

## Lecture 5: Householder QR details; QR via Gram–Schmidt

### 1.2.5. Computational complexity of the QR decomposition.

The following MATLAB program, an implementation of Trefethen and Bau's Algorithm 10.1, provides a more efficient implementation of the Householder QR factorization than the one presented in the last lecture.

```
function [V,R] = householder_qr(A)

% Compute the QR factorization of real A using Householder reflectors
% See Trefethen and Bau, Numerical Linear Algebra, Algorithm 10.1 (page 73)
% The object V is a "cell array": vk (a vector of length m-k+1) is stored in V{k}.

[m,n] = size(A);
Q = eye(m);
for k=1:min(m-1,n)
    ak = A(k:m,k); % vector to be zeroed out
    vk = ak; vk(1) = vk(1) + sign(ak(1))*norm(ak); % 1. construct vk that defines the reflector
    vk = vk/norm(vk); % 2. normalize vk
    A(k:m,k:n) = A(k:m,k:n) - 2*vk*(vk'*A(k:m,k:n)); % 3. update A
    V{k} = vk; % store vk in a cell array
end
R = A;
```

How much time will it take this algorithm to run for a given  $m$  and  $n$ ? An important aspect of numerical analysis is the determination of the *computational complexity* of a given algorithm. The first step, the only one considered in this course, involves counting floating point operations. (More subtle aspects of this craft include the analysis of how efficiently data is accessed through a computer's memory hierarchy, and how effectively the algorithm may be implemented on a parallel computer. Topics of this sort are addressed in CAAM 420 and 520.)

First we count the number of operations for the lines numbered 1, 2, and 3 in the `householder_qr` MATLAB code above. We let  $\ell = m - k + 1$  and  $p = n - k + 1$ . We assume that all basic arithmetic operations (add, subtract, multiply, divide, and square root) are each accomplished in the same amount of time. (In practice, the square root and sometimes division require considerably more time than the other operations.)

1. one 2-norm ( $2\ell$  operations), one addition
2. one 2-norm ( $2\ell$  operations),  $\ell$  divisions
3.  $\ell$  scalar multiplications (for  $2*vk$ )  
 $p$  inner products of length  $\ell$  vectors (for  $vk'*A(k:m,k:n)$ ): ( $2\ell p - p$  operations)  
 one outer product update (the subtraction): ( $2\ell p$  operations for one multiply, add per entry)

Adding these up, we see that each pass of the loop requires  $4\ell p + 6\ell - p + 1$  operations. (Recall that  $\ell = m - k + 1$  and  $p = n - k + 1$ .) The  $4\ell p$  term will dominate. Summing over  $k = 1, \dots, n$ , we find the number of floating point operations required, to leading order, is

$$\sum_{k=1}^n 4\ell p = \sum_{k=1}^n 4(m-k+1)(n-k+1) \approx 2mn^2 - \frac{2n^3}{3}.$$

### 1.2.6. The matrix $\mathbf{Q}$ .

Note that the `householder_qr` program does not compute the  $m \times m$  matrix  $\mathbf{Q}$ . This is because one does not typically need (or want) compute the matrix  $\mathbf{Q}$  explicitly. Instead, one simply saves the vectors  $\mathbf{v}_k$  at each step; from these vectors, one can compute products of the form  $\mathbf{Q}\mathbf{x}$  or  $\mathbf{Q}^*\mathbf{b}$  more efficiently than if  $\mathbf{Q}$  was itself computed.

For example, the following MATLAB code will compute  $\mathbf{Q}^*\mathbf{b}$ , using the cell array  $\mathbf{V}$  computed by `householder_qr`. (This is Algorithm 10.2 in Trefethen and Bau, page 74.) We apply  $\mathbf{Q}^* = \mathbf{Q}_n\mathbf{Q}_{n-1}\cdots\mathbf{Q}_1$  (using the notation from the last lecture) one Householder reflector at a time. Since the upper-left  $(k-1) \times (k-1)$  block of  $\mathbf{Q}_k \in \mathbb{C}^{m \times m}$  is the identity matrix, we need only work with entries  $k$  through  $m$  of  $\mathbf{b}$  (that is,  $\mathbf{b}(k:m)$ ) at step  $k$ .

```
for k=1:min(m-1,n)
    b(k:m) = b(k:m) - 2*V{k}*(V{k}'*b(k:m));
end
```

How would this algorithm change if we wanted to work with  $\mathbf{Q}$  instead of  $\mathbf{Q}^*$ ?

In many applications the matrix  $\mathbf{Q}$  is actually quite useful. Suppose that  $r_{jj} \neq 0$  for  $j = 1, \dots, n$ . Then  $\mathbf{A}$  is *full rank*, i.e., all its columns are linearly independent. In this case, the first  $n$  columns of  $\mathbf{Q} \in \mathbb{C}^{m \times m}$  form an orthonormal basis for  $\text{Ran}(\mathbf{A})$ . One could form  $\mathbf{Q}$  explicitly from the Householder reflectors; an alternative, as we shall see, is to bypass the  $m$ -by- $m$  matrix  $\mathbf{Q}$ , computing the leading  $n$  columns of  $\mathbf{Q}$  directly. The result is called a ‘skinny QR factorization’.

### 1.2.7. QR Decomposition via the Gram–Schmidt Algorithm.

While the Householder QR algorithm described in the last lecture is a novel idea for many students, there is another way to obtain a QR factorization that is probably more familiar, though perhaps you have never seen it written down in QR-style: the Gram–Schmidt algorithm for computing an orthonormal basis for a set of vectors.

Given a set of linearly independent vectors  $\mathbf{a}_1, \dots, \mathbf{a}_n \in \mathbb{C}^m$  for  $m \geq n$ , the Gram–Schmidt process constructs an orthonormal basis  $\{\mathbf{q}_1, \dots, \mathbf{q}_n\}$  for  $\text{span}\{\mathbf{a}_1, \dots, \mathbf{a}_n\}$  as follows:

$$\begin{aligned}\mathbf{q}_1 &= \mathbf{a}_1 / \|\mathbf{a}_1\|_2 \\ \hat{\mathbf{q}}_2 &= (\mathbf{I} - \mathbf{q}_1\mathbf{q}_1^*)\mathbf{a}_2 \\ \mathbf{q}_2 &= \hat{\mathbf{q}}_2 / \|\hat{\mathbf{q}}_2\|_2 \\ \hat{\mathbf{q}}_3 &= (\mathbf{I} - \mathbf{q}_1\mathbf{q}_1^* - \mathbf{q}_2\mathbf{q}_2^*)\mathbf{a}_3 \\ \mathbf{q}_3 &= \hat{\mathbf{q}}_3 / \|\hat{\mathbf{q}}_3\|_2 \\ &\vdots\end{aligned}$$

Note that since  $\|\mathbf{q}_j\|_2 = 1$ , the matrix  $\mathbf{q}_j\mathbf{q}_j^*$  is an orthogonal projector. So too is

$$\mathbf{I} - \mathbf{q}_1\mathbf{q}_1^* - \cdots - \mathbf{q}_k\mathbf{q}_k^*;$$

it projects onto the orthogonal complement of  $\text{span}\{\mathbf{q}_1, \dots, \mathbf{q}_k\}$ . Hence, the Gram–Schmidt algorithm constructs a new vector  $\mathbf{q}_{k+1}$  by projecting  $\mathbf{a}_{k+1}$  onto the orthogonal complement of the previous basis vectors, and then normalizing.

Can you spot the underlying QR factorization? The notation is suggestive. Let

$$\mathbf{A} = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_n]$$

and

$$\mathbf{Q} = [\mathbf{q}_1 \ \mathbf{q}_2 \ \cdots \ \mathbf{q}_n]$$

and define  $\mathbf{R}$  entrywise as

$$r_{j,k} = \begin{cases} \mathbf{q}_j^* \mathbf{a}_k, & j < k; \\ \|\widehat{\mathbf{q}}_j\|_2, & j = k; \\ 0, & j > k. \end{cases}$$

Thus the Gram–Schmidt factorization computes  $\mathbf{A} = \mathbf{QR}$ , with  $\mathbf{Q} \in \mathbb{C}^{m \times n}$  and  $\mathbf{R} \in \mathbb{C}^{n \times n}$ ; the columns of  $\mathbf{Q}$  form an orthonormal basis for  $\text{Ran}(\mathbf{A})$ . When  $m \neq n$ , this matrix  $\mathbf{Q}$  is not square, and hence it is not unitary. Since the columns of  $\mathbf{Q}$  are orthonormal, we have  $\mathbf{Q}^* \mathbf{Q} = \mathbf{I} \in \mathbb{C}^{n \times n}$ , but  $\mathbf{Q} \mathbf{Q}^* \neq \mathbf{I} \in \mathbb{C}^{m \times m}$ . (Note the important difference between this  $\mathbf{Q}$  and the square, unitary matrix  $\mathbf{Q}$  one obtains from the Householder QR algorithm, for which  $\mathbf{Q}^* \mathbf{Q} = \mathbf{Q} \mathbf{Q}^* = \mathbf{I} \in \mathbb{C}^{m \times m}$ .)

The MATLAB implementation suggested by the above construction is known as the *classical Gram–Schmidt algorithm*.

```
function [Q,R] = cgs_qr(A)
% Computation of the "skinny" QR decomposition of an m-by-n matrix
% (m>=n) using the the Classical Gram-Schmidt algorithm.
% See Trefethen and Bau, Algorithm 8.1

[m,n] = size(A);
if m<n, fprintf('ERROR: A should be an m-by-n matrix with m >= n.\n'); end

Q = zeros(m,n);
R = zeros(n,n);
for k=1:n
    Q(:,k) = A(:,k);
    for j=1:k-1
        R(j,k) = Q(:,j)'*A(:,k);
        Q(:,k) = Q(:,k) - R(j,k)*Q(:,j);
    end
    R(k,k) = norm(Q(:,k));
    Q(:,k) = Q(:,k)/R(k,k);
end
```

As we shall see in class, this implementation turns out to have unfortunate numerical properties: the vectors  $\mathbf{q}_1, \dots, \mathbf{q}_n$  constructed as the columns of  $\mathbf{Q}$  can actually be far from orthogonal, due to rounding errors in finite precision arithmetic. Notice that the classical Gram–Schmidt algorithm forms

$$\widehat{\mathbf{q}}_k = \mathbf{a}_k - r_{1,k} \mathbf{q}_1 - \cdots - r_{k-1,k} \mathbf{q}_{k-1},$$

where

$$r_{j,k} = \mathbf{q}_j^* \mathbf{a}_k. \quad (1)$$

Effectively, we obtain  $\widehat{\mathbf{q}}_k$  by subtracting from  $\mathbf{q}_k$  all the orthogonal projections of  $\mathbf{a}_k$  along the directions  $\mathbf{q}_1, \dots, \mathbf{q}_{k-1}$ . Notice, however, that for all  $j = 1, \dots, k-1$ ,

$$\mathbf{q}_j^* \mathbf{a}_k = \mathbf{q}_j^* (\mathbf{a}_k - r_{1,k} \mathbf{q}_1 - \dots - r_{j-1,k} \mathbf{q}_{j-1})$$

since  $\mathbf{q}_j^* \mathbf{q}_\ell = 0$  for all  $\ell = 1, \dots, j-1$ . Thus, we could have instead computed

$$r_{j,k} = \mathbf{q}_j^* (\mathbf{a}_k - r_{1,k} \mathbf{q}_1 - \dots - r_{j-1,k} \mathbf{q}_{j-1}). \quad (2)$$

The formulation (2) might look like more work than (1) at first, but we can arrange the computations efficiently, according to the following Modified Gram–Schmidt algorithm.

```

for  $j = 1, \dots, n$ 
   $\widehat{\mathbf{q}}_j := \mathbf{a}_j$ 
end

for  $j = 1, \dots, n$ 
   $r_{j,j} := \|\widehat{\mathbf{q}}_j\|_2$ 
  for  $k = j + 1, \dots, n$ 
     $r_{j,k} := \mathbf{q}_j^* \widehat{\mathbf{q}}_k$ 
     $\widehat{\mathbf{q}}_k := \widehat{\mathbf{q}}_k - r_{j,k} \mathbf{q}_j$ 
  end
end
end
```

Why should this formulation be superior? Here is a heuristic explanation: We compute  $\widehat{\mathbf{q}}_k$  by successively subtracting off terms like  $r_{j,k} \mathbf{q}_j$ . Small computer arithmetic errors will inevitably be made in the formation of  $r_{j,k}$  and  $\mathbf{q}_j$ . Since the Modified Gram–Schmidt algorithm computes  $r_{j,k}$  from the formula (2), the coefficient  $r_{j,k}$  it computes will adjust to these errors, in some sense, attempting to ‘project out the errors’ at each step.

An illustrative example should clarify this situation.<sup>†</sup> Let

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ \varepsilon & 0 & 0 \\ 0 & \varepsilon & 0 \\ 0 & 0 & \varepsilon \end{bmatrix} \in \mathbb{R}^{4 \times 3},$$

where  $\varepsilon \in \mathbb{R}$  is some number sufficiently small that computer arithmetic will round  $1 + \varepsilon^2$  down to 1. (See Lecture 8 for further details on such arithmetic systems.) In MATLAB,  $\varepsilon \leq 10^{-8}$  will suffice.

We first apply the Classical Gram–Schmidt algorithm, the steps of which we summarize below.

$$\widehat{\mathbf{q}}_1 = \begin{bmatrix} 1 \\ \varepsilon \\ 0 \\ 0 \end{bmatrix}, \quad r_{1,1} = 1 \text{ (rounded)}, \quad \mathbf{q}_1 = \begin{bmatrix} 1 \\ \varepsilon \\ 0 \\ 0 \end{bmatrix}$$

<sup>†</sup>This is Problem/Solution 18.9 in N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996. Higham attributes the example to Björck (1967).

$$r_{1,2} = \mathbf{q}_1^* \mathbf{a}_2 = 1, \quad \hat{\mathbf{q}}_2 = \begin{bmatrix} 1 \\ 0 \\ \varepsilon \\ 0 \end{bmatrix} - 1 \begin{bmatrix} 1 \\ \varepsilon \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -\varepsilon \\ \varepsilon \\ 0 \end{bmatrix}, \quad r_{2,2} = \varepsilon\sqrt{2}$$

$$\mathbf{q}_2 = \begin{bmatrix} 0 \\ -1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{bmatrix}$$

$$r_{1,3} = \mathbf{q}_1^* \mathbf{a}_3 = 1, \quad r_{2,3} = \mathbf{q}_2^* \mathbf{a}_3 = 0, \quad \hat{\mathbf{q}}_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \varepsilon \end{bmatrix} - 1 \begin{bmatrix} 1 \\ \varepsilon \\ 0 \\ 0 \end{bmatrix} - 0 \begin{bmatrix} 1 \\ 0 \\ \varepsilon \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -\varepsilon \\ 0 \\ \varepsilon \end{bmatrix}, \quad r_{3,3} = \varepsilon\sqrt{2}$$

$$\mathbf{q}_3 = \begin{bmatrix} 0 \\ -1/\sqrt{2} \\ 0 \\ 1/\sqrt{2} \end{bmatrix}.$$

But note that  $\mathbf{q}_2$  and  $\mathbf{q}_3$  are far from being orthogonal!

Now repeat the same calculation with Modified Gram–Schmidt. The first two steps, leading to  $\mathbf{q}_1$  and  $\mathbf{q}_2$ , are identical, but now to form  $\mathbf{q}_3$  we have the following operations:

$$r_{1,3} = \mathbf{q}_1^* \mathbf{a}_3 = 1, \quad r_{2,3} = \mathbf{q}_2^* (\mathbf{a}_3 - r_{1,3} \mathbf{q}_1) = [0 \quad -1/\sqrt{2} \quad 1/\sqrt{2} \quad 0] \begin{bmatrix} 0 \\ -\varepsilon \\ 0 \\ \varepsilon \end{bmatrix} = \varepsilon/\sqrt{2}.$$

The coefficient  $r_{1,3}$  is the same as for Classical Gram–Schmidt, but now  $r_{2,3} = \varepsilon/\sqrt{2}$  instead of  $r_{2,3} = 0$ . This modest change makes all the difference, for now we compute

$$\hat{\mathbf{q}}_3 = (\mathbf{a}_3 - r_{1,3} \mathbf{q}_1) - r_{2,3} \mathbf{q}_2 = \begin{bmatrix} 0 \\ -\varepsilon \\ 0 \\ \varepsilon \end{bmatrix} - \frac{\varepsilon}{\sqrt{2}} \begin{bmatrix} 0 \\ -1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -\varepsilon/2 \\ -\varepsilon/2 \\ \varepsilon \end{bmatrix},$$

so  $r_{3,3} = \sqrt{3/2}$  and hence

$$\mathbf{q}_3 = \begin{bmatrix} 0 \\ -1/\sqrt{6} \\ -1/\sqrt{6} \\ \sqrt{2/3} \end{bmatrix}.$$

This vector is entirely different from vector  $\mathbf{q}_3$  computed by the Classical Gram–Schmidt process.

Note that the  $\mathbf{q}_1$ ,  $\mathbf{q}_2$ , and  $\mathbf{q}_3$  vectors computed by the Modified Gram–Schmidt process are nearly orthogonal – the best we can hope for in computer arithmetic.

The following MATLAB code implements Modified Gram–Schmidt orthogonalization.

```
function [Q,R] = mgs_qr(A)
% Computation of the "skinny" QR decomposition of an m-by-n matrix
% (m>=n) using the the Modified Gram-Schmidt algorithm.
% See Trefethen and Bau, Algorithm 8.1

[m,n] = size(A);
if m<n, fprintf('ERROR: A should be an m-by-n matrix with m >= n.\n'); end
Q = zeros(m,n);
R = zeros(n,n);
Q = A;
for j=1:n
    R(j,j) = norm(Q(:,j));
    Q(:,j) = Q(:,j)/R(j,j);
    for k=j+1:n
        R(j,k) = Q(:,j)'*Q(:,k);
        Q(:,k) = Q(:,k) - R(j,k)*Q(:,j);
    end
end
end
```

### 1.2.8. QR for rank-deficient matrices.

What if the columns of  $\mathbf{A}$  are not linearly independent? In this case, the standard QR algorithm can fail. For example, the Gram–Schmidt orthogonalization process will eventually construct  $\hat{\mathbf{q}}_k = \mathbf{0}$  for some  $k \leq n$ , giving an error for  $\mathbf{q}_k = \hat{\mathbf{q}}_k / \|\hat{\mathbf{q}}_k\|_2$ .

If one knows the rank of  $\mathbf{A}$ , say  $\text{rank}(\mathbf{A}) = \dim(\text{Ran } \mathbf{A}) = r$ , then one can swap the columns of  $\mathbf{A}$  to allow a factorization of the form

$$\mathbf{Q}^* \mathbf{A} \mathbf{\Pi} = \begin{pmatrix} \mathbf{R}_{11} & \mathbf{R}_{12} \\ \mathbf{0} & \mathbf{0} \end{pmatrix},$$

where  $\mathbf{\Pi} \in \mathbb{C}^{n \times n}$  is a permutation matrix<sup>†</sup> that affects the column swapping and  $\mathbf{R}_{11} \in \mathbb{C}^{r \times r}$  is an upper triangular matrix with all diagonal entries nonzero. For details and an explanation of how to implement such a factorization using Householder reflectors, see Section 5.4.1 of G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins, Baltimore, 1996. In MATLAB, one can obtain a column-pivoted QR factorization by calling the `qr` routine as

$$[\mathbf{Q}, \mathbf{R}, \mathbf{Pi}] = \text{qr}(\mathbf{A});$$

which computes the factorization  $\mathbf{A} \mathbf{\Pi} = \mathbf{Q} \mathbf{R}$ .

---

<sup>†</sup>A permutation matrix is a square matrix in which each row and each column has exactly one entry equal to one, with all other entries equal to zero.