Algorithm 844: Computing Sparse Reduced-Rank Approximations to Sparse Matrices

MICHAEL W. BERRY and SHAKHINA A. PULATOVA
University of Tennessee, Knoxville
and
G. W. STEWART
University of Maryland, College Park

In many applications—latent semantic indexing, for example—it is required to obtain a reduced rank approximation to a sparse matrix A. Unfortunately, the approximations based on traditional decompositions, like the singular value and QR decompositions, are not in general sparse. Stewart [(1999), 313–323] has shown how to use a variant of the classical Gram–Schmidt algorithm, called the quasi–Gram-Schmidt–algorithm, to obtain two kinds of low-rank approximations. The first, the SPQR, approximation, is a pivoted, Q-less QR approximation of the form $(XR_{11}^{-1})(R_{11}\ R_{12})$, where X consists of columns of A. The second, the SCR approximation, is of the form the form $A \cong XTY^{\mathrm{T}}$, where X and Y consist of columns and rows A and T, is small. In this article we treat the computational details of these algorithms and describe a MATLAB implementation.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse, structured, and very large systems (direct and iterative method)*

General Terms: Algorithms

Additional Key Words and Phrases: Sparse approximations, Gram–Schmidt algorithm, MATLAB

1. INTRODUCTION

In a number of applications [Berry et al. 1999; Jiang and Berry 2000; Stuart and Berry 2003; Berry and Martin 2004] one is given a large matrix A and wishes

The research of M.W. Berry and S. A. Pulatova was supported in part by the National Science Foundation under grant CISE-EIA-99-72889. The research of G. W. Stewart was supported in part by the National Science Foundation under grant CCR0204084. Part of this work was performed as faculty appointee at the Mathematical and Computational Sciences Division of the National Institute for Standards and Technology.
Authors' addresses: M. W. Berry and S. A. Pulatova, Department of Computer Science, 203 Claxton Complex, University of Tennessee, Knoxville, TN 37996; email: {berry, pulatova}@cs.utk.edu; G. W. Stewart, Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; email: stewart@cs.umd.edu.

to find a reduced-rank approximation to A. This approximation is invariably expressed in the form

$$A \cong XTY^{\mathrm{T}} \tag{1}$$

where X and Y are full-rank matrices and T is nonsingular (T may be the identity matrix). When A is $m \times n$ and T is of order k, this approximation requires $(m + n + k)k$ words to store, as opposed to mn for the full A. Moreover, the matrix-vector product Ax requires $(m + n + k)k$ additions and multiplications to compute, as opposed, again, to mn additions and multiplications for the full A. Clearly, if k is small, great savings are to be had by using the reduced-rank approximation (1).

A widely used reduced-rank approximation is the truncated singular value decomposition, which is known to be optimal in the sense that the Frobenius norm $\|A - XTY^{\mathrm{T}}\|$ is minimized. There are stable direct methods for its computation; however, these methods compute the full decomposition and are not suitable for very large matrices. Fortunately, there are iterative methods that produce the approximation (1) without having to compute the full SVD. These methods require only the formation of matrix-vector products and do not alter A.

An alternative is the pivoted QR decomposition, which generally gives results comparable to the SVD (see, e.g., Stewart [1980] or Stewart [1998, §5.2]). For large A, the Gram–Schmidt algorithm can be adapted to compute this decomposition. Again, A is not altered, and the principle operations are matrix-vector multiplications. This article is concerned with elaborations of this approach to reduced-rank approximations.

When A is large and sparse the situation is not as simple. For A, the storage and operation counts given above become proportional to the number of nonzero elements in A. Since the factors X, T, and Y are generally not sparse, the storage and operation counts for the approximation remain the same. Thus as k increases, we will reach a point where it becomes necessary to abandon the factored form. Note that we do not have the ability to choose k, since the accuracy required of the approximation, which depends on k, is governed by the application.

In this article we are going to describe two approximations based on the pivoted QR decomposition that produce approximations in which X or both X and Y are sparse. The first approximation is called *sparse pivoted QR approximation* (SPQR). It is computed by an algorithm, called the *quasi-Gram–Schmidt* algorithm, that produces a factorization in which X consists of a selection of columns of A. In the second approximation, called the *sparse column-row (SCR) approximation,* X consists of a selection of the columns of A, and Y consists of a selection of the rows of A, so that when A is sparse so are both X and Y. These methods were first described by Stewart [1999], and the quasi-Gram–Schmidt method has been analyzed in Stewart [2004].

At this point we should mention the sparse low-rank approximation of Zhang, Zha, and Simon [Zhang et al. 2002]. The idea is to approximate the dominant singular vectors by sparse vectors x and y and form $\hat{A} = A - \delta xy^{\mathrm{T}}$, where δ is chosen to minimize the Frobenius of $\hat{A}$. The process is then repreated

recursively on $\hat{A}$, until a satisfactorily accurate approximation is obtained. Experimental results are promising; and the algorithm should be considered a serious alternative to the approximations given here.

The purpose of this article is to give the computational details leading to the accompanying MATLAB functions for the SPQR and the SCR approximations. In the next section we introduce the pivoted QR decomposition. In Section 3 we derive the quasi-Gram–Schmidt method and apply it to the computation of the sparse pivoted QR approximation. In Section 4 we show how to compute the SCR approximation. We also show how it can be applied to an information retrieval process known as latent semantic indexing. The implementation details for our algorithms are described in Section 5. In Section 6 we compare the SPQR approximation with the singular value decomposition. Finally, in Section 7 we discuss some sparsity issues that arise in producing implementations in compiled programming languages like C or Fortran. The MATLAB programs are listed in appendices.

Throughout this article, $\|\cdot\|$ will denote the Frobenius norm defined by

$$\|A\|^2 = \sum_{i,j} a_{ij}^2,$$

and $\|\cdot\|_2$ the spectral norm defined by

$$\|A\|_2 = \max_{\|x\|=1} \|Ax\|.$$

2. THE PIVOTED QR FACTORIZATION

As above, let A be an $m \times n$ matrix, not necessarily sparse. A pivoted QR (PQR) factorization has the form

$$AP = QR, \tag{2}$$

where P is a permutation matrix, Q is orthonormal, and R is upper triangular. The exact form depends on the the sizes of m and n. If $m \geq n$, then Q is $m \times n$ and R is $n \times n$. If $m < n$, then Q is $m \times m$ and R is $m \times n$. Although our algorithms apply to both cases, for ease of exposition we will assume that $m \geq n$ in what follows.

A rank k approximation to A can be obtained by partitioning the factorization (2). Let $B = AP$ and write

$$\begin{pmatrix} B_1^{(k)} & B_2^{(k)} \end{pmatrix} = \begin{pmatrix} Q_1^{(k)} & Q_2^{(k)} \end{pmatrix} \begin{pmatrix} R_{11}^{(k)} & R_{12}^{(k)} \\ 0 & R_{22}^{(k)} \end{pmatrix}, \tag{3}$$

where $B_1^{(k)}$ has k columns. Then our approximation is

$$\tilde{B}^{(k)} = Q_1^{(k)} \begin{pmatrix} R_{11}^{(k)} & R_{12}^{(k)} \end{pmatrix}. \tag{4}$$

Note that

$$B - \tilde{B}^{(k)} = Q_2^{(k)} \begin{pmatrix} 0 & R_{22}^{(k)} \end{pmatrix}.$$

Since $Q_2^{(k)}$ is orthonormal, the error in $\tilde{B}^{(k)}$ as an approximation to B is

$$\left\| B - \tilde{B}^{(k)} \right\| = \left\| R_{22}^{(k)} \right\|. \tag{5}$$

We will not compute the entire decomposition (3). Rather we will bring in columns of A one at a time and use each to compute an additional column of Q and row of R. Thus at the end of the kth step of this algorithm we will have computed the approximation (4).

The process of selecting columns is called *column pivoting*, or for short simply *pivoting*. The order in which the columns are selected determines the permutation P. Equation (5) suggests that at the beginning of the kth step we should choose the column of A in such a way as to make $\|R_{22}^{(k)}\|$ small. The classical choice is to bring in the column of A that corresponds to the column of $R_{22}^{(k-1)}$ of largest norm. (For more on this choice see Björck [1996]; Stewart [1998].)

Surprisingly, we can implement this strategy without computing $R_{22}^{(k-1)}$ itself. Consider the partition (3), in which the superscripts (k) are replaced by $(k-1)$. Let b_j and r_j denote the jth columns of B and R. Because Q is orthonormal, we have

$$\|b_j\| = \|r_j\|. \tag{6}$$

Now for $j \geq k$ partition

$$r_j = \begin{pmatrix} r_1^{(j)} \\ r_2^{(j)} \end{pmatrix}$$

where $r_1^{(j)}$ has $k-1$-components. Thus $r_1^{(j)}$ is the jth column of $R_{12}^{(k-1)}$, and $r_2^{(j)}$ is the jth column of $R_{22}^{(k-1)}$. It then follows that

$$\left\| r_2^{(j)} \right\|^2 = \|b_j\|^2 - \left\| r_1^{(j)} \right\|^2 = \|b_j\|^2 - r_{1j}^2 - r_{2j}^2 - \cdots - r_{k-1,j}^2. \tag{7}$$

Thus at each stage we can compute the squares of the norms of the columns of $R_{22}^{(k-1)}$. Moreover, the sum of these numbers is $\|R_{22}^{(k-1)}\|^2$, so that by (5) we get, almost for free, the value of the norm of the error in our reduced-rank approximation. This number can be used to determine when we have a satisfactorily accurate reduced-rank approximation.

Unfortunately, the expression (7) has a dark side. If $\|r_2^{(j)}\|^2$ is small compared to $\|b_j\|^2$, there will be cancellation in the computation of $\|r_2^{(j)}\|^2$. In particular, in IEEE double-precision arithmetic, if $\|r_2^{(j)}\|^2 \leq 10^{-16}\|b_j\|^2$ we can expect no accuracy in the computed value. On taking square roots we find that we can use the formula (7) only when

$$\left\| r_2^{(j)} \right\| > 10^{-8}\|b_j\|.$$

This means that if all the columns of A have norm one, we cannot reliably compute reduced-rank approximation that is more accurate than 10^{-8}. However, this accuracy is usually more than enough.

3. THE QUASI-GRAM–SCHMIDT METHOD

In this section we will describe the quasi-Gram–Schmidt method. We will begin with a description of the classical Gram–Schmidt method.

Suppose we have a QR factorization

$$B = QR \tag{8}$$

of B and wish to compute a QR factorization

$$(B \;\; a) = (Q \;\; q)\begin{pmatrix} R & r \\ 0 & \rho \end{pmatrix}$$

of $(B \;\; a)$. The second column of this equality gives us the relation

$$a = Qr + \rho q.$$

Since $Q^{\mathrm{T}}Q = I$ and $Q^{\mathrm{T}}q = 0$, we have

$$r = Q^{\mathrm{T}}a. \tag{9}$$

Since $\|q\| = 1$, we have

$$\rho = \|a - Qr\| \tag{10}$$

and

$$q = \rho^{-1}(a - Qr). \tag{11}$$

Equations (9), (10), and (11) are effectively an algorithm for extending our original QR factorization.

Unfortunately, cancellation in the formation of $a - Qr$ can cause the computed q to be far from orthogonal to the columns of Q. The cure for this problem is *reorthogonalization*, in which the process is repeated on $a - Qr$. Specifically, we have the following algorithm, in which we use MATLAB notation.

```
1. r = Q'*a
2. q = a - Q*r
3. s = Q'*q
4. r = r + s
5. q = q - Q*s
6. rho = norm(q)
7. q = q/rho
```
(12)

Typically, this algorithm produces a q that is orthogonal to the columns of Q to working accuracy.[1]

We can use this algorithm to compute a PQR factorization of A simply by selecting columns of A and updating the QR factorization of B. To start the process off, one sets $R_{11}^{(1)} = \rho = \|a\|$, and $Q_1^{(1)} = \rho^{-1}a$, where a is the first column

[1]In extremely unlikely cases the algorithm may produce a zero q at step 2 or in which case special action must be taken (see Stewart [1998, §5.1.4]). In practice, most programs that use Gram–Schmidt orthogonalization ignore this problem.

selected from the columns of A. In this way we compute the decompositions

$$B_1^{(k)} = Q_1^{(k)} R_{11}^{(k)},$$

where $R_{11}^{(k)}$ is $k \times k$.

A problem with this algorithm is that it only computes the factor $R_{11}^{(k)}$ in (3). However it is easy to see that row k of $R_{12}^{(k)}$ is simply $q_k^{\mathrm{T}} \hat{A}$, where q_k is the kth column of $Q_1^{(k)}$ and $\hat{A}$ consists of the $n-k$ columns of A that are not in $B_1^{(k)}$. If n is large, this computation may be the most expensive part of the algorithm. Note that even if we do not want the $R_{12}^{(k)}$ we must still form (and discard) the product $q_k^{\mathrm{T}} \hat{A}$ in order to compute the column norms of $R_{22}^{(k)}$ as described in the last section.

Returning now to the Gram–Schmidt algorithm, we note that even if A is sparse, Q is in general not sparse. If m is very large, we may be unable to store Q. To circumvent this problem, we observe that it follows from (8) that

$$Q = BR^{-1}.$$

Consequently, we can form the product `Q'*a` in (12) by the following algorithm.

```
1. d = a'*B
2. r = (d/R)'
```

Similarly we can form the product `Q*r` by

```
1. p = R\r
2. q = B*p
```

This leads to the following *quasi-Gram–Schmidt step* (in which we have put the code for the classical Gram-Schmidt step on the right).

```
1.  d = a'*B                    r = Q'*a
2.  r = (d/R)'                  q = a - Q*r
3.  p = R\r
4.  q = a - B*p
5.  d = q'*B                    s = Q'*q
6.  s = (d/R)'                                        (13)
7.  r = r + s                   r = r + s
8.  p = R\s                     q = q - Q*s
9.  q = q - B*s
10. rho = norm(q)               rho = norm(q)
```

This code computes only `r` and `rho`—the quantities needed to update `R`. It does not compute `q` in the form `q/rho`, as does the classical Gram–Schmidt algorithm. Instead, `q` is defined by the relation

$$q = (B \;\; a) \begin{pmatrix} R & r \\ 0 & \rho \end{pmatrix}^{-1} = \rho^{-1}(a - BR^{-1}r), \tag{14}$$

and is so computed in our algorithms.

The quasi-Gram–Schmidt step can be applied successively to columns of A, as described above for the classical Gram–Schmidt algorithm, to produce a pivoted, Q-less PQR factorization, which we will call a *sparse-PQR (SPQR) factorization.*

We will call the corresponding approximation $(B_1^{(k)} R_{11}^{(k)-1})(R_{11}^{(k)} \; R_{12}^{(k)})$ the *SPQR approximation.* The algorithm not only dispenses with the storage for Q, but it replaces dense products involving Q with sparse products involving columns of A. The only strictly dense operations involve R_{11} and R_{12}. But since the order of $(R_{11} \; R_{12})$ is $k \times n$, if $m \gg n$ these operations account for little of the total work.

Once again there is a dark side—there may be a progressive loss of orthogonality in the matrix BR^{-1}. However, an analysis of the quasi-Gram–Schmidt algorithm [Stewart 2004] shows that the loss of orthogonality is proportional to the condition number $\|R\|\|R^{-1}\|$ of R, which is usually good enough.

A nice feature of the SPQR approximation (and QR, approximations in general) is that having computed an approximation of order k, one has immediately all the approximations of order $\ell < k$. Simply, work with the first ℓ columns of B and rows of R.

4. SPARSE COLUMN-ROW APPROXIMATIONS

When $m \gg n$, the SPQR approximation is satisfactory. But when m and n are nearly equal, the storage of R becomes a problem. We can circumvent this problem at the cost of performing another factorization.

Specifically, first apply the quasi-Gram–Schmidt algorithm to the columns of A to get a representative set of columns X of A and an upper triangular matrix R corresponding to R_{11}. Let the error in the corresponding reduced-rank decomposition be ϵ_{col}. Now apply the same algorithm to A^{T} to get a representative set Y^{T} of rows and another upper triangular matrix S. Let the error be ϵ_{row}. We then seek a matrix T such that

$$\|A - XTY^{\mathrm{T}}\|^2 = \min .$$

In Stewart [1999] it is shown that the minimizer is

$$T = R^{-1}R^{-\mathrm{T}}(X^{\mathrm{T}}AY)S^{-1}S^{-\mathrm{T}}.$$

Moreover,

$$\|A - XTY^{\mathrm{T}}\|^2 \leq \epsilon_{\text{col}}^2 + \epsilon_{\text{row}}^2. \tag{15}$$

We will call this approximation a sparse column-row approximation, or for short an SCR approximation. Such approximations are economical to use. For example, to compute $y = XTY^{\mathrm{T}}x$ we compute $r = Y^{\mathrm{T}}x$, $s = Tr$, and $y = Xs$. This requires two sparse matrix-vector multiplications and one dense matrix-vector multiplication in which the matrix is small.

It may happen that X and Y do not have the same number of columns, in which case T will not be square. This causes no problems in matrix-vector multiplications.

Some care must be taken in computing the matrix T. The crux of the matter is to form $X^{\mathrm{T}}AY$ correctly. If, for example, m is large and we first calculate AY, we end up with a large, potentially full matrix. The cure for this problem is to partition Y by columns, writing

$$X^{\mathrm{T}}A(y_1 \; y_2 \; \cdots y_k) .$$

We can then calculate $X^{\mathrm{T}}AY$ column by column as follows.

```
1. T = [];
2. for j=1:k
3.     T = [T, X'*(A*Y(:,j))];
4. end
```

A variant of this decomposition may be useful in latent semantic indexing (LSI), a device for retrieving documents from a query vector of terms [Berry and Browne 1999; Berry et al. 1995, 1999; Deerwester et al. 1990]. Briefly (here we follow Deerwester et al. [1990]), one starts with a term-document matrix A whose (i, j)-element is the number of times term i occurs in document j. One then calculates the singular value approximation

$$A = U_k \Sigma_k V_k^{\mathrm{T}}. \tag{16}$$

In the parlance of LSI, the columns of U_k are called term vectors and columns of V_k^{T} are called document vectors. Given a query vector q of terms, one computes a corresponding document vector by the formula

$$d = \Sigma_k^{-1} U_k^{\mathrm{T}} q$$

and compares it with the columns of V_k to determine which columns are related to the the query vector q. For example, one might compute the cosines of the angles between d and the columns of V_k and choose the columns corresponding to the larger ones.

We can rewrite the XTY^{T} in the form of (16). Specifically,

$$XTY^{\mathrm{T}} = (XR^{-1})(R^{-\mathrm{T}}X^{\mathrm{T}}AYS^{-1})(S^{-\mathrm{T}}Y^{\mathrm{T}}) \equiv PWQ^{\mathrm{T}}.$$

Now mathematically, P and Q are orthogonal. Consequently, if we compute the singular value decomposition $W = M\Sigma N^{\mathrm{T}}$ of W and set

$$U = PM \quad \text{and} \quad V = QN,$$

then U and V are orthonormal, and

$$XTY^{\mathrm{T}} = U\Sigma V^{\mathrm{T}}. \tag{17}$$

We can use this decomposition as described above to perform LSI. Of course we do not explicitly form U and V; rather we keep and apply them in factored form:

$$U = XR^{-1}M \quad \text{and} \quad V = YS^{-1}N.$$

It should be stressed that the relation between (17) and (16) is purely formal. It is an open question whether LSI performed using the former will share the good properties of ordinary LSI. Theorem 6.1 below encourages us to conjecture that it will.

5. A MATLAB IMPLEMENTATION

In this section we will describe a MATLAB function `spqr` to compute SPQR approximations. The basic algorithm is simple and the chief implementation problem is how to package it. We begin by looking at the input parameters.

The essential input is the matrix `A` and a tolerance `tol` to tell when to stop the factorization. Although the algorithm always terminates after a finite number of steps, if `tol` is too small, `spqr` may be committed to performing an unacceptably large number of operations. For this reason, a third parameter `maxcol` puts an upper bound on the number of columns of A to be used in the approximation. Of course, one can always set `maxcol` greater than or equal to n, it which case it has no effect. The program terminates at the first step k for which $\|R_{22}^{(k)}\| < \texttt{tol}$. Consequently, if `tol` is zero, `spqr` is forced to include `maxcol` columns of A.

Thus the basic calling sequence for `spqr` is

```
spqr(A, tol, maxcol)
```

Although `A` is presumed to be a MATLAB sparse matrix, `spqr` also works when `A` is dense. We will now turn to the output parameters.

When `spqr` finishes we need to know four things.

(1) The number columns of A involved in the approximation.
(2) The matrix $(R_{11}\ R_{12})$.
(3) The relation of the columns of $(R_{11}\ R_{12})$ to those of A.
(4) The error in the approximation.

The first is returned in the output parameter `ncols`. The second in the output matrix `R`.

The third and fourth items are connected with the way `spqr` implements pivoting. The function begins with two arrays: `colx`, which is initialized to `1:n`, and `norms`, which is initialized so that `norms(j)` is the norm of the `j`th column of `A`. At step `k`, `spqr` determines the first index `j` $\geq$ `k` for which `norms(j)` is maximal and swaps components `j` and `k` of both `colx` and `norms`, along with the corresponding columns of `R`. The column `A(:,colx(j))` is then used to advance the approximation. After the quasi-Gram–Schmidt orthogonalization has been computed the elements of the `k`th row of $R_{22}^{(k)}$ are computed and used to downdate `norms(k+1:n)`. The error in the current approximation is also computed and stored in `norms(k)`.

From this description it follows that on return

$$B = A(:, \texttt{colx}),$$

which provides the relation between R and A. Moreover, the error in the approximation is `norms(ncols)`. However, we get a little more. For `j` $\leq$ `ncols`, the arrays `colx` and `R(1:j,:)` contain the SPQR approximation associated with `A(:,colx(1:j))`, and by construction its error is `norms(j)`. Thus by setting `tol` to zero, we can track the quality of all the approximations from `1` to `maxcols`.

The function `spqr` has three optional input arguments used to fine tune the decomposition. The first `fullR` has a default value of `1` (`true`). If it is present and `0` (`false`), then only $R_{11}^{(\texttt{ncols})}$ is computed. This is useful when the primary concern is with the space spanned by the columns `A(:,colx(1:ncols))`. The second optional parameter `pivot` has the default value `1`. If it is present and `0` pivoting is suppressed—the columns of `A` are processed in their natural order. Finally, the optional parameter `cn` (for compute norms) has a default value of `1`.

If it is present and `fullR | pivot | cr` is zero, then the computation of `norms` is suppressed and on return `norms = []`.

Thus the final calling sequence is

```
[ncols, R, colx, norms] = spqr(A, tol, maxcol, fullR, pivot, cn)
```

in which `fullR`, `pivot`, and `cn` are optional. For a concise summary see the prologue to `spqr`.

6. COMPARISON WITH THE SVD

In this section we will make some theoretical and timing comparisons between the quasi-QR and the SVD reduced-rank approximations. SVD approximations are rightly regarded as the ones to beat. Gaps in the singular values reveal numerical rank with great reliability, and the reduced rank-approximations it produces are optimal. However, as we shall see, it can be expensive to compute.

Since the SVD is so highly regarded, it is sometimes objected that other approximations may not reproduce the row and column spaces from the SVD to sufficient accuracy. This is particularly important in applications where we are not interested in the approximations themselves but in the subspaces they define. We are now going to show that if any reduced-rank approximation is accurate then it contains good approximations to the singular vectors corresponding to large singular values.

THEOREM 6.1. *Let $A = XY^{\mathrm{T}} + E$. Let $\mathcal{X}$ be the space spanned by the columns of X, and $\mathcal{Y}$ be the space spanned by the columns of Y. Let $\sigma > 0$ be a singular value of A with normalized left and right singular vectors u and v, so that $Av = \sigma u$ and $u^{\mathrm{T}}A = \sigma v^{\mathrm{T}}$. Then*

$$\sin \angle(u, \mathcal{X}),\ \sin \angle(v, \mathcal{Y}) \leq \frac{\|E\|_2}{\sigma}. \tag{18}$$

PROOF. We will establish the first inequality, the second being established similarly. Let $X_\perp$ be an orthonormal basis for the orthogonal complement of $\mathcal{X}$. Then $\|X_\perp^{\mathrm{T}} u\|$ is the sine of the angle between u and $\mathcal{X}$ [Stewart 2001, §4.2.a]. Now

$$X_\perp^{\mathrm{T}} A v = \sigma X_\perp^{\mathrm{T}} u,$$

and

$$X_\perp^{\mathrm{T}} A v = X_\perp^{\mathrm{T}} X Y^{\mathrm{T}} v + X_\perp^{\mathrm{T}} E v = X_\perp^{\mathrm{T}} E v,$$

since by construction $X_\perp^{\mathrm{T}} X = 0$. It then follows that $\sigma X_\perp^{\mathrm{T}} u = X_\perp^{\mathrm{T}} E v$, whence $\|X_\perp^{\mathrm{T}} u\|_2 \leq \|E\|_2/\sigma$, which is just the first inequality in (18). □

This theorem says that if a reduced-rank approximation is accurate, then its column space must contain accurate approximations to the left singular vectors corresponding to singular values that are large compared to $\|E\|$. An analogous statement is true of the row space and the right singular vectors.

Turning now to timing examples, we will use matrices A of order $n = 10{,}000$ generated by the MATLAB function `sprandn`, which produces a "random" sparse matrix with a given distribution of singular values (for more, issue the command

`edit sprandn` in MATLAB). The first distribution we consider is given by the MATLAB statement

```
s = logspace(0, -6, n)
```

Thus the common logarithms of the singular values are equally spaced between 0 and −6. For `ncr=10:5:40` we timed the call

```
[ncr, cx, nr, rx, T, rsd] = cra(A, 1e-5, ncr);
```

which will produce an approximation of rank `ncr`. We also timed the MATLAB function

```
[U, S, V] = svds(A, ncr);
```

which produces the matrices required to compute an SVD approximation of rank `ncr`.

The results are summarized in the following table, in which the time is reported in seconds.

ncr	SPQR	SVD
10	2.6	42.4
15	3.0	35.7
20	3.4	52.6
25	3.7	57.3
30	4.1	70.5
35	4.4	91.4
40	4.8	120.0

It is seen that the SVD times are worse by factors ranging from 16 for `ncr` = 10 to 25 for `ncr` = 40. Regarding storage, the SVD requires $(n+m)k$ floating-point words, whereas the SQR requires only k^2 words.

In the above example the singular values of the test matrix had no gaps, and consequently the reduced-rank approximations are not very good—either for the SVD or the SPQR approximations. In a different experiment, we generated singular values by the statements

```
s = logspace(0, -4, n);
s(20:n) = 1e-6*s(20:n);
```

This places a multiplicative gap of about 10^{-6} between the 19th and 20th singular values. For `nc` = 19, 20 we timed the call

```
spqr(A, 1.e-2, nc, 1)
```

and the above call to `svds`. The results were

nc	SPQR	SVD
19	1.8	4.4
20	1.8	323.6

The improved performance for the SVD when `nc` = 19 is explained by the fact that `svds` is being asked to find a singular subspace whose singular values are well separated from the remaining singular values. Under such circumstances

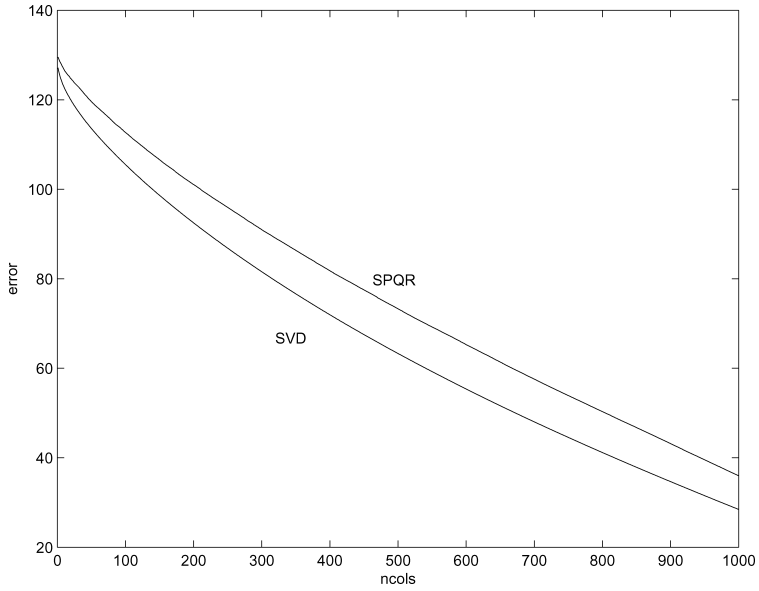

Fig. 1. Error in SVD and SPQR approximation for CRAN.

iterative methods for the SVD converge rapidly. The dismal performance of the SVD for `nc` $= 20$ is harder to explain. The function `svds` is being asked to find the 20th singular value, which is small compared with $\|A\|$ and is poorly separated from the other small singular values. Experience has shown this to be a difficult task. Be that as it may, the 20th singular value has to be computed to estimate the error in the approximation with `nc` $= 19$.

As a final example we consider a term-document matrix from an LSI application.[2] The matrix (or rather its transpose) is 4,612 by 1,398. Figure 1 graphs the error (Frobenius norm) in the SVD and SPQR approximations for `ncols` ranging from 1–1000. The SVD approximation is better, as it must be, but the SPQR approximation tracks it nicely. More specifically from $k = 200$ to $k = 1000$ the ratios of the error in SPQR to those in SVD vary almost linearly between 1.09 and 1.26. The slope of the least squares linear approximation is $2.1{\cdot}10^{-4}$, which implies that the discrepancy between the SVD and SPQR approximations grows very slowly with k.

The SVD approximations were not actually computed: that would have taken too long. Instead the norms of the SVD approximation were computed from the singular values of A, which were computed via the statement;

```
R = qr(A);
sig = svd(full(R));
```

The total time was about 4 minutes. By contrast the time to compute the entire SPQR approximation with 1000 columns (and hence all the decompositions with fewer columns) was about 2.5 minutes.

[2]Specifically the matrix CRAN generated from the Cranfield collection and available at `http://www.cs.utk.edu/~lsi/`.

```
   1   2   3  4
1 a           b
2     c
3 d
4         e
5 f           g
6         h

val        :  a d f c e h b g   (floating-point nnz)
col_start  :  1 4 5 7 9         (integer n+1)
rx         :  1 3 5 2 4 6 1 5   (integer nnz)
```

Fig. 2. Compressed column representation of a 6×4 matrix.

7. SPARSITY CONSIDERATIONS

The timings of the last section show that the MATLAB implementation of the SCR approximation is considerably faster than computing the SVD to obtain an equivalent approximation. The code is simple because MATLAB hides the implementation of the sparse matrix-vector multiplications that are at the heart of the algorithms. It is therefore natural to try to improve on the performance of the MATLAB implementation by writing the algorithm—sparse operations and all—in a compiled language like C or Fortran. This section is devoted to sparsity issues that must inform such an attempt.

For definiteness we will consider the problem of computing a pivoted SPQR approximation for a matrix A of order n where n is large. We will assume that the number of columns `nc` in the approximation is small compared to n. Finally, we will assume that A is represented in compressed column (CC) form, which we will now briefly describe. For definiteness, we will assume 1-based indexing and use MATLAB statements in the examples.

An example of CC representation is given in Figure 2. The nonzero values of the elements of the sparse matrix A are stored in column major order in an array `val`. The length of the array is `nnz`—the number of nonzero elements of A. An integer array `rx` of length `nnz` contains the row indices of the corresponding elements in `val`. Another integer array, `col_start`, of length $n + 1$ tells where the columns start in `val` and `rx`. Specifically, the first element in column `j` is `val[col_start[j]]`. The value of `col_start[n+1]` is set to `nnz+1`. This means that the length of column `j` is `col_start[j+1]-col_start[j]`.

Formation of the matrix-vector products Ax and $x^{\mathrm{T}}A$ are easy in this representation. The following code computes $y = Ax$

```
y(1:n) = 0;
for j=1:n
   for ii = col_start(j):col_start(j+1)-1;
      i = rx(ii);                                    (19)
      y(i) = y(i) + val(ii)*x(j);
   end
end
```

Similarly, we can compute $y = x^{\mathrm{T}}A$ as follows.

```
for j=1:n
   y(j) = 0;
   for ii = col_start(j):col_start(j+1)-1;
      i = rx(ii);                                    (20)
      y(j)  = y(j) + x(i)*val(ii);
   end
end
```

Both algorithms require `nnz` additions and multiplications. Both traverse the array `val` in its natural order, which makes for good cache usage. The first traverses `x` in its natural order, but its references to `y` jump around; the reverse is true for the second algorithm.

Now if we examine the quasi-Gram–Schmidt algorithm, we find we must perform the following operations involving the matrix A.

(1) Extract the pivot column from `A`.
(2) Calculate matrix-vector products of the form `x'*A(:,colx(1:k-1))`.
(3) Calculate matrix-vector products of the form `A(:,colx(:,1:k-1))*x`.
(4) Calculate matrix-vector products of the form `x'*A(:,colx(:,k+1:n))`. (As we have mentioned, when n is large, this calculation is the most expensive part of the algorithm.)

CC storage is ideal for performing all these operations. For example, to calculate `A(:,colx(1:k-1))*x`, we need only replace the outer `for` statement in (19) by

```
for j=colx(1:k-1)
```

Again, to compute `x'*A(:,colx(:,k+1:n))` we change the outer `for` statement in (20) with

```
for j=colx(k+1:n)
```

The algorithms no longer access `val` sequentially, but access within an individual column is concentrated in the contiguous part of `val` where its elements lie. Thus the CC representation goes hand-in-glove with the computation of the SPQR approximation.

The situation is different when we must compute the SPQR approximation of A^{T}, as is required when we wish to compute an SCR approximation. There are two major alternatives. We can work with A^{T}, or we can write a row-oriented version of `spqr` that works directly with A.

Regarding the first alternative, the SPARSKIT package by Saad [1994] gives an algorithm for transposing a matrix in condensed format in place.[3] The algorithm requires an additional working integer array of size `nnz` and `O(nnz)` operations. A disadvantage is that the elements of each column, though they remain contiguous in `val`, no longer occur in their natural order. This makes no difference for our algorithms for forming matrix-vector products.

[3]The algorithm assumes condensed row format, but it can easily be adapted to CC format.

The second alternative is to write a row-oriented version of spqr to compute a decomposition of the form

$$\hat{P}^{\mathrm{T}} A = \hat{R}^{\mathrm{T}} \hat{Q}^{\mathrm{T}},$$

where $\hat{P}$ is a permutation, $\hat{R}$ is upper trapezoidal, and $\hat{Q}$ is orthogonal. The algorithm is completely analogous to the SPQR; however, it pivots on rows of A producing a pivot array rowx corresponding to colx in SPQR. The operations we must perform are essentially the transposes of the operations listed above for SPQR.

(1) Extract the pivot row from A.
(2) Calculate matrix-vector products of the form A(rowx(1:k-1),:)*x.
(3) Calculate matrix-vector products of the form x'*A(rowx(1:k-1),:).
(4) Calculate matrix-vector products of the form A(rowx(k+1:n,:))*x.

The natural way to implement the row-oriented algorithm is to transform A into compressed row format. Once again, SPARSKIT provides an algorithm. The advantage of this approach is that the translation from spqr to the row-oriented version is purely mechanical. The disadvantage is that the storage requirements are doubled.

An alternative is to work with the CC format, perhaps augmented by additional arrays. However, this creates difficulties in implementing the row-oriented algorithm. Specifically, consider the adaptation of (19) to compute A(rowx(1:k-1),:)*x.

```
y(1:n) = 0;
for j=1:n
   for ii = col_start(j):col_start(j+1)-1;
      i = rx(ii);
      if i in rowx(1:k-1)                                    (21)
         y(i) = y(i) + val(ii)*x(j);
      end
   end
end
```

There are two problems with this algorithm—one easily solved, the other more difficult.

The first problem is that with each iteration of the inner loop rowx(1:k-1) must be searched to determine if it contains i as an entry. The cure is to negate the indices of rx corresponding to row i when row i is brought into the factorization. Then we may replace the the conditional part of the inner loop with

```
      if rx(i) < 0
         y(-i) = y(-i) + val(ii)*x(j);
      end
```

The second problem is that by our assumptions k $\ll$ n. Now the loops in (21) traverse all the nnz elements in the matrix A. But we actually work with only those few elements in the rows indexed by rowx(1:k-1). In other words, for most of the time, the body of the double loop does nothing. The cure for

```
     1   2   3  4
1 a          b
2     c
3 d
4         e
5 f           g
6         h


val        :  a d f c e h b g   (floating-point nnz)
col_start  :  1 4 5 7 9         (integer n)
row_index  :  1 3 5 2 4 6 1 5   (integer nnz)
row_start  :  1 3 4 5 6 8 9     (integer m+1)
row_elp    :  1 7 4 2 5 3 8 6   (integer nnz) (elp = element pointer)
col_index  :  1 1 1 2 3 3 4 4   (integer nnz)
```

Fig. 3. Compressed-column representation of a 6×4 matrix with row links.

this problem is to store a copy of the matrix `A(rowx(1:k-1)` in compressed-row format. Because `k` $\ll$ `n`, the extra storage is insignificant. Moreover, it is then easy to perform the operations `x'*A(rowx(1:k-1),:)` and `A(rowx(1:k-1),:)*x`. We choose the compressed row-form because it is easy to add additional rows to it as `k` increases.

Now consider the product `A(rowx(k+1:n,:))*x`. Assuming that we have negated the elements of `rx` corresponding to rows `rowx(1:k)`, we can perform this multiplication by modifying the body of the loop (21) as follows.

```
if rx(i) > 0
   y(i) = y(i) + val(ii)*x(j);
end
```

Since `k` $\ll$ `n`, the body of the loop is performing useful work most of the time.

Surprisingly, the problem of extracting the pivot row from a compressed column form is also difficult. For definiteness, let the index of that row be `ipvt`. The following algorithm does the job.

```
for j=1:n
   for ii=col_start(j):col_start(j+1)-1
      if rx(ii) > ipvt, break, end
      if rx(ii) == ipvt
         % A(ipvt, j) = val(ii) is in row ipvt;
         break;
      end
   end
end
```

Unfortunately, if `ipvt` = `n`, we must traverse the entire matrix just to extract the pivot row. Thus the use of this algorithm has the potential to add $O($`nzz`$)$ work at each step of the algorithm.

A solution is to augment the compressed column format to allow access to the rows. Figure 3 shows one such scheme, which we will call compressed-column, linked-row representation (CCLR representation). With it we can access the row `ipvt` as follows.

```
for jjj = row_start(ipvt):row_start(ipvt+1)-1
   jj = row_elp(jjj);
   j = cx(jj);
   % A(ipvt, j) = val(jj) is in row ipvt
end
```

This ability to traverse rows allows one to implement the row-oriented algorithm in exactly the same manner as the column oriented algorithm. However, there are two differences that may affect efficiency. First there are two levels of indirection from `jjj` to `jj` to `j`. Second, row traversals do not access the elements of `val` sequentially. Thus it may still pay to maintain a copy of `A(rowx(1:k-1),:)` and to compute `A(rowx(1:k-1),:)*x` directly from the column oriented form.

To sum up, if we assume that we have 4-byte integers and 8-byte floating-point words, then compressed column storage requires $12\,\mathtt{nnz} + 4\,\mathtt{n}$ bytes of memory. To compute the SPQR approximation of A^{T} we have the following options.

(1) Transpose A in place. Storage: $16\,\mathtt{nnz} + 4\,\mathtt{n}$ ($4\,\mathtt{nnz}$ of which is temporary and can be allocated as an automatic variable). Additional work: $O(\mathtt{nnz})$ for the initial transpose.
(2) Copy A to compressed row format. Storage: $24\,\mathtt{nnz} + 8\,\mathtt{n}$. Additional work: $O(\mathtt{nnz})$ for the conversion.
(3) Use CC representation, and copy `A(:,colx(1:k-1))`. Storage $12\,\mathtt{nnz} + 4\,\mathtt{n}$. Additional work: up to $O(\mathtt{nnz})$ per step to extract rows.
(4) Use CCLR representation, copy `A(:,colx(1:k-1))`, and use the row links only to extract the pivot row. Storage: $20\,\mathtt{nnz} + 8\,\mathtt{n}$. Additional work $O(1)$ per step.
(5) Use CCLR representation: $20\,\mathtt{nnz} + 8\,\mathtt{n}$. Additional work: $O(\mathtt{nnz})$ per step from extra overhead in processing rows.

Items 1, 2, and 4 emerge as the strongest options, playing off storage, work, and ease of programming against each other. Item 1 is attractive because of its low storage requirements and the fact that one does not have to code a row-oriented version of `spqr`. Item 2 doubles the storage, but makes the coding of the row-oriented version trivial. Item 4 almost doubles the storage, and the copying complicates the row-orient algorithm. But it is attractive when additional row operations involving A are anticipated.

It should be stressed that the above analysis was done under a number of special hypotheses—e.g., $\mathtt{nc} \ll \mathtt{n}$. Change the hypotheses and the the results may change. Moreover, the nature of the problem may make other storage schemes preferable. However, the analysis illustrates the questions that should be asked by someone implementing the MATLAB algorithms in a language where sparseness must be explicitly taken into account.

REFERENCES

BERRY, M. AND BROWNE, M. 1999. *Understanding Search Engines: Mathematical Modeling and Text Retrieval*. SIAM, Philadelphia, PA.

BERRY, M. W., DRMAČ, Z., AND JESSUP, E. 1999. Matrices, vector spaces, and information retrieval. *SIAM Rev. 41*, 335–362.

BERRY, M. W., DUMAIS, S. T., AND O'BRIEN, G. W. 1995. Using linear algebra for intelligent information retrieval. *SIAM Rev. 37*, 573–595.

BERRY, M. W. AND MARTIN, D. I. 2004. Principal component analysis for information retrieval. In *Handbook of Parallel Computing and Statistics*. Marcel Dekker, New York. To appear.

BJÖRCK, Å. 1996. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia.

DEERWESTER, S., DUMAIS, S., FURNAS, G., LANDAUER, T., AND HARSHMAN, R. 1990. Indexing by latent semantic analysis. *J. Amer. Soc. Info. Sci. 41*, 391–407.

JIANG, P. AND BERRY, M. W. 2000. Solving total least squares problems in information retrieval. *Lin. Alg. Appl. 316*, 137–156.

SAAD, Y. 1994. SPARSEKIT: A basic tool kit for sparse matrix computations. Available at `www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

STEWART, G. W. 1980. The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM J. Num. Anal. 17*, 403–409.

STEWART, G. W. 1998. *Matrix Algorithms I: Basic Decompositions*. SIAM, Philadelphia.

STEWART, G. W. 1999. Four algorithms for the the efficient computation of truncated pivoted qr approximations to a sparse matrix. *Numerische Mathematik 83*, 313–323.

STEWART, G. W. 2001. *Matrix Algorithms II: Eigensystems*. SIAM, Philadelphia.

STEWART, G. W. 2004. Error analysis of the quasi-Gram–Schmidt algorithm. Tech. Rep. CMSC TR-4572, Department of Computer Science, University of Maryland.

STUART, G. W. AND BERRY, M. W. 2003. A comprehensive whole genome bacterial phylogeny using correlated peptide motifs defined in a high dimensional vector space. *J. Bioinformatics and Computational Bio. 1*, 475–493.

ZHANG, Z., ZHA, H., AND SIMON, H. 2002. Low-rank approximations with sparse factors I: Basic algorithms and error analysis. *SIAM J. Matrix Anal. Appl. 23*, 706–727.

Received June 2004; revised December 2004; accepted January 2005