University of Maryland College Park

Institute for Advanced Computer Studies TR–2004–84
Department of Computer Science TR–4642

The Gram-Schmidt Algorithm and Its Variations*

G. W. Stewart†

December, 2004

ABSTRACT

The Gram–Schmidt algorithm is a widely used method for orthogonalizing a sequence of vectors. It comes in two forms: classical Gram–Schmidt and modified Gram–Schmidt, each of whose operations can be ordered in different ways. This expository paper gives a systematic treatment of this confusing variety of algorithms. It also treats the numerical issue of loss of orthogonality and reorthogonalization as well as the implementation of column pivoting.

*This report is available by anonymous ftp from `thales.cs.umd.edu` in the directory `pub/reports` or on the web at `http://www.cs.umd.edu/~stewart/`.

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 (`stewart@cs.umd.edu`). This research was supported in part by the National Science Foundation under grant CCR0204084.

The Gram-Schmidt Algorithm and Its Variations

G. W. Stewart

ABSTRACT

The Gram–Schmidt algorithm is a widely used method for orthogonalizing a sequence of vectors. It comes in two forms: classical Gram–Schmidt and modified Gram–Schmidt, each of whose operations can be ordered in different ways. This expository paper gives a systematic treatment of this confusing variety of algorithms. It also treats the numerical issue of loss of orthogonality and reorthogonalization as well as the implementation of column pivoting.

1. Introduction

The Gram–Schmidt algorithm is a method for orthogonalizing a sequence of linearly independent n-vectors $x_1, x_2, \ldots$. Specifically, it produces a sequence of vectors $q_1, q_2, \ldots,$ with q_j a linear combination of $x_1, \ldots, x_j$ that satisfy the *orthonormality conditions*

$$\|q_j\| = 1 \ (j = 1, 2, \ldots) \quad \text{and} \quad q_i^{\mathrm{T}} q_j = 0 \ (i \neq j). \tag{1.1}$$

Here q^{T} is the *transpose* of q, so that $q_i^{\mathrm{T}} q_j$ is the *inner* or *dot product* of q_i and q_j, and the quantity $\|q\|$ is the *Euclidean norm* of q, defined by

$$\|q\|^2 = q^{\mathrm{T}} q.$$

The Gram–Schmidt orthogonalization procedure has many application. Perhaps the most important is to approximate vectors by linearly combinations of the x_j, or equivalently of the q_j. Specifically, given a vector y, let

$$\hat{y} = c_1 q_1 + c_2 q_2 + \cdots c_k q_k,$$

where

$$c_j = q_j^{\mathrm{T}} y, \qquad j = 1, \ldots, k. \tag{1.2}$$

then it can be shown that $\hat{y}$ approximates y optimally in the sense that of all linear combinations of the q_i

$$\|u\| \equiv \|y - \hat{y}\|^2 \text{ is minimal.} \tag{1.3}$$

We say that $\hat{y}$ is the *least squares approximation* to y. The minimality of $\|y - \hat{y}\|^2$ is easily established by elementary methods of multivariate calculus.

Note the simplicity of the formulas (1.2) for the least squares coefficients. If we had tried to write $\hat{y}$ as a linear combination of the vectors x_i we would have ended up with a linear system of order k for the coefficients called the *normal equations.* It is little wonder, then, that the Gram–Schmidt algorithm plays an important role least squares computations.

The basic Gram–Schmidt algorithm is deceptively simple. But it comes in a number of variants, of which the main types are classical Gram–Schmidt, modified Gram–Schmidt, and Gram–Schmidt (classical or modified) with reorthogonalization. Each of the types can be varied to compute the orthogonalization coefficients in different orders. All these variants can be a source of confusion for the novice — and sometimes for the expert. This paper provides a guided tour of the world of Gram–Schmidt. The emphasis will be primarily on the structure of the various algorithms, although we will also touch on issues of efficiency and rounding error. Not much is supposed of the reader except a familiarity with elementary linear algebra and Matlab, which is the algorithmic language of this paper.

In what follows $\|\cdot\|$ will also denote the *spectral norm* defined by

$$\|X\| = \max_{\|w\|=1} \|Xw\|.$$

We will also use the *Frobenius norm* defined by

$$\|X\|_{\mathrm{F}}^2 = \sum_{i,j} x_{ij}^2.$$

We will make use of the following fact. If Q has orthonormal columns, then

$$\|QR\| = \|R\|,$$

and the same is true for the Frobenius norm.

When we come to talk about the effects of rounding error, we will have to specify the precision of computation. This is commonly summarized in a number ϵ_{M} whose logarithm approximates the number of digits (in the base of the logarithm) carried in the computation. For IEEE double-precision floating-point computation, $\epsilon_{\mathrm{M}} \cong 2.2{\cdot}10^{-16}$.

2. Matrix formulation

Although the Gram–Schmidt algorithm nominally computes a system orthonormal vectors of vectors, it actually computes an important matrix factorization. To see this we must cast the algorithm in terms of matrices.

Let us begin by assuming the vectors q_j exist. Since q_k is a linear combination of $x_1, \ldots, x_k$, we can write it in the form

$$q_k = s_{1k}x_1 + s_{2k}x_2 + \cdots + s_{kk}x_k.$$

If we set

$$X^{(k)} = (x_1\ x_2\ x_3\ \cdots\ x_k),$$
$$Q^{(k)} = (q_1\ q_2\ q_3\ \cdots\ q_k)$$

and

$$S^{(k)} = \begin{pmatrix} s_{11} & s_{12} & s_{13} & \cdots & s_{1k} \\ 0 & s_{22} & s_{23} & \cdots & s_{2k} \\ 0 & 0 & s_{33} & \cdots & s_{3k} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & s_{kk} \end{pmatrix},$$

then $S^{(k)}$ is upper triangular and

$$Q^{(k)} = X^{(k)} S^{(k)}.$$

To get a formula for $S^{(k)}$, we use the fact that the orthonormality conditions (1.1) can be written in matrix form as

$$Q^{(k)\mathrm{T}} Q^{(k)} = I,$$

where I is the identity matrix of order k (we say that $Q^{(k)}$ is *orthonormal*). Hence

$$S^{(k)\mathrm{T}} X^{(k)\mathrm{T}} X^{(k)} S^{(k)} = I.$$

Since I is nonsingular, $S^{(k)}$ must also be nonsingular. Hence if we write

$$R^{(k)} = S^{(k)-1},$$

then R_k is upper triangular and

$$X^{(k)\mathrm{T}} X^{(k)} = R^{(k)\mathrm{T}} R^{(k)}. \tag{2.1}$$

Let us look at equation (2.1) in greater detail. Since the matrix $X^{(k)}$ has linearly independent columns, the *cross-product matrix* $X^{(k)\mathrm{T}} X^{(k)}$ is positive definite; that is,

$$u \neq 0 \implies u^{\mathrm{T}}(X^{(k)\mathrm{T}} X^{(k)})u > 0.$$

It can be shown that any positive definite matrix can be factored in the form $R^{\mathrm{T}}R$, where R is a nonsingular upper triangular matrix. (The factorization is unique if we require the diagonal elements of R to be positive.) This factorization is called the *Cholesky decomposition* and R is called the *Cholesky factor.*

Since the Cholesky decomposition can be computed by off-the-shelf software, we have the following algorithm for computing $Q^{(k)}$.

```
1. Compute the Cholesky factor $R^{(k)}$ of $X^{(k)\mathrm{T}} X^{(k)}$
2. $Q^{(k)} = X^{(k)} R^{(k)-1}$                                    (2.2)
```

Although this algorithm represents a constructive proof of the existence of $Q^{(k)}$, it is seldom, if ever, used. Not only is it more expensive than the alternatives, but it is numerically less stable. We will now consider a better way.

3. The QR factorization and the classical Gram–Schmidt algorithm

We are going to derive the classical Gram–Schmidt algorithm by showing that it computes a QR factorization. Specifically, the *QR factorization* is a factorization of a matrix X into the produce QR of an orthonormal matrix and an upper triangular matrix. An example is the factorization $X^{(k)} = Q^{(k)}R^{(k)}$ implicit in line 2 of (2.2). QR factorizations are widely used in numerical applications. Let us pause briefly to examine one of the most important.

Suppose, as in §1 we wish to approximate a vector y as a linear combination of the vectors $x_1, \ldots, x_k$ (instead of $q_1, \ldots, q_k$) in the least squares sense. We can write this problem in matrix form as: Determine a vector b such that

$$\|y - X^{(k)}b\|^2 = \min .$$

Then it can be shown that the vector b is the solution of the the upper triangular system

$$R^{(k)}b = Q^{(k)\mathrm{T}}y,$$

where $X^{(k)} = Q^{(k)}R^{(k)}$ is the QR factorization of $X^{(k)}$. It is a worthwhile exercise derive this from (1.2) and (1.3).

There are many ways of computing QR decompositions, of which the Gram–Schmidt algorithm is just one. To derive the classical Gram–Schmidt (CGS) algorithm, we assume that we have the QR factorization

$$X^{(k-1)} = Q^{(k-1)}R^{(k-1)}, \tag{3.1}$$

and wish to compute the factorization $X^{(k)} = Q^{(k)}R^{(k)}$, which we will partition in the form

$$(X^{(k-1)} \;\; x_k) = (Q^{(k-1)} \;\; q_k) \begin{pmatrix} R^{(k-1)} & r_k \\ 0 & \rho_k \end{pmatrix} .$$

Computing the first column of this partition, we get $X^{(k-1)} = Q^{(k-1)}R^{(k-1)}$, which is just (3.1). But if we compute the second column we get something new:

$$x_k = Q^{(k-1)}r_k + \rho_k q_k. \tag{3.2}$$

Since $Q^{(k-1)\mathrm{T}}Q^{(k-1)} = I$, and $Q^{(k-1)\mathrm{T}}q_k = 0$, we have on multiplying the above relation by $Q^{(k-1)\mathrm{T}}$ that

$$r_k = Q^{(k-1)\mathrm{T}}x_k. \tag{3.3}$$

Rewriting (3.2) in the form

$$\rho_k q_k = x_k - Q^{(k-1)}r_k \equiv u_k \tag{3.4}$$

and recalling that $\|q_k\| = 1$, we have

$$\rho_k = \|u_k\|. \tag{3.5}$$

Finally,

$$q_k = \rho_k^{-1}\|u_k\|. \tag{3.6}$$

Equations (3.3), (3.4), (3.5), and (3.6) are effectively an algorithm for computing the expanded decomposition. We can start the process by observing that $\rho_1 = \|x_1\|$, and $q_1 = x_1/\rho_1$. All this leads to the following algorithm.

```
        Startup
 1.     $\rho_1 = \|x_1\|$
 2.     $q_1 = x_1/\rho_1$
        Main loop
 3.     for $k = 2, 3, \ldots$
 4.        $r_k = Q^{(k-1)\mathrm{T}} x_k$
 5.        $u_k = x_k - Q^{(k-1)} r_k$                                   (3.7)
 6.        $\rho_k = \|u_k\|$
 7.        $q_k = u_k/\rho_k$
 8.        $Q^{(k)} = (Q^{(k-1)}\ \ q_k)$
 9.        $R^{(k)} = \begin{pmatrix} R^{(k-1)} & r_k \\ 0 & \rho_k \end{pmatrix}$
10.     end
```

In §1 we defined q_k as being a linear combination of $x_1, \ldots, x_k$. But when we tried to compute the coefficients of this linear combination, we obtained the awkward algorithm (2.2). The CGS algorithm builds q_k as a linear combination of x_k, $q_1, \ldots, q_{k-1}$, which accounts for its basic simplicity. In particular, the R-factor in the QR factorization of $X^{(k)}$ is built up column by column.

Note that r_k contains the coefficients for the least-squares approximation to x_k [see (1.2)]. This means that u_k is the residual that is left over after the least squares approximation is subtracted out [see (1.3)]. If $\rho_k = \|u_k\|$ is small, then x_k is nearly dependent on $x_1, \ldots, x_{k-1}$. We shall see later that in the presence of rounding error this results in loss of orthogonality in q_k. It can even happen that u_k, computed in line 5, is zero. In that case our algorithm would die in line 7 with a divide by zero. An industrial strength implementation would check ρ_k and take corrective action if it is zero. Because the algorithms in this paper are for expository purposes only, we will omit such tests.

In practice, we would not define separate matrices $Q_1, Q_2, \ldots$. Instead we would allocate storage for the largest Q and R that we expect to encounter and build up Q

and R within that storage. The following Matlab code shows how this can be done. It is called `cgscol` because it builds up the matrix R column by column.

```
 1. function [Q, R] = cgscol(X)
 2.   [n, p] = size(X);              % X is nxp
 3.   Q = zeros(n,p);                % So is Q
 4.   R = zeros(p);                  % R is pxp
 5.   for k=1:p
 6.      R(1:k-1,k) = Q(:,1:k-1)'*X(:,k);                (3.8)
 7.      u = X(:,k) - Q(:,1:k-1)*R(1:k-1,k);
 8.      R(k,k) = norm(u);
 9.      Q(:,k) = u/R(k,k);
10.   end
11. return
```

Note that this code contains is no equivalent of the startup in (3.7). Instead it takes advantage of the fact that Matlab can manipulate matrices with zero dimensions — sometimes called *null* or *empty* matrices. Specifically, when `k=1`, `R(1:k-1,k)` in line 6 is a `0x1` matrix formed as the product of the `0xn` matrix `Q(:,1:k-1)'` and the `nx1` matrix `X(:,k)`. In line 7, the product `Q(:,1:k-1)*R(1:k-1,k)` is `nx1`, since `Q(:,1:k-1)` is `nx0` and `R(1:k-1,k)` `0x1`. When a nonnull matrix appears out of thin air, as does this product, Matlab initializes it to zero. Thus `u` is just `X(:,k)`, which is what we want when $k = 1$.

As we have noted, our classical Gram–Schmidt algorithm — either (3.7) or (3.8) — builds up Q and R a column at a time. Alternatively, we can build up Q a column at a time and R a row at a time. Specifically, from the relation $X = QR$, we have

$$R = Q^{\mathrm{T}}X.$$

Hence if we know q_k, we can generate the kth row r_k^{T} of R by the formula

$$r_k^{\mathrm{T}} = q_k^{\mathrm{T}}X. \tag{3.9}$$

Of course, there is no need to compute the first k elements of r_k^{T}, since they are zero.

The following function implements this idea. In analogy with `mgscol`, which generates R by columns, it is called `mgsrow`.

```
function [Q, R] = cgsrow(X)
   [n,p] = (size(X));
   Q = zeros(n,p);
   R = zeros(p,p);
   for k=1:p
     u = X(:,k) - Q(:,1:k-1)*R(1:k-1,k);
     R(k,k) = norm(u);
     Q(:,k) = u/R(k,k);
     R(k,k+1:p) = Q(:,k)'*X(:,k+1:p);
   end
return
```

(3.10)

At the beginning of the loop on `k` is it assumed that we have computed the first `k-1` columns of `Q` and rows of `R`. This means that we can immediately compute `u` as in line 7 in (3.8), and go on to compute `R(k,k)` and `Q(:,k)` as usual. One then uses `Q(:,k)` to compute `R(k,k+1:p)` as suggested by (3.9).

In some applications x_k will not be known until q_{k-1} has been computed. For example, in the Lanczos and Arnoldi algorithms for computing eigenpairs of a matrix A, the vector x_k is Aq_{k-1}. In this case the row algorithm (3.10) cannot be used, since it presupposes that we have all of X available. The column algorithm (3.8) does not actually use x_k until it is time to form q_k, but it still takes the full X as input.

A solution to these problems is the following program that performs a single step of CGS.

```
function [q, r, rho] = cgsinc(Q, x)
   r = Q'*x;
   u = x - Q*r;
   rho = norm(u);
   q = u/rho;
return
```

(3.11)

The following script shows how `cgsinc` (`inc` for *incremental*) is used.

```
n = 5; p = 3;
Q = zeros(n,p);
X = zeros(n,p);
R = zeros(p);
for k=1:p
   X(:,k) = randn(n,1);   % Or whatever.
   [Q(:,k), R(1:k-1,k), R(k,k)] = cgsinc(Q(:,1:k-1), X(:,k));
end
```

Of course, the code for `cgsinc` is so simple that it could simply be inlined into the application program in question, as is often done in practice.

In comparing algorithms it is sometimes useful to know how many floating-point operations the algorithms take. For the function `cgsinc`, as the number of columns k of `Q` increases, the bulk of the arithmetic is concentrated in lines 2 and 3, each requiring about nk additions and multiplications. Thus the total number of additions and multiplications to compute the QR factorization of an $n \times p$ matrix is

$$2\sum_{k=1}^{p} nk \cong np^2.$$

When $n = p$ this count is n^3, which may be compared with a count n^3 for a matrix multiplication or $\frac{1}{3}n^3$ for Gaussian elimination.

When the variants of the CGS algorithm are executed with rounding error they behave similarly. In fact, if the requisite matrix and vector operations are implemented according to their 'natural' definitions, the algorithms produce exactly the same results. The reason is that the algorithms consist of independent computational tasks that can be reordered without changing the rounding errors. For example, in both the row and column algorithms r_{ij} is computed as $q_i^{\mathrm{T}} x_j$, although the individual r_{ij} are not computed in the same order.

We will return to rounding error later when we consider loss of orthogonality. But first, we shall consider the modified Gram–Schmidt algorithm.

4. The modified Gram–Schmidt algorithm

The three variants of the CGS algorithm are essentially the same in their operation counts and numerical properties. The modified Gram–Schmidt (MGS) algorithm has the same operation count but different numerical properties. In this section we will concern ourselves with the algorithm itself, and treat its numerical properties in §6.

The modified Gram–Schmidt algorithm can be derived by considering the computation of u_k in line 5 in the column algorithm (3.7). Specifically, we have

$$u_k = x_k - r_{1k}q_1 - r_{2k}q_2 - \cdots - r_{k-1,k}q_{k-1}, \tag{4.1}$$

where r_{jk} is computed from the formula $r_{jk} = q_j^{\mathrm{T}} x_k$ (line 4). In the CGS algorithm the coefficients r_{jk} are all computed in one step (line 4) and then u_k is computed in the next (line 5).

But by the orthogonality of the q's we have an alternative formula for r_{jk}: namely,

$$r_{jk} = q_j^{\mathrm{T}}(x_k - r_{1k}q_1 - r_{2k}q_2 - \cdots - r_{j-1,k}q_{j-1})$$

Hence we can alternate the computation of the r_{jk} with the subtraction of $r_{jk}q_j$ as shown below.

```
1. $u_k = x_k$
2. **for** $j = 1$ **to** $k-1$
3.     $r_{jk} = q_j^{\mathrm{T}} u_k$
4.     $u_k = u_k - r_{jk} q_j$
5. **end**
```

These ideas lead to the following MGS column code.

```
function [Q, R] = mgscol(X)
   [n,p] = (size(X));
   Q = zeros(n,p);
   R = zeros(p,p);
   for k=1:p
      u = X(:,k);
      for j=1:k-1
         R(j,k) = Q(:,j)'*u;
         u = u - R(j,k)*Q(:,j);
      end
      R(k,k) = norm(u);
      Q(:,k) = u/R(k,k);
   end
return
```

There is a row version of the MGS algorithm. It is short, slick, and not easy to derive. We begin by considering the partitioned QR decomposition

$$X = (X_1^{(k-1)} \; X_2^{(k-1)}) = (Q_1^{(k-1)} \; Q_2^{(k-1)}) \begin{pmatrix} R_{11}^{(k-1)} & R_{12}^{(k-1)} \\ 0 & R_{22}^{(k-1)} \end{pmatrix}.$$

Here $R_{11}^{(k-1)}$ is $(k-1)\times(k-1)$. Note that our old friends $X^{(k-1)}$, $Q^{(k-1)}$, and $R^{(k-1)}$ have acquired subscripts to indicate their positions in the partition.

Now suppose we have computed $Q^{(k-1)}$ and

$$(R_{11}^{(k-1)} \; R_{12}^{(k-1)}),$$

Suppose, in addition, we have computed

$$Y_2^{(k-1)} \equiv (y_k^{(k-1)} \; \cdots \; y_p^{(k-1)}) = X_2^{(k-1)} - Q_1^{(k-1)} R_{12}^{(k-1)}. \qquad (4.2)$$

Now for $j \geq k$

$$y_j^{(k-1)} = x_j - r_{1j} q_1 - \cdots - r_{k-1,j} q_{k-1}.$$

Comparing this with (4.1), we see that $y_{k-1}^{(j)}$ contains x_j partially reduced by $q_1, \ldots, q_{k-1}$. In particular, $y_k^{(k-1)}$ is fully reduced so that with $r_{kk} = \|y_{k-1}^{(k)}\|$ we have $q_k = y_k/r_{kk}$. Moreover, the last row of $R_{12}^{(k)}$ is

$$q_k^{\mathrm{T}}(y_{k+1}^{(k-1)} \ \cdots \ y_p^{(k-1)}) \equiv \hat{r}^{\mathrm{T}}. \tag{4.3}$$

Furthermore,

$$Y_2^{(k)} = (y_{k+1}^{(k-1)} \ \cdots \ y_p^{(k-1)}) - q_k\hat{r}^{\mathrm{T}}. \tag{4.4}$$

There is one final trick. We do not need to maintain $Y_2^{(k)}$ separately. If we start with $Q = X$ we can perform all the above manipulations in the array containing Q. Here is the Matlab algorithm.

```
function [Q, R] = mgsrow(X)
[n,p] = size(X);
Q = X;
R = zeros(p,p);
for k=1:p
   R(k,k) = norm(Q(:,k));                                        (4.5)
   Q(:,k) = Q(:,k)/R(k,k);
   R(k,k+1:p) = Q(:,k)'*Q(:,k+1:p);          [see (4.3)]
   Q(:,k+1:p) =
        Q(:,k+1:p) - Q(:,k)*R(k,k+1:p);      [see (4.4)]
end
```

To follow this code, note that at the beginning of the kth step, `Q(:,1:k-1)` contains $X^{(k-1)}$ while `Q(:,k:p)` contains $Y_2^{(k-1)}$.

There is also an incremental MGS algorithm. Here is the code.

```
function [q, r, rho] = cgsinc(Q, x)
   p = size(Q, 2);
   r = zeros(p,1);
   u = x;
   for k=1:p
      r(k) = Q(:,k)'*u;
      u = u - r(k)*Q(:,k);
   end
   rho = norm(u);
   q = u/rho;
return
```

5. Some timings

We now have six algorithms: the column, row, and incremental versions of CGS and MGS. They all require np^2 floating-point additions and multiplications to compute the QR factorization of an $n\times p$ matrix. In this section we will see how this translates into timings.

The table below gives the times in seconds required to process a 5000×200 matrix on two UltraSPARCs — one at the University Maryland (900 MHz) and the other at NIST (360 MHz). The Matlab versions were 6.5.0 (UMD) and 6.5.1 (NIST). To indicate the variability in the timings, two times are given for each combination of algorithm and machine.

	UMD			NIST	
	cgs	mgs		cgs	mgs
col	9.9/10.8	5.2/ 5.0	col	9.1/ 9.4	10.6/10.8
row	9.7/ 9.5	16.8/16.4	row	10.1/12.2	29.0/28.0
inc	7.1/ 6.0	9.7/10.3	inc	6.9/ 7.9	12.7/13.6

The numbers seem to reflect more the vagaries of Matlab than the properties of the algorithm. They are all in the same ball park, but are not consistent in ranking the algorithms. At UMD `mgscol` beats `cgscol`, but at NIST they are approximately equal. On both machines `cgsinc` is good, but the disparity with `mgsinc` is greater at NIST than at UMD. At both places `mgsrow` is the big loser.

The numbers do not reflect the optimum speedup of 2.5, corresponding to the ratio of the clock rates at UMD and NIST. To see if such speedups are possible, the statements

```
tic, [Q, R] = qr(X, 0), toc
```

were executed on both machines. This times the computation of a QR factorization by an LAPACK routine that, properly supported, should run at close to peak speed. The times in seconds were 4.8 (NIST) and 2.0 (UMD), whose ratio of 2.4 is comfortably near the optimum speedup.

6. Loss of orthogonality

The curse of Gram–Schmidt orthogonalization — either classical or modified — is that it may not produce orthogonal vectors in the presence of rounding error. Figure 6.1 shows a simple example of dramatic loss of orthogonality. The results of each statement were rounded to five decimal digits before assignment using a utility function `rnd` — e.g., the actual statement that produced `q1` was

```
q1 = rnd(x1/r11, 5)
```

```
 1.  n = 4
 2.  X = condgen(n, 2, 4)
        1.4370e-01  -1.5931e-01
        1.4545e-01  -1.6144e-01
       -6.3207e-01   7.0098e-01
        8.4332e-02  -9.3573e-02
 3.  x1 = X(:,1); x2 = X(:,2);
 4.  r11 = norm(x1)
        6.6965e-01
 5.  q1 = x1/r11
        2.1459e-01
        2.1720e-01
       -9.4388e-01
        1.2593e-01
 6.  r12 = q1'*x2
       -7.4268e-01
 7.  r12q1 = r12*q1
       -1.5937e-01
       -1.6131e-01
        7.0100e-01
       -9.3526e-02
 8.  u = x2 - r12q1
        6.0000e-05
       -1.3000e-04
       -2.0000e-05
       -4.7000e-05
 9.  r22 = norm(u)
        1.5202e-04
10.  q2 = u/r22
        3.9468e-01
       -8.5515e-01
       -1.3156e-01
       -3.0917e-01
11.  q1'*q2 =
       -1.5801e-02
```

Figure 6.1: Loss of orthogonality in the Gram–Schmidt algorithm

This rounding means that we cannot expect `q1'*q2` to be much less than 10^{-5}.

The statement

```
   X = condgen(n, 2, 4)
```

generates a random $n\times 2$ matrix with singular values of 1 and 10^{-4} (more on singular values later). The orthogonalization proceeds without apparent exception up to the computation of `u`. The vectors `x2` and `r12q1` agree to about three decimal digits, and consequently there is cancellation of significant digits in the computation of `u`, as evidenced by the small size of `u` and the zero digits in its components. The normalized `q2` has a dot product with `q1` that is three orders of magnitude greater than the desired value of 10^{-5}.

It is sometimes asserted that the cancellation in line 8 is responsible for the loss of orthogonality. But it is easy to verify that the computation of `u` entails no rounding error. If the entire computation were exact, the zero digits in `u` would have had nonzero values. But the information required to compute those values was lost when we rounded `r12q1` to five digits. That, not the cancellation, is what causes the loss of orthogonality.

The brevity of the computation also makes it clear that accumulation of rounding error over a period of time is not the cause of loss of orthogonality. In fact, the five rounding errors made in rounding `r12p1` are alone sufficient to cause the loss of orthogonality.

We have observed that $\|u_k\|$ is small if and only if x_k is nearly dependent on $x_1, \ldots, x_{k-1}$ [see the discussion following (3.7)]. When it is small, its computation will naturally involve cancellation. Consequently, there is an association between linear dependence among the columns of X and loss of orthogonality.

To develop this idea we must introduce singular values and their associated vectors. Specifically, for any $n\times p$ matrix x with $n \geq p$, there are two systems of orthonormal vectors $u_1, u_2, \ldots, u_p$ and $v_1, v_2, \ldots, v_p$ such that

$$Xv_j = \sigma_j u_j \quad \text{and} \quad X^{\mathrm{T}} u_j = \sigma_j v_j, \qquad j = 1, \ldots, p, \tag{6.1}$$

where

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0.$$

The scalars σ_j are the *singular values* of X and the vectors u_j and v_j are the *left and right singular vectors* of X.

The connection of singular values with linear dependence is contained in the following result. Let $E = -\sigma_p u_p v_p^{\mathrm{T}}$. Then $\|E\| = \sigma_p$ and $(X+E)v_p = 0$. [This fact can be verified directly from (6.1).] Writing this relation in the form

$$v_1^{(p)}(x_1 + e_1) + v_2^{(p)}(x_2 + e_2) + \cdots v_p^{(p)}(x_{p1} + e_p) = 0,$$

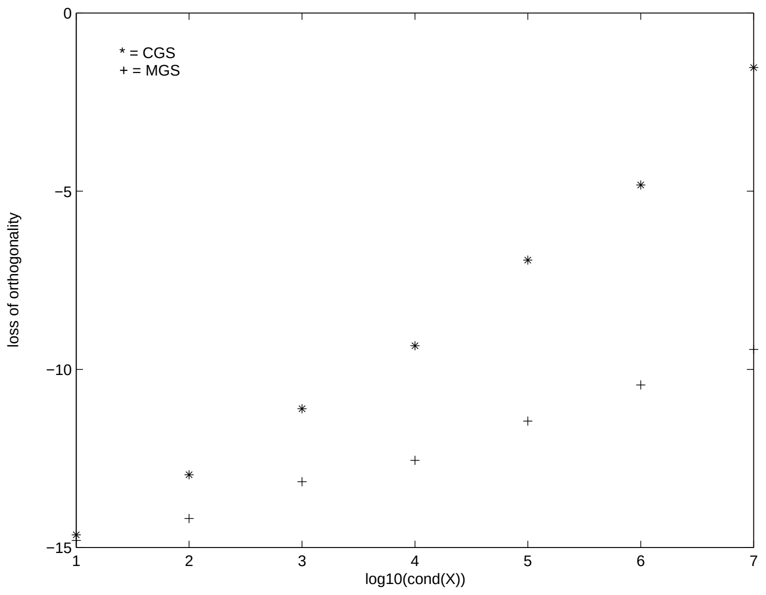

Figure 6.2: Loss of orthogonality in 50×20 matrices of incresing condition number.

we see that the columns of $X + E$ are linearly dependent. Thus if σ_p is small, X is near in norm to a rank-degenerate matrix.

In practice we must qualify the term 'small' in the preceding sentence. If X is multiplied by a constant α, then the singular values of X are multiplied by α. Thus σ_p can be made as small as we like simply by scaling X by a constant. But such scaling should not affect the independence of the columns of X.

To get around this problem it is customary to work with the quantity

$$\kappa(X) = \sigma_1/\sigma_p,$$

which is easily seen to be independent of the scaling of X. It is called the *condition number* of X, and if it is large, then the columns of X are nearly dependent. (Actually, this statement needs further qualification. For the condition number to be meaningful, the columns of X must all be of the same order of magnitude. Unfortunately, a discussion of this fascinating topic would lead us too far astray.)

The major difference between the CGS and MGS methods is the rate at which they loose orthogonality. This fact is illustrated by the graphs in Figure (6.2). It plots the common logarithm of the loss of orthogonality as measured by $\|I - Q^{\mathrm{T}}Q\|$ against the common logarithm of the condition number for a sequence of 50×20 matrices. For both the CGS and the MGS algorithms the relations are approximately linear, but the slope of the line for the CGS method is approximately two, whereas for the MGS method it is approximately one. Since the slope of a log-log plot indicates a power relation, in this example the loss of orthogonality in the MGS method is proportional to the condition number, whereas in the CGS method is proportional to the square of the condition number.

This result can be proved rigorously, provided that $\kappa(X)\epsilon_{\mathrm{M}}$ is sufficiently less than one ('sufficiently' depends on the dimensions of the matrix X). This means that in applications where it is desired to retain orthogonality, the MGS method is to be preferred to the CGS method.

7. Reorthogonalization

For the price of doubling the work in the Gram–Schmidt algorithm one can obtain a Q that is orthogonal to working accuracy. The idea is to repeat the orthogonalization. The following code gives the CGS column version.

```
 1. function [Q, R] = cgsrocol(X)
 2.    [n,p] = (size(X));
 3.    Q = zeros(n,p);
 4.    R = zeros(p,p);
 5.    for k=1:p
 6.       r1 =  Q(:,1:k-1)'*X(:,k);
 7.       u1 = X(:,k) - Q(:,1:k-1)*r1;
 8.       r2 = Q(:,1:k-1)'*u1;
 9.       u2 = u1 - Q(:,1:k-1)*r2;
10.       R(1:k-1,k) = r1 + r2;
11.       R(k,k) = norm(u2);
12.       Q(:,k) = u2/R(k,k);
13.    end
14. return
```

From this it is seen that having computed `u1` (which is `u` in our other algorithms), one orthogonalizes it against `Q`. The result `u2` is accepted as the unnormalized `Q(:,k)`. To preserve the relation $X = QR$, it is necessary to combine the two sets of orthogonalization coefficients, as is done in line 10.

The remarkable fact about this algorithm is that if $\kappa(X)\epsilon_{\mathrm{M}}$ is sufficiently less than one then the computed Q is orthogonal to working accuracy in the sense that $\|I - Q^{\mathrm{T}}Q\|$ is near ϵ_{M}. What makes this fact remarkable is that only one reorthogonalization is required to produce this degree of orthogonality. However, if the hypothesis on the condition number of X is violated, then `u1` or `u2` in the algorithm may be zero or `u2` may also suffer loss of orthogonality. A complete implementation would take these problems into account.

The reorthogonalization can be skipped if there is no cancellation in computing `u1` in line 7. This will be true if `norm(u1)/norm(X(:,k)) > 0.5`. If p is even moderately large large, say greater than 10, the extra norm computation in this test will be an insignificant part of the calculation.

Reorthogonalization is applicable to all our six of our CGS and MGS algorithms. Thus we have an is an ensemble twelve variants of the Gram–Schmidt algorithm. However, with reorthogonalization, the MGS algorithm has no numerical advantages over the CGS algorithm. Since the CGS algorithm is simpler, it is often preferred in this context.

8. Reduced-rank approximations and pivoting

Caution: This section is more difficult that its predecessors and may be skipped with out loss of continuity.

In many applications it is necessary to approximate an $n\times p$ matrix X by a matrix of lower rank, say rank k. Such an approximation can be written in the form

$$X \cong WZ^{\mathrm{T}},$$

where W is $n\times k$ Z and is $p\times k$, each having rank k. Such an approximation can save both storage and computations. For example, it requires $(n+p)k$ floating-point words to store W and Z as opposed to np for X. Likewise, the operation count for computing the matrix-vector product $WZ^{\mathrm{T}}a$ is $(n+p)k$ is $(n+p)k$ additions and multiplication, as opposed to np to form Xa. If a satisfactory approximation can be found for small k, the savings can be impressive.

The QR factorization furnishes a reduced-rank approximation. To see this, let us partition the QR decomposition of X in the form

$$X = (X_1^{(k)}\ X_2^{(k)}) = (Q_1^{(k)}\ Q_2^{(k)}) \begin{pmatrix} R_{11}^{(k)} & R_{12}^{(k)} \\ 0 & R_{22}^{(k)} \end{pmatrix}. \tag{8.1}$$

Multiplying out this decomposition, we have

$$X = Q_1^{(k)}(R_{11}^{(k)}\ R_{12}^{(k)}) + Q_2^{(k)}(0\ R_{22}^{(k)}).$$

Dropping the second term in this sum, we obtain our approximation

$$X \cong Q_1^{(k)}(R_{11}^{(k)}\ R_{12}^{(k)})$$

The error in the approximation is the norm of the term we have ignored:

$$\|Q_2^{(k)}R_{22}^{(k)}\| = \|R_{22}^{(k)}\|.$$

We can use either `cgsrow` or `mgsrow` to compute this decomposition. However, only `mgsrow` provides the wherewithal to calculate $\|R_k^{(22)}\|$. To see this, note that from (4.2) and (8.1) we have

$$Y_2^{(k)} = X_2^{(k)} - Q_1^{(k)}R_{12}^{(k)} = Q_2^{(k)}R_{22}^{(k)},$$

It follows that

$$\|R_{22}^{(k)}\|_{\mathrm{F}} = \|Y_2^{(k)}\|_{\mathrm{F}}.$$

Since `mgsrow` computes $Y_2^{(k)}$ in line 9 of (4.5), we can compute its norm and check if the current approximation is sufficiently accurate.

Unfortunately, the particular order in which the columns of X appear may not give a good reduced-rank approximation to X. For example, consider the matrix

$$X_{\mathrm{bad}} = \begin{pmatrix} 1.0000 & 1.0000 & 0.0000 \\ 1.0000 & 1.0010 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{pmatrix} \tag{8.2}$$

The R-factor computed by Matlab is

```
R_bad =
  -1.4142e+00  -1.4149e+00            0
            0   7.0711e-04            0
            0            0   1.0000e+00
```
(8.3)

From this we see that a rank two approximation to X obtained from the QR factorization will have a error norm of one — corresponding to the element in the southeast corner of `R_bad`. On the other hand, if we interchange the second and third columns of X_{bad} to give

$$X_{\mathrm{good}} = \begin{pmatrix} 1.0000 & 0.0000 & 1.0000 \\ 1.0000 & 0.0000 & 1.0010 \\ 0.0000 & 1.0000 & 0.0000 \end{pmatrix}$$

we get the R-factor

```
R_good =
  -1.4142e+00            0  -1.4149e+00
            0  -1.0000e+00            0
            0            0  -7.0711e-04
```

Thus a rank two approximation based on `R_good` will have an error norm of about $7{\cdot}10^{-4}$. If we interchange the second and third columns of this approximation, we get an equally good approximation to X_{bad}.

Thus we wish to adaptively interchange columns as the QR decomposition is computed to enhance the rank of $X^{(k)}$. The most common strategy for selecting a column is the following. Suppose that we have computed $X_1^{(k-1)}$ and $Y_2^{(k-1)}$. Then choose the column for which $\|y_j^{(k-1)}\|$ $(j = k, \ldots, p)$ is maximal. When this column is interchanged with the kth column of $Y_2^{(k-1)}$, the diagonal element will be r_{kk} as large as possible, and this tends to make $R_{11}^{(k)}$ well conditioned. In particular, it would not allow the small element to appear as the second diagonal element of `R_bad` in (8.3). Note that when we interchange the columns of $Y_2^{(k-1)}$, we must also interchange the corresponding columns of $R_{12}^{(k-1)}$.

The function `mgscp` (`cp` for ‘column pivoting’) in Figure 8.1 implements this pivoting strategy. The function takes as input the matrix X and a error tolerance, which is used to determine the rank of the approximation. Returned are the Q- and R-factors and the rank of the approximation, along with an array of pivot columns.

The basic loop is the one in the function `mgsrow` but with two additions at the front end. In the first the norms of the columns of $Y_2^{(k-1)}$ are computed and stored in the array `normy`. From this the Frobenius norm of R_{22} is computed and used to determine if the rank `k-1` approximation already computed is adequate. If it is, `Q`, `R`, and `pvt` are trimmed, and the function returns.

The second addition determines the pivot column. Note that `pvt(k)` contains the index of the column that was swapped with column `k`. The swapping is actually done on both $Y_2^{(k-1)}$ and $R_{22}^{(k-1)}$, as mentioned above.

The MGS step is unaltered. It could easily be expanded to include reorthogonalization, and for most applications probably should be. The main reason we have not done so here, is to allow the code to fit on a single page.

When this algorithm is applied to X_{bad} in (8.2), with `err` $=$ `0.01` the output is

```
Q =
   7.0675e-01            0
   7.0746e-01            0
            0   1.0000e+00
R =
   1.4149e+00            0   1.4142e+00
            0   1.0000e+00            0
rank =
     2
pvt =
function [Q, R, rank, pvt] = mgscp(X, err)
   [n,p] = size(X);
   Q = X;
   R = zeros(p,p);
   normy = zeros(1,p);
   pvt = zeros(1,p);
   for k=1:p
%
%     Compute the norms of y and test for convergence.
%
      for j=k:p
         normy(j) = norm(Q(:,j));
      end
      if norm(normy(k:n)) <= err    % same as norm(R22) <= err
         rank = k-1;
         Q = Q(:,1:rank); R = R(1:rank,:); pvt = pvt(1:rank);
         return;
      end
%
%     Determine the pivot column and exchange.
%
      [maxnormy, pvt(k)] = max(normy(k:p));
      pvt(k) = pvt(k) + k - 1;
      temp=Q(:,k); Q(:,k)=Q(:,pvt(k)); Q(:,pvt(k))=temp;
      temp=R(1:k-1,k); R(1:k-1,k)=R(1:k-1,pvt(k)); ...
                           R(1:k-1,pvt(k))=temp;
%
%     MGS step.
%
      R(k,k) = norm(Q(:,k));
      Q(:,k) = Q(:,k)/R(k,k);
      R(k,k+1:p) = Q(:,k)'*Q(:,k+1:p);
      Q(:,k+1:p) = Q(:,k+1:p) - Q(:,k)*R(k,k+1:p);
   end
   rank = p;
return
```

Figure 8.1: MGS with column pivoting

2 3 0

Note that to get X_{good} in our example, we exchanged columns 2 and 3 of X_{bad}. The algorithm `msgcp`, on the other hand, makes two interchanges: first between columns 1 and 2 and then between columns 2 and 3. The reason for the first interchange is that column 2 is slightly larger than column 3. But in the end, the result is an approximation with essentially the same error.

The computation of the norms increases the operation count by $\frac{1}{2}np^2$ additions and multiplications over np^2 for the basic algorithm without reorthogonalization or $2np^2$ with reorthogonalization. Alternatively, the formula

$$\|y_j^{(k)}\|^2 = \|y_j^{(k-1)}\|^2 - r_{jk}^2$$

could be used to update the norms as the computation proceeds. But this formula is tricky to use in the presence of rounding error.

We should stress that the pivoting strategy adopted here is not foolproof — there are counterexamples where it fails to find approximations of suitably low rank — even though such exist. But these failures are very rare, and the alternatives are very complicated.

9. Envoi

We have seen that that there are twelve version of the Gram–Schmidt algorithm: classical and modified versions that compute R by rows, columns, or incrementally, with or without reorthogonalization. The choice of which to use in a given situation will depend on the problem at hand — especially on how the vectors x_j are generated and what parts of R are needed at any given time. If no reorthogonalization is to be performed, then MGS will help control the loss of orthogonality. With reorthogonalization the balance shifts to CGS.

The alternative to Gram–Schmidt is orthogonal triangularization, which forms Q as the initial p columns of a product of certain elementary orthogonal matrices — either Householder transformations or plane rotations. In the case of Householder transformations, the product is not explicitly computed. Instead vectors from which the transformations can be recovered are stored. Orthogonality to working accuracy is guaranteed. Plane rotations are generally used on structured matrices where full Householder transformations or Gram–Schmidt algorithms are inappropriate. Hence any comparison comes down to Householder vs. Gram–Schmidt.

For an $n{\times}p$ matrix X, the ratio of operations counts of Householder to Gram-Schmidt is $1 - \frac{1}{3}\frac{p}{n}$. Thus when $p = n$, Householder triangularization has $\frac{2}{3}$ the count of Gram–Schmidt. But as n increases, the ratio quickly approaches one. To guarantee orthogonality with Gram–Schmidt, however, one must reorthogonalize, which increases the ratio to two. Given these ratios and guaranteed orthogonality, one can ask why use Gram–Schmidt methods at all. There are several answers.

First, although it is easy to code an incremental version of Householder triangularization (if you know how), none of the major linear algebra packages provide software to do it. Consequently, Gram–Schmidt is preferred in orthogonalizing Krylov sequences and their relatives.

Second, Householder triangularization represents Q in a coded form that is not easy to manipulate. In fact, there are tasks that cannot be done efficiently, or even at all, without generating Q. Examples are computing the diagonal elements of QQ^{T} or recomputing the factorization after a row is appended to X. In these cases, Q must be generated explicitly from the Householder transformations, which puts it on a par with CGS with reorthogonalization.

Third, Householder reduction is subject to subtle instabilities when the rows of X vary widely in magnitude — instabilities that do not affect the Gram–Schmidt algorithm.

Finally, we have confined ourselves to the Euclidean inner product $u^{\mathrm{T}}v$. The Gram-Schmidt can easily be adapted to oblique inner products. Although there exist generalizations of Householder transformations to vector spaces with oblique inner products, there is no off-the-shelf software supporting them.

These reasons coupled with the basic simplicity of the Gram–Schmidt process insure that Gram–Schmidt in its several versions will remain a part of the general toolkit for matrix computations.

10. Bibliography

Both Gram and Schmidt were concerned with the orthogonalization of functions rather than vectors. Gram [3] developed determinantal expressions for the orthogonalized sequence and made the connection with least squares. Schmidt's algorithm [5] is essentially classical the classical Gram–Schmidt algorithm in the context of integral equations.

There is a large corpus on Gram–Schmidt. Fortunately, much of it has been incorporated, with historical comments, in general texts on numerical linear algebra; e.g. [1, 2, 6]. These texts also discuss pivoting, orthogonal triangularization, and other topics touched on in this paper.

For more on oblique Householder transformations see [4].

Acknowledgement

I am grateful to the Mathematical and Computational Sciences Division for their support.

References

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

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

[3] J. P. Gram. Über die Entwicklung reeller Functionen in Reihen mittelst der Methode der kleinsten Quadrate. *Journal für die reine und angewandte Mathematik*, 94:41–73, 1883.

[4] D. S. Mackey, N. Mackey, and F. Tisseur. G-reflectors in scalar product spaces. Numerical Analysis Report 420, Manchester Center for Computational Mathematics, 2003.

[5] E. Schmidt. Zur Theorie der linearen und nichtlinearen Integralgleichungen. I Teil. Entwicklung willkürlichen Funktionen nach System vorgeschriebener. *Mathematische Annalen*, 63:433–476, 1907.

[6] G. W. Stewart. *Matrix Algorithms I: Basic Decompositions.* SIAM, Philadelphia, 1998.