SIAM J. SCI. COMPUT.
Vol. 31, No. 1, pp. 761–775

BLOCK GRAM–SCHMIDT ORTHOGONALIZATION*

G. W. STEWART†

Abstract. The classical Gram–Schmidt algorithm for computing the QR factorization of a matrix X requires at least one pass over the current orthogonalized matrix Q as each column of X is added to the factorization. When Q becomes so large that it must be maintained on a backing store, each pass involves the costly transfer of data from the backing store to main memory. However, if one orthogonalizes the columns of X in blocks of m columns, the number of passes is reduced by a factor of $1/m$. Moreover, matrix-vector products are converted into matrix-matrix products, allowing level-3 BLAS cache performance. In this paper we derive such a block algorithm and give some experimental results that suggest it can be quite effective for large scale problems, even when the matrix X is rank degenerate.

Key words. orthogonalization, Gram–Schmidt algorithm, blocked algorithm, QR factorization

AMS subject classifications. 65F25, 65G50, 65Y20

DOI. 10.1137/070682563

1. Background.

The purpose of this paper is to describe a block Gram–Schmidt algorithm for computing a QR factorization of a matrix X. The algorithm is derived on heuristic principles and is justified by the fact that it appears to work. At its heart is the basic Gram–Schmidt step for orthogonalizing a vector x against the columns of an orthonormal matrix Q. In its naive formulation the basic Gram–Schmidt step can fail, and it is a tricky matter to produce a version that works. Accordingly, we will begin with the basic Gram–Schmidt algorithm and only later (section 3) turn to the block variant.

The classical Gram–Schmidt step proceeds as follows. Given a matrix Q with orthonormal columns and a vector x, set

$$\hat{y} = (I - QQ^*)x = x - Q(Q^*x) \equiv x - Qr. \tag{1.1}$$

It is easily seen that $Q^*\hat{y} = 0$, so that $\hat{y}$ is orthogonal to the columns of Q. If, in addition, $\hat{y} \neq 0$, then $\rho = \|\hat{y}\| \neq 0$ and $y = \hat{y}/\rho$ is well defined. The matrix $(Q\ y)$ is then an orthonormal matrix that expands the column space of Q to contain x. Moreover, we have the important relation

$$x = (Q\ y)\begin{pmatrix} r \\ \rho \end{pmatrix}. \tag{1.2}$$

In fact, the Gram–Schmidt step may be derived directly from this relation and the requirement that $(Q\ y)$ be orthonormal.

The Gram–Schmidt step may be used to orthogonalize the columns of an $n \times p$ matrix X. Let the MATLAB function

```
function [y, r, rho] = gsstep(Q, x)
```

*Received by the editors February 13, 2007; accepted for publication (in revised form) July 7, 2008; published electronically November 12, 2008. This work was supported in part by the National Science Foundation under grant CCR0204084.
http://www.siam.org/journals/sisc/31-1/68256.html

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 (stewart@cs.umd.edu).

implement the Gram–Schmidt step (including an initial step where Q has zero columns). Then the following MATLAB loop computes the matrix Q containing the orthogonalized vectors:

```
(1.3)
1.  [n,p] = size(X);
2.  R = [];
3.  Q = zeros(n,0);
4.  for j = 1:p
5.    [y, r, rho] = gsstep(Q, x(:,j));
6.    R = [R r; zeros(1,j-1) rho];
7.    Q = [Q y];
8.  end
```

If all goes well, at the end of the loop we have an orthonormal Q and an upper triangular R satisfying

$$X = QR.$$

This is the QR factorization of X.

The well-known deficiencies of this approach stem from a combination of two facts. First, the computations must be carried out in finite-precision floating-point arithmetic. Second, the QR factorization is not unique unless X is of full column rank p. The combination implies that when X is near a rank-deficient matrix its QR factorization will be computed inaccurately. Thus we must first ask what it means to compute a QR factorization.

In this paper we will seek a factorization satisfying the following conditions:

$$\begin{aligned} &1. \quad \|I - Q^*Q\| \le C_Q \epsilon_{\mathrm{M}}, \\ &2. \quad \|X - QR\| \le C_X \epsilon_{\mathrm{M}}. \end{aligned} \tag{1.4}$$

Here ϵ_{M} is the rounding unit (about $2.2 \cdot 10^{-16}$ in IEEE double precision), and C_Q and C_X are modest constants that depend on the dimensions of the problem and the properties of the floating-point arithmetic used in the computation. Such a decomposition always exists and can be computed using Householder triangularization, even when X is rank deficient. As we will see, coaxing the Gram–Schmidt algorithm to give a decomposition satisfying (1.4) is more difficult.[1]

In general, the second condition in (1.4) is maintained naturally by the Gram–Schmidt step. However, the orthogonality of Q can deteriorate as algorithm (1.3) progresses. The loss of orthogonality is signaled by a decrease in norm owing to cancellation while passing from x to $\hat{y}$ in (1.1). The standard cure is to reorthogonalize y. A classic result [7, section 6.9] says that under most circumstances this reorthogonalization will suffice to produce a q that is orthogonal to Q to working accuracy. However, in extreme cases it is possible for cancellation to occur in the reorthogonalization, so that the new y lacks orthogonality or is even exactly zero. Thus a complete Gram–Schmidt procedure must take into account the appearance of a zero or unnaturally small y in the course of the computation.

A second deficiency, which arises in large-scale problems, is the computation of the matrix-vector products Q^*x and Qr in (1.1). The problem has two aspects.

[1] It has been pointed out that Q has full rank when X is rank degenerate and therefore does not provide a basis for the column space of A. However, if we determine a basis U for the column space of R, then QU is the required basis for X.

First, Q may become so large that it is necessary to maintain it on a backing store. Thus the time to compute r in (1.1) is increased by the time required to bring Q into main memory. A possible solution is to orthogonalize a block of vectors, say $X(:, j : j + m - 1)$, simultaneously, so that the transfer time for Q is prorated over the m columns of the block. The design of such a block algorithm is the ultimate goal of this paper.

The block algorithm also answers the second aspect of the problem. Namely, even when Q can be maintained in main memory, the replacement of the m matrix-vector products Q^*x_k $(k = j, \ldots, j+m-1)$ by the computation of $Q^*X(:, j : j+m-1)$ gives more scope for the efficient management of cache memory. Given a set of properly tuned basic linear algebra subprograms (BLAS), we can expect a blocked algorithm to run faster than the unblocked version.

There are many analyses of the Gram–Schmidt algorithm, of which the majority focus on the modified Gram–Schmidt algorithm, as opposed to the classical Gram–Schmidt algorithm used in this paper. Björck [1] gives a summary of this literature and an excellent bibliography (the interesting paper [2] postdates Björck's book). It is worth noting that most analyses assume an upper bound on the condition of X. Thus they do not apply to our problem, where X is permitted to be rank degenerate and even have repeated or zero columns.

There is a small body of literature on block Gram–Schmidt algorithms, mostly from the area of parallel computing [4], [5], [6], [10], [11]. The algorithms in these papers block Q so that it can be distributed over a system of processors, whereas our approach leaves Q intact and blocks the unprocessed part of X to make better use of level three BLAS. The papers [8], [9] are closer to ours inasmuch as they do some form of blocking to enhance cache performance. But none of the algorithms in these papers will produce a decomposition satisfying (1.4).

The next section of this paper is devoted to developing a Gram–Schmidt step that allows for the fact that reorthogonalization may fail. The solution involves the introduction of random vectors, and we will argue that such vectors are particularly well suited to this application. In section 3, we will develop the block Gram–Schmidt step. The key feature here is that failures, if they are not too frequent, can be handled by the strategic use of unblocked Gram–Schmidt steps. Moreover, a strategy called aggressive replacement can reduce the number of unblocked Gram–Schmidt steps, but at the cost of compromising the second of the conditions (1.4). In section 4 we give the results of numerical experiments.

2. The Gram–Schmidt step. In this section we will describe an algorithm for the basic Gram–Schmidt step. Its original form may be summarized as follows:

(2.1)
```
1.  function [y, r , rho] = gsstep(Q, x)
2.    r = Q'*x;
3.    y = x - Q*r;
4.    rho = norm(y);
5.    y = y/rho;
6.  return
```

We will now give a simple analysis of this algorithm that shows both its weak and its strong points. It is based on the assumption that Q is exactly orthonormal.[2]

[2] Accounting for the fact that Q may deviate from orthonormality is one of the complications of the error analyses cited in section 1. Remarkably, if the Gram–Schmidt step is iterated, this source of error washes out—see the appendix.

We begin by writing

$$x = x_Q + x_\perp,$$

where x_Q denotes the orthogonal projection of x onto the columns of Q and $x_\perp$ denotes the complementary projection. Along with this decomposition of x we will define the quantities

$$\sigma = \frac{\|x_Q\|}{\|x\|} \quad \text{and} \quad \gamma = \frac{\|x_\perp\|}{\|x\|}, \tag{2.2}$$

which are, respectively, the sine and cosine of the angle between x and $x_\perp$. The object of (2.1) is therefore to reduce the quantity σ.

We now suppose that (2.1) is executed in floating-point arithmetic with rounding unit ϵ_{M}. An elementary rounding-error analysis shows that the unormalized projected vector $\hat{y}$ satisfies

$$\hat{y} = x_\perp + e, \tag{2.3}$$

where

$$\frac{\|e\|}{\|x\|} \leq C\epsilon_{\text{M}} \equiv \epsilon. \tag{2.4}$$

Here, as in (1.4), C is a modest constant.

Since we are going to iterate the projection, let $\hat{\sigma}$ and $\hat{\gamma}$ be the sine and cosine defined for $\hat{y}$ in analogy with (2.2). Now, from (2.3), it follows that

$$\|\hat{y}\| \geq \|x_\perp\| - \|e\|$$

and

$$\|\hat{y}_Q\| \leq \|e\|.$$

Hence

$$\hat{\sigma} \leq \frac{\|e\|}{\|x_\perp\| - \|e\|} \leq \frac{\epsilon}{\gamma - \epsilon} = \frac{\epsilon}{\sqrt{1-\sigma^2} - \epsilon}. \tag{2.5}$$

The bound (2.5) shows how quickly the Gram–Schmidt step, when iterated, can approach orthogonality. Suppose, for example, that $\gamma = 3\epsilon$, so that x has almost no component along the orthogonal complement of the column space of Q. Then after one Gram–Schmidt step, we have $\hat{\sigma} \leq \frac{1}{2}$. Let $\check{y}$ be the result of projecting $\hat{y}$. Then ignoring second order terms in ϵ,

$$\check{\sigma} \lesssim \sqrt{\frac{4}{3}}\epsilon \leq 1.2\,\epsilon.$$

In other words, two Gram–Schmidt steps are sufficient to almost completely orthogonalize a vector that lies almost exactly in the column space of Q. For this reason, the Gram–Schmidt step with reorthogonalization is frequently called the twice-is-enough algorithm.[3]

[3]This appellation is frequently misunderstood. It was coined by W. Kahan in connection with orthogonalizing a Krylov sequence. What is actually meant is that either $\check{y}$ is almost fully orthogonal to the current Krylov subspace (represented by Q) or the norm of $\check{y}$ is so small that the Krylov subspace is effectively an invariant subspace of the matrix in question. See [7, section 6.9].

The analysis also provides an effective way to determine when the result of a Gram–Schmidt step is satisfactorily orthogonal. Suppose, for example, that

$$\|\hat{y}\| \geq \frac{1}{2}\|x\|. \tag{2.6}$$

Then $\|x_\perp\| + \|e\| \geq \frac{1}{2}\|x\|$, or $\gamma \geq \frac{1}{2} - \epsilon$. From (2.5) it follows that

$$\hat{\sigma} \lesssim 2\,\epsilon,$$

where we have again ignored second order terms in ϵ. In other words, if the projection step results in a reduction in norm that is not less than $1/2$, the resulting vector is almost fully orthogonal.

The above analysis shows why the Gram–Schmidt step with reorthogonalization is so effective. In almost all cases it will produce a fully orthogonal vector, and it provides a simple criterion to test the orthogonality of the result. But almost all is not the same as all. The analysis says nothing about vectors for which γ is 3ϵ or less. In such cases, the result $\hat{y}$ of the first projection may consist largely of rounding error (or may even be zero) and the second projection $\check{y}$ may fail to give a nonzero, fully orthogonal vector.

Note that this problem can occur only when $\gamma \leq 3\epsilon$, which implies that $\|\hat{y}\|/\|x\| \leq 4\epsilon$. Now another rounding-error analysis shows that

$$x = Qr + \hat{y} + f, \qquad \frac{\|f\|}{\|x\|} \leq \epsilon, \tag{2.7}$$

in which we may have to adjust the constant C upward in (2.4). This means that we can replace $\hat{y}$ by any vector of size $\|\hat{y}\|$ and the relation (2.7) continues to hold with perturbations of order ϵ. If this vector has a reasonable value of γ, then at most two subsequent orthogonalization steps will produce a fully orthogonal vector.

We are going to show that a good choice for the replacement is a random vector—or equivalently, that a random vector will have a good value of γ. Before proceeding, however, we must treat an argument that is tantamount to saying that we need to do nothing. Specifically, we know that if $\gamma \leq 3\epsilon$, the vector $\hat{y}$ consists primarily of rounding error. Now (the argument continues) rounding error is often successfully modeled as random error. Consequently, if a vector composed of random error can be expected to have a good value of γ, so can $\hat{y}$, and the next two projections will result in orthogonality.

Actually, there is a good bit of truth in this argument. With only rare exceptions twice is enough. However, the randomness of $\hat{y}$ is of low quality. The problem is that $\hat{y}$ will have been computed with extreme cancellation. Consequently, its components may have only one or two leading nonzero bits—the rest of the bits are zero. In such a restricted sample space, unhappy coincidences are likely to happen.

On the other hand, when the components of a replacement vector are independent random normal deviates, the situation is much different. To see this, suppose that Q is $n \times k$ and that $(Q\ Q_\perp)$ is orthogonal. Let x be a vector of normally distributed random variables with mean zero and variance one. Then

$$\gamma^2 = \frac{\|Q_\perp^* x\|^2}{\|Q_\perp^* x\|^2 + \|Q^* x\|^2}.$$

Now $Q_\perp^* x$ and $Q^* x$ are also independently normally distributed, and hence $\|Q_\perp^* x\|^2$ and $\|Q^* x\|^2$ are independent χ^2 random variables with $n-k$ and k degrees of freedom. It follows that γ^2 has a beta distribution with $(n-k)/2$ and $k/2$ degrees of freedom.

We can use the MATLAB function `betainc` to compute the probability that γ^2 is less than a prescribed amount. In particular, we will consider the probability that $\gamma^2 \leq 10^{-26}$ or $\gamma \leq 10^{-13}$. This is sufficiently greater than the IEEE double-precision rounding unit ($2.2 \cdot 10^{-16}$) to ensure that two orthogonalization steps are enough. It will be sufficient to consider the extreme case where $k = n - 1$, so that $\|Q_\perp^* x\|^2$ is, on the average, as small as possible and $\|Q^* x\|$ is as large as possible. The following is a table for various values of n of the probability P that γ will be less than 10^{-13}:

(2.8)

n	P
10^1	2.3e−13
10^2	7.9e−13
10^3	2.5e−12
10^4	8.0e−12
10^5	2.5e−11
10^6	8.0e−11

It is seen from the table that the probability of producing an unsatisfactory replacement vector of random normal deviates is extremely small.

Of course, things only get better as k decreases. The worst case in the examples we will treat in section 4 is $n = 10{,}000$ and $k = 500$. In this case the probability of γ being less than 0.9 is about 10^{-186}. In other words, the probability of needing more that one Gram–Schmidt step is astronomically small.

To the degree that $\|\hat{y}\|/\|x\|$ is greater than ϵ, random replacement will compromise the second condition in (1.4). Nonetheless, as we shall see in section 4, by being a little aggressive with random replacement, we can, in some cases, considerably improve the performance of our block algorithm.

We are now in a position to assemble a working algorithm for the Gram–Schmidt step. The MATLAB function `gssro` (**G**ram–**S**chmidt **s**tep with **re**o**r**thogonalization) has the following calling sequence:

```
function [y, r, rho, northog] = gssro(Q, x, nu, rpltol)
```

The arguments are self-explanatory, with the exceptions of the optional arguments `nu`, `rpltol`, and `northog`. The optional argument `nu` ordinarily contains the norm of the vector `x` to be orthogonalized, which is its default value if `nu` is absent. However, in our block algorithm, `x` will have undergone previous projections, and in this case `nu` is used to communicate the original norm of `x`. The number `rpltol` is used in determining whether to introduce a random vector, as described above. The output argument `northog`, if present, contains the number of projections performed by `gssro`.

The function has some initial boilerplate to set default values and to handle zero `x` and empty `Q`, along with some cleanup code at the end. This part also normalizes x before it is orthogonalized and then adjusts the output at the end to undo the normalization. The reason for this will be explained a little later.

The heart of the function is the orthogonalization loop in Figure 2.1. At this point the vector to be orthogonalized is `y` and has norm one. The projection step is done in lines 10–12. Note that `r` is updated in line 11. Except for the first time through the loop, the correction will be very small, probably negligible, but there is no point in not making it.

The remainder of the loop pits three norms against two tests. The norm `nu` is the size of the original `x` (this definition will be qualified later). The norm `nu1` is the norm of `y` before projection, and `nu2` is the norm after projection. The first test

```
 7.  nu1 = nu;
 8.  while true
 9.     if nargout == 4, northog = northog + 1; end
10.     s = Q'*y;
11.     r = r + s;
12.     y = y - Q*s;
13.     nu2 = norm(y);
14.     if nu2 > 0.5*nu1, break, end;
15.     if nu2 > rpltol*nu*eps
16.        nu1 = nu2;
17.     else
18.        nu = eps*nu
19.        nu1 = nu;
20.        y = rand(n,1) - 0.5;
21.        y = nu*y/norm(y);
22.     end
23.  end
```

FIG. 2.1. *Orthogonalization loop in* `gssro`.

(line 14) compares `nu1` and `nu2` and exits the loop if the orthogonality criterion (2.6) is satisfied. The second test (line 15) asks if the norm `nu` of `x` has shrunk to the level of `rpltol` times the rounding unit. If it has not, the orthogonalization continues. If it has, a small random replacement vector is generated. The replacement is aggressive in proportion as `rpltol` is greater than one, and the second condition in (1.4) will suffer accordingly. Note that this replacement counts as a new `y`, and `nu` is adjusted accordingly.

The vector `y` returned by `gssro` will be orthogonal to `q` to working accuracy. There is a theoretical possibility that the algorithm could loop indefinitely, but by (2.8) the probability of this happening is vanishingly small. This probability is made even smaller because the initial norm of `y` is one. In IEEE double precision this means that we can make at least 17 replacements before `y` underflows, which would be a fatal error. Thus the probability of a fatal underflow is less than (vanishingly small)^{17}.

3. The block Gram–Schmidt step. The block algorithm is a natural generalization of the Gram–Schmidt algorithm. As we noted in the introduction, there are two reasons for seeking such an algorithm. The first is to minimize the number of passes over the previously orthogonalized vectors by applying them to several unorthogonalized vectors simultaneously. The second is to improve cache performance by replacing matrix-vector multiplications by matrix-matrix multiplications.

The overall algorithm goes as follows. The matrix X to be orthogonalized is partitioned by columns into blocks, and each block is successively orthogonalized and incorporated into Q. Assume that X has p columns and that the block size is m. Further suppose we have an equivalent `bgssro` of `gssro` to perform a block Gram–Schmidt step. Specifically, the algorithm takes a block X_{block} and produces Y, R_{12}, and an upper triangular matrix R_{22}, all satisfying

$$X_{\text{block}} = (Q\ Y)\begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix}. \tag{3.1}$$

Then the algorithm for orthogonalizing X goes as follows:

```
Q = zeros(n,0);
R = zeros(p);
for jl=1:m:p
   ju = min([p, jl+m-1]);
   [Y, R12, R22] = bgssro(Q, X(:, jl:ju));
   Q = [Q, Y];
   R(1:jl-1, jl:ju) = R12;
   R(jl:ju, jl:ju) = R22;
end
```

Thus our problem becomes that of constructing the function `bgssro` to perform a block Gram–Schmidt step.

To this end, suppose we are given an orthonormal matrix Q and an $n \times m$ block X of vectors to orthogonalize. The block projection is analogous to (1.1):

```
R12 = Q'*X;
Y = X - Q*R12;
```
(3.2)

Mathematically, the columns of Y will be orthogonal to those of Q. However, they will not in general be orthogonal to each other. Thus we must orthogonalize them, for which we use `gssro` as illustrated below:

```
for k=1:m
   [yk,r,rho] = gssro(Y(:,1:k-1), Y(:,k), nu(k), rpltol);
   Y(:,k) = yk;
   R22(1:k-1,k) = r;
   R22(k,k) = rho;
end
```
(3.3)

If all goes well, the matrix $(Q\ Y)$ is orthonormal and (3.1) is satisfied.

Of course all does not go well. As in the ordinary Gram–Schmidt step, the columns of Y in (3.2) can fail to be orthogonal to Q. But there is a second problem not encountered in the unblocked case. Specifically, cancellation in the course of executing (3.3) can cause a further loss of orthogonality. To see how this can come about, suppose that in the orthogonalization of a column y of Y against the previous columns there is significant decrease in the norm of the projected y. Then the loss of significant digits in the final y will cause a corresponding loss of orthogonality with Q. Note that this phenomenon can occur even when all the columns of the original Y were exactly orthogonal to Q. If we try to correct the problem by projecting the new Y onto the orthogonal complement of Q. The columns of the resulting matrix may not be orthogonal to either Q or each other, and so on. The upshot is that there is no simple analysis, as there was in the Gram–Schmidt step, to support a twice-is-enough strategy. Nonetheless, twice does seem to be enough in many cases, and it turns out there is a reasonable fix when it fails.

Actually, the fix consists of two parts. During the first round of orthogonalization consisting of (3.2) and (3.3), we can note the norms of the unormalized projections of each column. If they are all greater than 0.5 times the original column, the results are orthogonal to working accuracy, and we can return.

If, however, one or more of the norms becomes less than 0.5 times the original column, then we must apply another round of orthogonalization. Once again we monitor the norms of the unormalized projections of y_k. Whenever we encounter a

norm that is less than 0.5—a situation we will call an *orthogonality fault*—we pause and use `gssro` to orthogonalize y_k against $(Q\ \hat{y}_1, \ldots, \hat{y}_{k-1})$. This ensures that y_k is fully orthogonal to its predecessors, at the cost of additional passes over Q.

Regarding the first round, it can be shown that if the condition number of the input matrix Y—that is, the ratio of its largest to smallest singular values—is less than 2, then there will be no need for a second round of orthogonalization. If the condition number is greater than 2, then it is likely (though not certain) that a second round will be required.

The second round takes exactly the same time as the first round if there are no orthogonality faults. Each orthogonality fault exacts a cost of one or two (or rarely more) additional passes over Q. To put this in perspective, suppose we are trying to orthogonalize 100 vectors with a block size of 10, and assume a second round is always required. Without orthogonality faults, the proposed algorithm would require 40 passes over Q. With one orthogonality fault per block, the count increases to between 60 and 80 passes. This should be compared with a minimum of 200 passes up to over 400 for the unblocked algorithm.

Unfortunately, the number of passes over Q does not measure how the algorithms will actually perform. If m is small compared with p, the two rounds of the blocked algorithm do twice the work as the unblocked algorithm, provided the latter performs no reorthogonalizations. If, on the other hand, the latter always performs one reorthogonalization for each column of X, then the two algorithms do essentially the same amount of work.

To complete the description of our block algorithm, we must determine how the two rounds of orthogonalizations hang together. To do this, note that, as in (1.2), the first cycle of the orthogonalization produces a matrix Y satisfying

$$X = QR_{12} + YR_{22}, \tag{3.4}$$

where R_{22} is upper triangular. The second cycle produces a matrix Z with (hopefully) orthonormal columns satisfying

$$Y = QS_{12} + ZS_{22}. \tag{3.5}$$

Combining (3.4) and (3.5), we get

$$X = Q(R_{12} + S_{12}R_{22}) + ZS_{22}R_{22}.$$

Unlike the algorithm `gssro`, which is rather convoluted, the algorithm sketched above is relatively straightforward. Its calling sequence is

```
function [Y, R12, R22] = bgssro(Q, X, rpltol)
```

The input arguments `Q` and `X` are self-explanatory. On return, the orthonormal matrix Y satisfies

```
X = Q*R12 + Y*R22.
```

The optional parameter `rpltol`, whose default value is `1`, is passed directly to `gssro`. We will see later how its use can reduce orthogonalization faults.

Assume that `X` has `m` columns. Figure 3.1 exhibits an outline of the heart of `bgssro`. We have assumed that `Q` is not void. The only tricky part is the updating of `S12` and `s` in line 17. The problem is that `s`, which is of dimension `mq+k-1`, must be split between the part `s(1:mq)` to be added to `S12(:,k)` and the remainder `s(mq+1:mq+k-1)` which will be incorporated into `S22`.

```
     First orthogonalization step.
1.   R12 = Q'*X;
2.   Y = X - Q*R12;
3.   R22 = zeros(m);
4.   for k=1:m
5.      Orthogonalize Y(:,k) against Y(:,1:k-1);
6.      Update R22;
7.   end
8.   If the reduction in norms of all the projected vectors
9.   is less than 0.5, return;

     Second orthogonalization step.
10.  S12 = Q'*Y;
11.  Y = Y - Q*S12;
12.  S22 = zeros(m);
13.  for k=1:m
14.     [yk,s,sig] = gssro(Y(:,1:k-1), Y(:,k), 1, rpltol);
15.     if sig < 0.5

           Orthogonalization fault
16.        [yk,s,sig] = gssro([Q, Y(:,1:k-1)], Y(:,k));
17.        Update S12 and truncate s;
18.     end
19.     Update S22;
20.  end

     Combine the orthogonalization steps.
21.  R12 = R12 + S12*R22;
22.  R22 = S22*R22;
```

FIG. 3.1. *Outline of* `bgssro`.

4. Experiments. We now turn to an empirical evaluation of the block Gram–Schmidt algorithm. There are two questions to be answered. First, does the algorithm perform as intended? By this we mean does it produce a decomposition satisfying (1.4) and does it do so without an undue number of orthogonalization faults? Second, is it faster than the conventional Gram–Schmidt algorithm with reorthogonalization? The answer to this question is not obvious because, as we have observed, the block Gram–Schmidt algorithm may always require two passes over Q, each of which is potentially as expensive as the conventional algorithm.

We begin with the first question. Our experiments are based on an $n \times p$ matrix of the form

$$X = U\Sigma V^*,$$

where U and V are random orthonormal matrices and Σ is a diagonal matrix of singular values decreasing geometrically from 1 to 10^{-t}. In the experiments, we take $n = 10{,}000$ and $p = 500$. To introduce additional spin, we set the 25th column of X equal to the first and the 35th column to zero. We orthogonalize X using `bgssro` with a block size $m = 20$, which is approximately optimal for our experiments.

In the experiments we report the following statistics:

1. qrsd: $\|I - Q^*Q\|$.
2. xrsd: $\|X - QR\|$.
3. qpass: the number of passes over Q, each pass scaled by the number of columns in Q.
4. orthstp: the total number of orthogonalization steps performed in `gssro`.
5. faults: the total number of orthogonalization faults.
6. fpass: the total scaled number of passes over Q due to faults.

Items 1 and 2 answer to the residuals in (1.4). Item 3 measures how well the algorithm reduces the number of passes over Q. Note that passes are not all equal because the size of Q increases as the orthogonalization proceeds. Accordingly, each pass adds the number of columns in the current Q to `qpass`. Since with the small block size, the projection steps account for most of the work in the algorithm, item 3 is a rough indication of the relative time the algorithm takes. Item 4 monitors how much work `gssro` is doing in orthogonalizing the blocks. Item 5 records the number of orthogonalization faults, and item 6 shows how much of `qpass` is due to faults.

For the first test $t = 10$, so that the matrix X is rather ill-conditioned. The results of five runs are listed below:

(4.1)

qrsd	xrsd	qpass	orthstp	faults	fpass
$1.9 \cdot 10^{-14}$	$2.0 \cdot 10^{-16}$	24020	1382	1	20
$1.6 \cdot 10^{-14}$	$2.1 \cdot 10^{-16}$	24020	1383	1	20
$1.3 \cdot 10^{-14}$	$2.0 \cdot 10^{-16}$	24020	1382	1	20
$1.8 \cdot 10^{-14}$	$1.8 \cdot 10^{-16}$	24020	1382	1	20
$1.6 \cdot 10^{-14}$	$2.0 \cdot 10^{-16}$	24020	1382	1	20

In spite of the ill-conditioning of X, the algorithm behaves well. There is some loss of orthogonality, and the residual norms are about as small as can be expected. There was only one orthogonalization fault, corresponding to the repetition of x_1 in the 25th column. This nice behavior persists as t increases to 15, so that the smallest singular value of X (excluding the repetition and the zero column) is very near the rounding unit.

When we take $t = 20$ we get quite different behavior:

(4.2)

qrsd	xrsd	qpass	orthstp	faults	fpass
$2.8 \cdot 10^{-13}$	$1.8 \cdot 10^{-16}$	52800	1451	59	28800
$3.3 \cdot 10^{-13}$	$1.6 \cdot 10^{-16}$	47420	1441	49	23420
$2.2 \cdot 10^{-13}$	$1.6 \cdot 10^{-16}$	49760	1443	55	25760
$1.3 \cdot 10^{-12}$	$1.9 \cdot 10^{-16}$	58740	1455	67	34740
$2.0 \cdot 10^{-12}$	$1.8 \cdot 10^{-16}$	60860	1459	69	36860

Each run has between 59 and 69 orthogonalization faults. The orthogonality has deteriorated, but not unacceptably so. More importantly, the number of scaled projections has almost doubled—the orthogonalization faults have a marked effect on the efficiency of the algorithm. The reason is that the orthogonalization faults all occur in blocks 20 through 25, where Q is largest and projections are most expensive to compute.

There is, however, a fix—namely, aggressive replacement. By setting `rpltol` to 100 we replace any vector whose norm has been reduced by a factor greater than about 10^{14} by a random vector (see line 15 in Figure 2.1). This destroys information contained in the last two digits of the vector and must therefore have a deleterious effect on `xrsd`. However, the following results show that it reduces the number of

orthogonalization faults:

(4.3)

qrsd	xrsd	qpass	orthstp	faults	fpass
$8.9 \cdot 10^{-13}$	$8.0 \cdot 10^{-15}$	27140	1394	6	3140
$7.0 \cdot 10^{-13}$	$6.3 \cdot 10^{-15}$	27580	1396	8	3580
$9.3 \cdot 10^{-14}$	$6.1 \cdot 10^{-15}$	24940	1392	2	940
$5.1 \cdot 10^{-13}$	$7.1 \cdot 10^{-15}$	24940	1393	2	940
$4.6 \cdot 10^{-13}$	$6.6 \cdot 10^{-15}$	26280	1395	4	2280

The orthogonality remains about the same; and, as predicted, the computed QR decomposition no longer reproduces X to working accuracy. This loss, however, is slight, while the reduction in orthogonalization faults is dramatic. In fact the performance of the algorithm is comparable to the case $t = 10$ in (4.1).

In a more extreme example, with $t = 10$, the last $p/2$ singular values were set to zero. The result is the following:

qrsd	xrsd	qpass	orthstp	faults	fpass
$7.5 \cdot 10^{-14}$	$2.1 \cdot 10^{-16}$	108800	1592	224	84800

There were 224 orthogonalization faults, which accounts for 78 percent of qpass. The orthogonality is reasonable. When `rpltol` is set to 100, we get the following results:

qrsd	xrsd	qpass	orthstp	faults	fpass
$6.3 \cdot 10^{-14}$	$1.1 \cdot 10^{-14}$	28580	1384	13	4580

The loss of orthogonality remains essentially the same, but the number of orthogonalization faults is sharply reduced along with `qproj`, so that the latter are comparable with (4.1). The price we pay for this improvement is some deterioration in `xrsd`.

We now turn to the question of whether our block Gram–Schmidt algorithm results in a speedup. The algorithm was implemented in Fortran and run on a Pentium 4, 3GH computer with 2GB of memory. The programs were compiled on the Lakey/Fujitsu compiler and used the subroutine DGEMM from the Lahey BLAS. We applied both the Gram–Schmidt algorithm with reorthogonalization and our block Gram–Schmidt algorithm to a $10{,}000 \times 500$ matrix with pseudorandom elements in $[-0.5, 0.5]$. Note that the ratio of the smallest to the largest singular value is with high probability greater than 0.6.[4] We handicapped our algorithm by turning off the feature that stops at the first round of orthogonalization. This makes it a particularly tough test for the blocked algorithm, since the unblocked algorithm is unlikely to perform many reorthogonalizations, while the blocked algorithm is committed to performing two sets of block projections. The quantity `rpltol` was set to 1.

The time required to execute the unblocked algorithm was 7.4 sec. The times for the blocked algorithm with varying values of the block size m are presented below:

m	time
10	5.4
20	4.2
30	4.5
40	4.5
50	4.8

[4]This was determined empirically. As might be expected for a problem of this size, the ratio remained essentially the same over repeated trials.

The best speedup is about 1.8 for $m = 20$. All timings were for the same matrix. But when the matrix was varied by changing the seed of the random number generator, the timings remained essentially the same. They also were insensitive to the optimization level of the compiler, which indicates that the optimized BLAS DGEMM was responsible for the speedup. When the number of rows was increased to 30,000, the time for the unblocked algorithm was 22.0 sec and for the blocked algorithm 12.7 sec. These are approximately thrice the times for the original problem, and hence the speedup remained about 1.8. It should be noted that the speedup would have been close to twice those observed above had our algorithm been allowed to take its natural course and suppress the second round of orthogonalizations.

These preliminary results are encouraging, but more experience with real-life applications on a variety of machines will be required for a final assessment of the algorithm. An important drawback to the algorithm is that the block size m and `rpltol` have to be determined by trial and error. The block-size problem is common to all blocked algorithms. The `rpltol` problem will have to wait for a deeper understanding of why it works; but the algorithm's excellent performance on the very hard test cases given above suggests that a value of 100 will be adequate for most applications.

The same experiments were repeated for the MATLAB versions. The times in seconds were

unblocked	33.2
blocked	9.2

The reasons for the improvement are interesting. It is a commonplace that MATLAB is slow because it is an interpretive language. But some conversations with Cleve Moler inclined me to assign the problem to MATLAB's way of managing memory. For example, MATLAB calls its functions by value, so that a call to any function involving the current Q involves a memory allocation and a copying of Q. In the unblocked Gram–Schmidt algorithm the the work involved in passing Q is proportional to the work performed with Q. In addition, MATLAB performs a copy in updating assignment like

```
Q = [Q y].
```

Thus the algorithm should be sped up by reducing the number of passes over Q and the number of updatings.

This conjecture was tested by running the unblocked algorithm with a gutted `gssro` that simply returned zeros for `y`, `r`, and `rho`. The wall clock time was 25.3 sec. Thus about three quarters of the running time of the unblocked algorithm was due to array manipulation. This strongly suggests that blocking could be a useful MATLAB technique in other applications.

5. Appendix: Rounding error and lack of orthogonality. In this appendix we give the rounding error analysis leading to (2.3). Although it is fairly routine, it shows where the assumption of the exact orthonormality of Q is required and how deviations from orthonormality will affect the results.

The rounding error itself enters through the basic operations of matrix-vector multiplications and vector addition. Since these have been treated extensively in many places (e.g., [3]), we will simply list the results. In what follows the function fl(*expression*) will denote the result of evaluating *expression* in floating-point arithmetic:

$$
\begin{aligned}
r &= \mathrm{fl}(Q^*x) = Q^*x + e_1, & \|e_1\| &\le \gamma_1\|x\| \equiv C_1\|x\|\epsilon_{\mathrm{M}},\\
d &= \mathrm{fl}(Qr) = Qr + e_2, & \|e_2\| &\le \gamma_2\|r\| \equiv C_2\|x\|\epsilon_{\mathrm{M}},\\
\hat{y} &= \mathrm{fl}(x - d) = x - d + e_3, & \|e_3\| &\le \gamma_3(\|x\| + \|d\|) \equiv C_3(\|x\| + \|d\|)\epsilon_{\mathrm{M}},
\end{aligned}
$$

where, as usual, ϵ_{M} is the rounding unit, and the C_i are constants that depend on the operation and the dimensions of the problem. It should be noted that the only assumption about Q required to derive the above results is that $\|Q\| \le 1$. Deviations in this assumption can be absorbed without difficulty in the constants C_i.

If we combine the three equations above, we get

$$\hat{y} = x - QQ^*x - Qe_1 - e_2 + e_3 \equiv x_\perp + e.$$

Note that it is the orthonormality of Q that allows us to write $x_\perp = QQ^*x$.

We must now bound the error terms. We begin by bounding the norms of r and d. First,

$$\|r\| \le \|Q^*x\| + \|e_1\| \le \|x\| + \gamma_1\|x\| = (1 + \gamma_1)\|x\|.$$

Second,

$$\|d\| \le \|Qr\| + \gamma_2\|r\| \le (1 + \gamma_2)\|r\| \le (1 + \gamma_1)(1 + \gamma_2)\|x\|.$$

We now have

$$
\begin{aligned}
\|e\| &\le \|e_1\| + \|e_2\| + \|e_3\|\\
&\le \gamma_1\|x\| + \gamma_2\|r\| + \gamma_3(\|x\| + \|d\|)\\
&\le \gamma_1\|x\| + \gamma_2(1 + \gamma_1)\|x\| + \gamma_3[1 + (1 + \gamma_1)(1 + \gamma_2)]\|x\|\\
&= \{\gamma_1 + \gamma_2(1 + \gamma_1) + \gamma_3[1 + (1 + \gamma_1)(1 + \gamma_2)]\}\|x\|.
\end{aligned}
$$

Thus if we set

$$C = C_1 + C_2(1 + C_1\epsilon_{\mathrm{M}}) + C_3[1 + (1 + C_1\epsilon_{\mathrm{M}})(1 + C_2\epsilon_{\mathrm{M}})] \cong C_1 + C_2 + 2C_3,$$

we have

$$\|e\| \le C\|x\|\epsilon_{\mathrm{M}}.$$

It is worthwhile to analyze the effects of the failure of Q to be exactly orthonormal. Following condition 1 in (1.4) write $E = I - Q^*Q$ and assume that E is small but not insignificant. Then $Q(I - E)^{-\frac{1}{2}}$ has orthonormal columns, and

$$P_\perp = I - Q(I - E)^{-1}Q^* \cong I - Q(I + E)Q^*,$$

where we have used the approximation $(I - E)^{-1} = (I + E) + O(\|E\|^2)$. Hence with $x = x_Q + x_\perp$, we have

$$(I - QQ^*)x \cong x_\perp + QEQ^*x_Q,$$

so that the error in the projection is proportional to $\|E\|$. Since QEQ^*x_Q lies in the column space of Q, we have

$$(I - QQ^*)(I - QQ^*)x \cong x_\perp + QEQ^*QEQ^*x_Q,$$

and the error is now proportional to $\|E\|^2$. Otherwise put, each inexact Gram–Schmidt step reduces the error by a factor of $\|E\|$. Thus the effect of the error is to slow the convergence of the iterated Gram–Schmidt process. Eventually, however, the error is reduced to the point where rounding error alone controls the size of the error. For example, if $\|E\| < 10^{-8}$, this point is reached after two Gram–Schmidt steps are performed in double precision.

Acknowledgments. The author would like to thank the Mathematical and Computational Sciences Division of the National Institute of Standards and Technology for the use of their facilities during the development of this project. William Mitchell, of the same division, got me up to speed with the timings. Thanks also to Cleve Moler for a very useful conversation. The author is indebted to one of the referees for suggesting that `bgssro` check to see if a second round of orthogonalization is needed.

REFERENCES

[1] Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996.
[2] L. Giraud, J. Langou, M. Rozložník, and J. van den Eshof, *Rounding error of the classical Gram–Schmidt orthogonalization process*, Numer. Math., 101 (2004), pp. 87–100.
[3] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002.
[4] W. Jalby and B. Philippe, *Stability analysis and improvement of the block Gram–Schmidt algorithm*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 1058–1073.
[5] D. P. O'Leary and P. Whitman, *Parallel QR factorization by Householder and modified Gram–Schmidt algorithms*, Parallel Comput., 16 (1990), pp. 99–112.
[6] S. Oliveira, L. Borges, M. Holzrichter, and T. Soma, *Analysis of different partitioning schemes for parallel Gram–Schmidt algorithms*, Parallel Algorithms Appl., 14 (2000), pp. 293–320.
[7] B. N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice–Hall, Englewood Cliffs, NJ, 1980; reprinted, SIAM, Philadelphia, 1998.
[8] G. Rünger and M. Schwind, *Comparison of different parallel modified Gram–Schmidt algorithms*, in Proceedings of the 11th International Euro-Par Conference, Lecture Notes in Comput. Sci. 3648, Springer, Berlin, 2005, pp. 826–836.
[9] A. Stathopoulos and K. Wu, *A block orthogonalization procedure with constant synchronization requirements*, SIAM J. Sci. Comput., 23 (2002), pp. 2165–2182.
[10] D. Vanderstraeten, *A stable and efficient parallel block Gram–Schmidt algorithm*, in Proceedings of the 5th International Euro-Par Conference, Lecture Notes in Comput. Sci. 1685, Springer, Heidelberg, 1999, pp. 1128–1135.
[11] T. Yokozawa, D. Takahashi, T. Boku, and N. Sato, *Efficient parallel implementation of classical Gram–Schmidt orthogonalization using matrix multiplication*, in Proceedings of the 4th International Workshop on Parallel Matrix Algorithms and Applications (PMAA '06), 2006, pp. 37–38.