r,normr,index,alpha,method)
```

```
%REORTH   Reorthogonalize a vector using iterated Gram-Schmidt
%
%   [R_NEW,NORMR_NEW,NRE] = reorth(Q,R,NORMR,INDEX,ALPHA,METHOD)
%   reorthogonalizes R against the subset of columns of Q given by INDEX.
%   If INDEX==[] then R is reorthogonalized all columns of Q.
%   If the result R_NEW has a small norm, i.e. if norm(R_NEW) < ALPHA*NORMR,
%   then a second reorthogonalization is performed. If the norm of R_NEW
%   is once more decreased by  more than a factor of ALPHA then R is
%   numerically in span(Q(:,INDEX)) and a zero-vector is returned for R_NEW.
%
%   If method==0 then iterated modified Gram-Schmidt is used.
%   If method==1 then iterated classical Gram-Schmidt is used.
%
%   The default value for ALPHA is 0.5.
%   NRE is the number of reorthogonalizations performed (1 or 2).

% References:
%  Aake Bjorck, "Numerical Methods for Least Squares Problems",
%  SIAM, Philadelphia, 1996, pp. 68-69.
%
%  J. W. Daniel, W. B. Gragg, L. Kaufman and G. W. Stewart,
%  ``Reorthogonalization and Stable Algorithms Updating the
%  Gram-Schmidt QR Factorization'', Math. Comp.,  30 (1976), no.
%  136, pp. 772-795.
%
%  B. N. Parlett, ``The Symmetric Eigenvalue Problem'',
%  Prentice-Hall, Englewood Cliffs, NJ, 1980. pp. 105-109

%  Rasmus Munk Larsen, DAIMI, 1998.

% Check input arguments.
if nargin<2
  error('Not enough input arguments.')
end
[n k1] = size(Q);

if nargin<3 | isempty(normr)
  normr = norm(r);
end

if nargin<4 | isempty(index)
  k = k1;
  index = [1:k]';
  simple = 1;
else
  k = length(index);
  if k==k1 & index(:)==[1:k]'
    simple = 1;
  else
    simple = 0;
  end
end
```

```
if nargin<5 | isempty(alpha)
  alpha=0.5; % This choice garanties that
             % || Q^T*r_new - e_{k+1} ||_2 <= 2*eps*||r_new||_2.
             % cf. Kahans ``twice is enough'' statement proved in
             % Parletts book.
end
if nargin<6 | isempty(method)
   method = 0;
end

normr_old=normr;

if method==1
  if simple
    r = r - Q*(Q'*r);
  else
    r = r - Q(:,index)*(Q(:,index)'*r);
  end
else
  for i=index, r = r-Q(:,i)*(Q(:,i)'*r); end
end
nre = 1;
normr = norm(r);
if normr < alpha*normr_old
  if method==1
    if simple
      r = r - Q*(Q'*r);
    else
      r = r - Q(:,index)*(Q(:,index)'*r);
    end
  else
    for i=index, r = r-Q(:,i)*(Q(:,i)'*r); end
  end
  normr_old=normr;
  normr = norm(r);
  if normr < alpha*normr_old
    % r is in span(Q) to full accuracy => accept r = 0 as the new vector.
    r = zeros(n,1);
    normr = 0;
  end
  nre = 2;
end
```

B.3.6 compute_L

This function computes the index sets $\mathcal{L}^{\mu}_{j+1}$ and $\mathcal{L}^{\nu}_{j}$ that determine which vectors to include in the reorthogonalization.

```
function L = compute_L(mu,j,delta,eta,LL,strategy,extra)
%COMPUTE_L:  Determine which Lanczos vectors to reorthogonalize against.
%
%      L = compute_L(mu,eta,LL,strategy,extra))
```

```
%
%   Strategy 0: Orthogonalize vectors v_{i-r-extra},...,v_{i},...v_{i+s+extra}
%               with nu>eta, where v_{i} are the vectors with  mu>delta.
%   Strategy 1: Orthogonalize all vectors v_{r-extra},...,v_{s+extra} where
%               v_{r} is the first and v_{s} the last Lanczos vector with
%               mu > eta.
%   Strategy 2: Orthogonalize all vectors with mu > eta.
%
%   Notice: The first LL vectors are excluded since the new Lanczos
%   vector is already orthogonalized against them in the main iteration.

%  Rasmus Munk Larsen, DAIMI, 1998.

if (delta<eta)
  error('DELTA should satisfy DELTA >= ETA.')
end
switch strategy
  case 0
    I0 = find(abs(mu(1:j))>=delta);
    if length(I0)==0
      [mm,I0] = max(abs(mu));
    end
    L = zeros(j,1);
    for i = 1:length(I0)
      for r=I0(i):-1:1
        if L(r)==1 | abs(mu(r))<eta,
          break;
        else
          L(r) = 1;
        end
      end
      L(max(1,r-extra+1):r) = 1;
      for s=I0(i)+1:j
        if L(s)==1 | abs(mu(s))<eta,
          break;
        else
          L(s) = 1;
        end
      end
      L(s:min(j,s+extra-1)) = 1;
    end
    if LL>0
      L(1:LL) = 0;
    end
    L = find(L);
  case 1
    L=find(abs(mu(1:j))>eta);
    L = max(LL+1,min(L)-extra):min(max(L)+extra,j);
  case 2
    L=find(abs(mu(1:j))>=eta);
end
```

B.3.7 update_mu, update_nu

These functions implement the updating rules for $\mu_{j+1,i}$ and ν_{ij}.

```
function mu = update_mu(muold,nu,j,eps1,alpha,beta,anorm)

% UPDATE_MU:  Update the mu-recurrence for the u-vectors.
%
%   mu_new = update_mu(mu,nu,j,eps1,alpha,beta,anorm)

%  Rasmus Munk Larsen, DAIMI, 1998.

mu = muold;
T =  anorm*eps1;
if j==1
  mu(1) = T / beta(2);
else
  mu(1) = alpha(1)*nu(1) - alpha(j)*mu(1);
  mu(1) = (mu(1) + sign(mu(1))*T) / beta(j+1);
  % Vectorized version of loop:
  k = 2:j-1;
  mu(k) = alpha(k).*nu(k) + beta(k).*nu(k-1) - alpha(j).*mu(k);
  mu(k) = (mu(k) + sign(mu(k)).*T) ./ beta(j+1);

  mu(j) = beta(j)*nu(j-1);
  mu(j) = (mu(j) + sign(mu(j))*T) / beta(j+1);
end
mu(j+1) = 1;


function nu = update_nu(nuold,mu,j,eps1,alpha,beta,anorm)

% UPDATE_MU:  Update the nu-recurrence for the v-vectors.
%
%  nu_new = update_nu(nu,mu,j,eps1,alpha,beta,anorm)

%  Rasmus Munk Larsen, DAIMI, 1998.

nu = nuold;
T = anorm*eps1;
k = 1:(j-1);
nu(k) = beta(k+1).*mu(k+1) + alpha(k).*mu(k) - beta(j).*nu(k);
nu(k) = (nu(k) + sign(nu(k)).*T) ./ alpha(j);
nu(j) = 1;
```

B.3.8 update_omega

This routine updates the ω-recurrence in the PRO algorithm.

```
function [omega,omega_old] = update_omega(omega, omega_old, j, ...
    alpha,beta,eps1,anorm)
% UPDATE_OMEGA:  Update Simon's omega_recurrence for the Lanczos vectors.
%
% [omega,omega_old] = update_omega(omega, omega_old,j,eps1,alpha,beta,anorm)
```

```
%

% Rasmus Munk Larsen, DAIMI, 1998.

% Estimate of contribution to roundoff errors from A*v
%   fl(A*v) = A*v + f,
% where ||f|| \approx eps1*||A||.
% For a full matrix A, a rule-of-thumb estimate is eps1 = sqrt(n)*eps.
T = eps1*anorm;
binv = 1/beta(j+1);

omega_old = omega;
% Update omega(1) using omega(0)==0.
omega_old(1)= beta(2)*omega(2)+ (alpha(1)-alpha(j))*omega(1) -  ...
    beta(j)*omega_old(1);
omega_old(1) = binv*(omega_old(1) + sign(omega_old(1))*T);
% Update remaining components.
k=2:j-2;
omega_old(k) = beta(k+1).*omega(k+1) + (alpha(k)-alpha(j)).*omega(k) ...
     + beta(k).*omega(k-1) - beta(j)*omega_old(k);
omega_old(k) = binv*(omega_old(k) + sign(omega_old(k))*T);
omega_old(j-1) = binv*T;
% Swap omega and omega_old.
temp = omega;
omega = omega_old;
omega_old = omega;
omega(j) =  eps1;
```

B.3.9 refinebounds

This routine refines the error bounds of the eigenvalues of a symmetric matrix based on the gap-structure.

```
function [bnd,gap] = refinebounds(D,bnd,tol1)
%REFINEBONDS  Refines error bounds for Ritz values based on gap-structure
% If the GAP for the ith eigenvalues is defined as
%      GAP(i) == min_{i ~= k} | LAMBDA(i) - LAMBDA(k) |
%  then BND = REFINEBOUNDS(LAMBDA,BND,TOL1) refines the error bounds for
%  isolated eigenvalues where GAP(i) > BND(i) by replacing
%  BND(i) with BND(i)^2/gap.
%
%  Eigenvalues closer than TOL1 are treated as a cluster.

% Rasmus Munk Larsen, DAIMI, 1998

j = length(D);
% Sort eigenvalues to use interlacing theorem correctly
[D,PERM] = sort(D);
bnd = bnd(PERM);

% Massage error bounds for very close Ritz values
eps34 = sqrt(eps*sqrt(eps));
[y,mid] = max(bnd);
```

```
for l=[-1,1]
  for i=((j+1)-l*(j-1))/2:l:mid-l
    if abs(D(i+l)-D(i)) < eps34*abs(D(i))
      if bnd(i)>tol1 & bnd(i+l)>tol1
        bnd(i+l) = pythag(bnd(i),bnd(i+l));
        bnd(i) = 0;
      end
    end
  end
end
% Refine error bounds
gap = inf*ones(1,j);
gap(1:j-1) = min([gap(1:j-1);[D(2:j)-bnd(2:j)-D(1:j-1)]']);
gap(2:j) = min([gap(2:j);[D(2:j)-D(1:j-1)-bnd(1:j-1)]']);
gap = gap(:);
I = find(gap>bnd);
bnd(I) = bnd(I).*(bnd(I)./gap(I));

bnd(PERM) =  bnd; % Undo permutation.
```

B.3.10 update_gbound

This routine updates the estimate of the 2-norm of a tridiagonal matrix.

```
function anorm = update_gbound(anorm,alpha,beta,j)
%UPDATE_GBOUND   Update Gerscgorin estimate of 2-norm
%  ANORM = UPDATE_GBOUND(ANORM,ALPHA,BETA,J) updates the Gersgorin bound
%  for the tridiagonal in the Lanczos process after the J'th step.
%  Applies Gerscgorins circles to T_k'*T_k instead of T_k itself
%  since this gives a tighter bound.

% Rasmus Munk Larsen, DAIMI, 1998

if j==1 % Apply Gerscgorin circles to T_k'*T_k to estimate || A ||_2
  i=j;
  % scale to avoid overflow
  scale = max(abs(alpha(i)),abs(beta(i+1)));
  alpha(i) = alpha(i)/scale;
  beta(i+1) = beta(i+1)/scale;
  anorm = 1.01*scale*sqrt(alpha(i)^2+beta(i+1)^2 + abs(alpha(i)*beta(i+1)));
elseif j==2
  i=1;
  % scale to avoid overflow
  scale = max(max(abs(alpha(1:2)),max(abs(beta(2:3)))));
  alpha(1:2) = alpha(1:2)/scale;
  beta(2:3) = beta(2:3)/scale;

  anorm = max(anorm, scale*sqrt(alpha(i)^2+beta(i+1)^2 + ...
      abs(alpha(i)*beta(i+1) + alpha(i+1)*beta(i+1)) + ...
      abs(beta(i+1)*beta(i+2))));
  i=2;
  anorm = max(anorm,scale*sqrt(abs(beta(i)*alpha(i-1) + alpha(i)*beta(i)) + ...
      beta(i)^2+alpha(i)^2+beta(i+1)^2 +  ...
```

```
      abs(alpha(i)*beta(i+1))) );
elseif j==3
  % scale to avoid overflow
  scale = max(max(abs(alpha(1:3)),max(abs(beta(2:4)))));
  alpha(1:3) = alpha(1:3)/scale;
  beta(2:4) = beta(2:4)/scale;
  i=2;
  anorm = max(anorm,scale*sqrt(abs(beta(i)*alpha(i-1) + alpha(i)*beta(i)) + ...
      beta(i)^2+alpha(i)^2+beta(i+1)^2 +  ...
      abs(alpha(i)*beta(i+1) + alpha(i+1)*beta(i+1)) + ...
      abs(beta(i+1)*beta(i+2))) );
  i=3;
  anorm = max(anorm,scale*sqrt(abs(beta(i)*beta(i-1)) + ...
      abs(beta(i)*alpha(i-1) + alpha(i)*beta(i)) + ...
      beta(i)^2+alpha(i)^2+beta(i+1)^2 +  ...
      abs(alpha(i)*beta(i+1))) );
else
  % Avoid scaling, which is slow. At j>3 the estimate is usually quite good
  % so just make sure that anorm is not made infinite by overflow.
  i = j-1;
  anorm1 = sqrt(abs(beta(i)*beta(i-1)) + ...
      abs(beta(i)*alpha(i-1) + alpha(i)*beta(i)) + ...
      beta(i)^2+alpha(i)^2+beta(i+1)^2 +  ...
      abs(alpha(i)*beta(i+1) + alpha(i+1)*beta(i+1)) + ...
      abs(beta(i+1)*beta(i+2)));
  if isfinite(anorm1)
    anorm = max(anorm,anorm1);
  end
  i = j;
  anorm1 = sqrt(abs(beta(i)*beta(i-1)) + ...
      abs(beta(i)*alpha(i-1) + alpha(i)*beta(i)) + ...
      beta(i)^2+alpha(i)^2+beta(i+1)^2 +  ...
      abs(alpha(i)*beta(i+1)));
  if isfinite(anorm1)
    anorm = max(anorm,anorm1);
  end
end
```