Contents

Lecture 7. QR Factorization

One algorithmic idea in numerical linear algebra is more important than all the others: QR factorization.

Reduced QR Factorization

For many applications, we find ourselves interested in the column spaces of a matrix A. Note the plural: these are the *successive* spaces spanned by the columns $a_1, a_2, \ldots$ of A:

$$\langle a_1 \rangle \;\subseteq\; \langle a_1, a_2 \rangle \;\subseteq\; \langle a_1, a_2, a_3 \rangle \;\subseteq\; \ldots.$$

Here, as in Lecture 5 and throughout the book, the notation $\langle \cdots \rangle$ indicates the subspace spanned by whatever vectors are included in the brackets. Thus $\langle a_1 \rangle$ is the one-dimensional space spanned by a_1, $\langle a_1, a_2 \rangle$ is the two-dimensional space spanned by a_1 and a_2, and so on. The idea of QR factorization is the construction of a sequence of orthonormal vectors $q_1, q_2, \ldots$ that span these successive spaces.

To be precise, assume for the moment that $A \in \mathbb{C}^{m\times n}$ $(m \geq n)$ has full rank n. We want the sequence $q_1, q_2, \ldots$ to have the property

$$\langle q_1, q_2, \ldots, q_j \rangle = \langle a_1, a_2, \ldots, a_j \rangle, \qquad j = 1, \ldots, n. \tag{7.1}$$

From the observations of Lecture 1, it is not hard to see that this amounts to the condition

$$\left[\begin{array}{c|c|c|c} & & & \\ a_1 & a_2 & \cdots & a_n \\ & & & \end{array}\right] = \left[\begin{array}{c|c|c|c} & & & \\ q_1 & q_2 & \cdots & q_n \\ & & & \end{array}\right] \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & & \vdots \\ & & \ddots & \\ & & & r_{nn} \end{bmatrix}, \tag{7.2}$$

where the diagonal entries r_{kk} are nonzero—for if (7.2) holds, then $a_1, \dots, a_k$ can be expressed as linear combinations of $q_1, \dots, q_k$, and the invertibility of the upper-left $k \times k$ block of the triangular matrix implies that, conversely, $q_1, \dots, q_k$ can be expressed as linear combinations of $a_1, \dots, a_k$. Written out, these equations take the form

$$\begin{aligned} a_1 &= r_{11}q_1, \\ a_2 &= r_{12}q_1 + r_{22}q_2, \\ a_3 &= r_{13}q_1 + r_{23}q_2 + r_{33}q_3, \\ &\vdots \\ a_n &= r_{1n}q_1 + r_{2n}q_2 + \cdots + r_{nn}q_n. \end{aligned} \tag{7.3}$$

As a matrix formula, we have

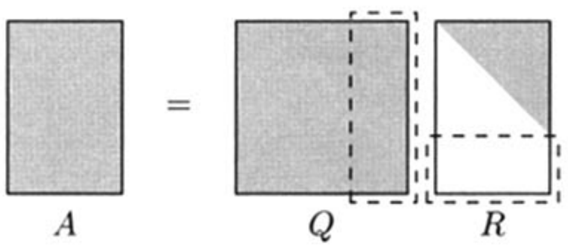
$$A = \hat{Q}\hat{R}, \tag{7.4}$$

where $\hat{Q}$ is $m \times n$ with orthonormal columns and $\hat{R}$ is $n \times n$ and upper-triangular. Such a factorization is called a *reduced QR factorization of A.*

Full QR Factorization

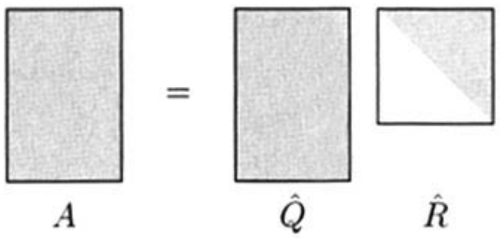
A *full QR factorization* of $A \in \mathbb{C}^{m\times n}$ $(m \geq n)$ goes further, appending an additional $m - n$ orthonormal columns to $\hat{Q}$ so that it becomes an $m \times m$ unitary matrix Q. This is analogous to the passage from the reduced to the full SVD described in Lecture 4. In the process, rows of zeros are appended to $\hat{R}$ so that it becomes an $m \times n$ matrix R, still upper-triangular. The relationship between the full and reduced QR factorizations is as follows.

Full QR Factorization $(m \geq n)$



In the full QR factorization, Q is $m \times m$, R is $m \times n$, and the last $m-n$ columns of Q are multiplied by zeros in R (enclosed by dashes). In the reduced QR factorization, the silent columns and rows are removed. Now $\hat{Q}$ is $m \times n$, $\hat{R}$ is $n \times n$, and none of the rows of $\hat{R}$ are necessarily zero.

Reduced QR Factorization ($m \geq n$)



Notice that in the full QR factorization, the columns q_j for $j > n$ are orthogonal to range(A). Assuming A is of full rank n, they constitute an orthonormal basis for range$(A)^{\perp}$ (the space orthogonal to range(A)), or equivalently, for null(A^*).

Gram–Schmidt Orthogonalization

Equations (7.3) suggest a method for computing reduced QR factorizations. Given $a_1, a_2, \ldots,$ we can construct the vectors $q_1, q_2, \ldots$ and entries r_{ij} by a process of successive orthogonalization. This is an old idea, known as *Gram–Schmidt orthogonalization*.

The process works like this. At the jth step, we wish to find a unit vector $q_j \in \langle a_1, \ldots, a_j \rangle$ that is orthogonal to $q_1, \ldots, q_{j-1}$. As it happens, we have already considered the necessary orthogonalization technique in (2.6). From that equation, we see that

$$v_j = a_j - (q_1^* a_j)q_1 - (q_2^* a_j)q_2 - \cdots - (q_{j-1}^* a_j)q_{j-1} \tag{7.5}$$

is a vector of the kind required, except that it is not yet normalized. If we divide by $\|v_j\|_2$, the result is a suitable vector q_j.

With this in mind, let us rewrite (7.3) in the form

$$\begin{aligned} q_1 &= \frac{a_1}{r_{11}}, \\ q_2 &= \frac{a_2 - r_{12} q_1}{r_{22}}, \\ q_3 &= \frac{a_3 - r_{13} q_1 - r_{23} q_2}{r_{33}}, \\ &\vdots \\ q_n &= \frac{a_n - \sum_{i=1}^{n-1} r_{in} q_i}{r_{nn}}. \end{aligned} \tag{7.6}$$

From (7.5) it is evident that an appropriate definition for the coefficients r_{ij} in the numerators of (7.6) is

$$r_{ij} = q_i^* a_j \qquad (i \neq j). \tag{7.7}$$

The coefficients r_{jj} in the denominators are chosen for normalization:

$$|r_{jj}| = \left\| a_j - \sum_{i=1}^{j-1} r_{ij} q_i \right\|_2. \tag{7.8}$$

Note that the sign of r_{jj} is not determined. Arbitrarily, we may choose $r_{jj} > 0$, in which case we shall finish with a factorization $A = \hat{Q}\hat{R}$ in which $\hat{R}$ has positive entries along the diagonal.

The algorithm embodied in (7.6)–(7.8) is the Gram–Schmidt iteration. Mathematically, it offers a simple route to understanding and proving various properties of QR factorizations. Numerically, it turns out to be unstable because of rounding errors on a computer. To emphasize the instability, numerical analysts refer to this as the *classical Gram–Schmidt iteration*, as opposed to the *modified Gram–Schmidt iteration*, discussed in the next lecture.

Algorithm 7.1. Classical Gram–Schmidt (unstable)

```
for $j = 1$ to $n$
    $v_j = a_j$
    for $i = 1$ to $j - 1$
        $r_{ij} = q_i^* a_j$
        $v_j = v_j - r_{ij} q_i$
    $r_{jj} = \|v_j\|_2$
    $q_j = v_j / r_{jj}$
```

Existence and Uniqueness

All matrices have QR factorizations, and under suitable restrictions, they are unique. We state first the existence result.

Theorem 7.1. *Every $A \in \mathbb{C}^{m \times n}$ $(m \geq n)$ has a full QR factorization, hence also a reduced QR factorization.*

Proof. Suppose first that A has full rank and that we want just a reduced QR factorization. In this case, a proof of existence is provided by the Gram–Schmidt algorithm itself. By construction, this process generates orthonormal columns of $\hat{Q}$ and entries of $\hat{R}$ such that (7.4) holds. Failure can occur only if at some step, v_j is zero and thus cannot be normalized to produce q_j.

However, this would imply $a_j \in \langle q_1, \dots, q_{j-1}\rangle = \langle a_1, \dots, a_{j-1}\rangle$, contradicting the assumption that A has full rank.

Now suppose that A does not have full rank. Then at one or more steps j, we shall find that (7.5) gives $v_j = 0$, as just mentioned. At this moment, we simply pick q_j arbitrarily to be any normalized vector orthogonal to $\langle q_1, \dots, q_{j-1}\rangle$, and then continue the Gram–Schmidt process.

Finally, the full, rather than reduced, QR factorization of an $m \times n$ matrix with $m > n$ can be constructed by introducing arbitrary orthonormal vectors in the same fashion. We follow the Gram–Schmidt process through step n, then continue on an additional $m - n$ steps, introducing vectors q_j at each step.

The issues discussed in the last two paragraphs came up already in Lecture 4, in our discussion of the SVD. □

We turn now to uniqueness. Suppose $A = \hat{Q}\hat{R}$ is a reduced QR factorization. If the ith column of $\hat{Q}$ is multiplied by z and the ith row of $\hat{R}$ is multiplied by z^{-1} for some scalar z with $|z| = 1$, we obtain another QR factorization of A. The next theorem asserts that if A has full rank, this is the only way to obtain distinct reduced QR factorizations.

Theorem 7.2. *Each $A \in \mathbb{C}^{m\times n}$ $(m \geq n)$ of full rank has a unique reduced QR factorization $A = \hat{Q}\hat{R}$ with $r_{jj} > 0$.*

Proof. Again, the proof is provided by the Gram–Schmidt iteration. From (7.4), the orthonormality of the columns of $\hat{Q}$, and the upper-triangularity of $\hat{R}$, it follows that any reduced QR factorization of A must satisfy (7.6)–(7.8). By the assumption of full rank, the denominators (7.8) of (7.6) are nonzero, and thus at each successive step j, these formulas determine r_{ij} and q_j fully, except in one place: the sign of r_{jj}, not specified in (7.8). Once this is fixed by the condition $r_{jj} > 0$, as in Algorithm 7.1, the factorization is completely determined. □

When Vectors Become Continuous Functions

The QR factorization has an analogue for orthonormal expansions of functions rather than vectors.

Suppose we replace $\mathbb{C}^m$ by $L^2[-1,1]$, a vector space of complex-valued functions on $[-1,1]$. We shall not introduce the properties of this space formally; suffice it to say that the inner product of f and g now takes the form

$$(f,g) = \int_{-1}^{1} \overline{f(x)}\, g(x)\, dx. \tag{7.9}$$

Consider, for example, the following "matrix" whose "columns" are the monomials x^j:

$$A = \left[\begin{array}{c|c|c|c|c} & & & & \\ 1 & x & x^2 & \cdots & x^{n-1} \\ & & & & \end{array}\right]. \tag{7.10}$$

Each column is a function in $L^2[-1,1]$, and thus, whereas A is discrete as usual in the horizontal direction, it is continuous in the vertical direction. It is a continuous analogue of the Vandermonde matrix (1.4) of Example 1.1.

The "continuous QR factorization" of A takes the form

$$A = QR = \left[\begin{array}{c|c|c|c} & & & \\ q_0(x) & q_1(x) & \cdots & q_{n-1}(x) \\ & & & \end{array}\right] \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & & \vdots \\ & & \ddots & \\ & & & r_{nn} \end{bmatrix},$$

where the columns of Q are functions of x, orthonormal with respect to the inner product (7.9):

$$\int_{-1}^{1} \overline{q_i(x)}\, q_j(x)\, dx \;=\; \delta_{ij} \;=\; \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

From the Gram–Schmidt construction we can see that q_j is a polynomial of degree j. These polynomials are scalar multiples of what are known as the *Legendre polynomials*, P_j, which are conventionally normalized so that $P_j(1) = 1$. The first few P_j are

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \tfrac{3}{2}x^2 - \tfrac{1}{2}, \quad P_3(x) = \tfrac{5}{2}x^3 - \tfrac{3}{2}x; \tag{7.11}$$

see Figure 7.1. Like the monomials $1, x, x^2, \ldots$, this sequence of polynomials spans the spaces of polynomials of successively higher degree. However, $P_0(x), P_1(x), P_2(x), \ldots$ have the advantage that they are orthogonal, making them far better suited for certain computations. In fact, computations with such polynomials form the basis of *spectral methods*, one of the most powerful techniques for the numerical solution of partial differential equations.

What is the "projection matrix" $\hat{Q}\hat{Q}^*$ (6.6) associated with $\hat{Q}$? It is a "$[-1,1] \times [-1,1]$ matrix," that is, an integral operator

$$f(\cdot) \;\mapsto\; \sum_{j=0}^{n-1} q_j(\cdot) \int_{-1}^{1} \overline{q_j(x)}\, f(x)\, dx \tag{7.12}$$

mapping functions in $L^2[-1,1]$ to functions in $L^2[-1,1]$.

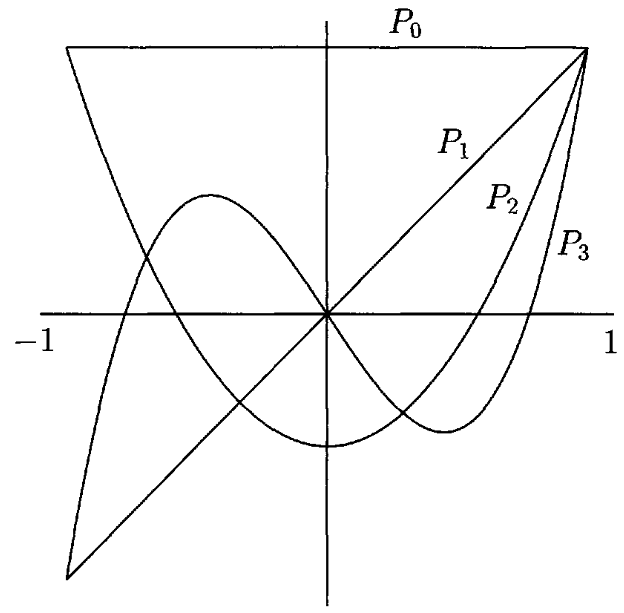


Figure 7.1. *The first four Legendre polynomials* (7.11). *Apart from scale factors, these can be interpreted as the columns of $\hat{Q}$ in a reduced QR factorization of the* "$[-1,1] \times 4$ *matrix*" $[1,\ x,\ x^2,\ x^3]$.

Solution of $Ax = b$ by QR Factorization

In closing this lecture we return for a moment to discrete, finite matrices. Suppose we wish to solve $Ax = b$ for x, where $A \in \mathbb{C}^{m \times m}$ is nonsingular. If $A = QR$ is a QR factorization, then we can write $QRx = b$, or

$$Rx = Q^* b. \tag{7.13}$$

The right-hand side of this equation is easy to compute, if Q is known, and the system of linear equations implicit in the left-hand side is also easy to solve because it is triangular. This suggests the following method for computing the solution to $Ax = b$:

1. Compute a QR factorization $A = QR$.
2. Compute $y = Q^* b$.
3. Solve $Rx = y$ for x.

In later lectures we shall present algorithms for each of these steps.

The combination 1–3 is an excellent method for solving linear systems of equations; in Lecture 16, we shall prove this. However, it is not the standard method for such problems. Gaussian elimination is the algorithm generally used in practice, since it requires only half as many numerical operations.

Exercises

7.1. Consider again the matrices A and B of Exercise 6.4.

(a) Using any method you like, determine (on paper) a reduced QR factorization $A = \hat{Q}\hat{R}$ and a full QR factorization $A = QR$.

(b) Again using any method you like, determine reduced and full QR factorizations $B = \hat{Q}\hat{R}$ and $B = QR$.

7.2. Let A be a matrix with the property that columns $1, 3, 5, 7, \ldots$ are orthogonal to columns $2, 4, 6, 8, \ldots$. In a reduced QR factorization $A = \hat{Q}\hat{R}$, what special structure does $\hat{R}$ possess? You may assume that A has full rank.

7.3. Let A be an $m \times m$ matrix, and let a_j be its jth column. Give an algebraic proof of *Hadamard's inequality*:

$$|\det A| \le \prod_{j=1}^{m} \|a_j\|_2.$$

Also give a geometric interpretation of this result, making use of the fact that the determinant equals the volume of a parallelepiped.

7.4. Let $x^{(1)}$, $y^{(1)}$, $x^{(2)}$, and $y^{(2)}$ be nonzero vectors in $\mathbb{R}^3$ with the property that $x^{(1)}$ and $y^{(1)}$ are linearly independent and so are $x^{(2)}$ and $y^{(2)}$. Consider the two planes in $\mathbb{R}^3$,

$$P^{(1)} = \langle x^{(1)}, y^{(1)} \rangle, \qquad P^{(2)} = \langle x^{(2)}, y^{(2)} \rangle.$$

Suppose we wish to find a nonzero vector $v \in \mathbb{R}^3$ that lies in the intersection $P = P^{(1)} \cap P^{(2)}$. Devise a method for solving this problem by reducing it to the computation of QR factorizations of three 3×2 matrices.

7.5. Let A be an $m \times n$ matrix $(m \ge n)$, and let $A = \hat{Q}\hat{R}$ be a reduced QR factorization.

(a) Show that A has rank n if and only if all the diagonal entries of $\hat{R}$ are nonzero.

(b) Suppose $\hat{R}$ has k nonzero diagonal entries for some k with $0 \le k < n$. What does this imply about the rank of A? Exactly k? At least k? At most k? Give a precise answer, and prove it.

Lecture 8. Gram–Schmidt Orthogonalization

The Gram–Schmidt iteration is the basis of one of the two principal numerical algorithms for computing QR factorizations. It is a process of "triangular orthogonalization," making the columns of a matrix orthonormal via a sequence of matrix operations that can be interpreted as multiplication on the right by upper-triangular matrices.

Gram–Schmidt Projections

In the last lecture we presented the Gram–Schmidt iteration in its classical form. To begin this lecture, we describe the same algorithm again in another way, using orthogonal projectors.

Let $A \in \mathbb{C}^{m\times n}$, $m \geq n$, be a matrix of full rank with columns $\{a_j\}$. Before, we expressed the Gram–Schmidt iteration by the formulas (7.6)–(7.8). Consider now the sequence of formulas

$$q_1 = \frac{P_1 a_1}{\|P_1 a_1\|}, \quad q_2 = \frac{P_2 a_2}{\|P_2 a_2\|}, \quad \ldots, \quad q_n = \frac{P_n a_n}{\|P_n a_n\|}. \tag{8.1}$$

In these formulas, each P_j denotes an orthogonal projector. Specifically, P_j is the $m \times m$ matrix of rank $m-(j-1)$ that projects $\mathbb{C}^m$ orthogonally onto the space orthogonal to $\langle q_1, \ldots, q_{j-1}\rangle$. (In the case $j = 1$, this prescription reduces to the identity: $P_1 = I$.) Now, observe that q_j as defined by (8.1) is orthogonal to $q_1, \ldots, q_{j-1}$, lies in the space $\langle a_1, \ldots, a_j \rangle$, and has norm 1. Thus we see that (8.1) is equivalent to (7.6)–(7.8) and hence to Algorithm 7.1.

The projector P_j can be represented explicitly. Let $\hat{Q}_{j-1}$ denote the $m \times (j-1)$ matrix containing the first $j-1$ columns of $\hat{Q}$,

$$\hat{Q}_{j-1} = \left[\begin{array}{c|c|c|c} & & & \\ q_1 & q_2 & \cdots & q_{j-1} \\ & & & \end{array} \right]. \tag{8.2}$$

Then P_j is given by

$$P_j = I - \hat{Q}_{j-1}\hat{Q}^*_{j-1}. \tag{8.3}$$

By now, the reader may be familiar enough with our notation and with orthogonality ideas to see at a glance that (8.3) represents the operator applied to a_j in (7.5).

Modified Gram–Schmidt Algorithm

In practice, the Gram–Schmidt formulas are not applied as we have indicated in Algorithm 7.1 and in (8.1), for this sequence of calculations turns out to be numerically unstable. Fortunately, there is a simple modification that improves matters. We have not discussed numerical stability yet; this will come in the next lecture and then systematically beginning in Lecture 14. For the moment, it is enough to know that a stable algorithm is one that is not too sensitive to the effects of rounding errors on a computer.

For each value of j, Algorithm 7.1 computes a single orthogonal projection of rank $m - (j-1)$,

$$v_j = P_j a_j. \tag{8.4}$$

In contrast, the modified Gram–Schmidt algorithm computes the same result by a sequence of $j-1$ projections of rank $m-1$. Recall from (6.9) that $P_{\perp q}$ denotes the rank $m-1$ orthogonal projector onto the space orthogonal to a nonzero vector $q \in \mathbb{C}^m$. By the definition of P_j, it is not difficult to see that

$$P_j = P_{\perp q_{j-1}} \cdots P_{\perp q_2} P_{\perp q_1}, \tag{8.5}$$

again with $P_1 = I$. Thus an equivalent statement to (8.4) is

$$v_j = P_{\perp q_{j-1}} \cdots P_{\perp q_2} P_{\perp q_1} a_j. \tag{8.6}$$

The modified Gram–Schmidt algorithm is based on the use of (8.6) instead of (8.4).

Mathematically, (8.6) and (8.4) are equivalent. However, the sequences of arithmetic operations implied by these formulas are different. The modified algorithm calculates v_j by evaluating the following formulas in order:

$$\begin{aligned}
v_j^{(1)} &= a_j, \\
v_j^{(2)} &= P_{\perp q_1} v_j^{(1)} &&= v_j^{(1)} - q_1 q_1^* v_j^{(1)}, \\
v_j^{(3)} &= P_{\perp q_2} v_j^{(2)} &&= v_j^{(2)} - q_2 q_2^* v_j^{(2)}, \\
&\vdots && \vdots \\
v_j = v_j^{(j)} &= P_{\perp q_{j-1}} v_j^{(j-1)} &&= v_j^{(j-1)} - q_{j-1} q_{j-1}^* v_j^{(j-1)}.
\end{aligned} \tag{8.7}$$

In finite precision computer arithmetic, we shall see that (8.7) introduces smaller errors than (8.4).

When the algorithm is implemented, the projector $P_{\perp q_i}$ can be conveniently applied to $v_j^{(i)}$ for each $j > i$ immediately after q_i is known. This is done in the description below.

Algorithm 8.1. Modified Gram–Schmidt

```
for $i = 1$ to $n$
    $v_i = a_i$
for $i = 1$ to $n$
    $r_{ii} = \|v_i\|$
    $q_i = v_i / r_{ii}$
    for $j = i + 1$ to $n$
        $r_{ij} = q_i^* v_j$
        $v_j = v_j - r_{ij} q_i$
```

In practice, it is common to let v_i overwrite a_i and q_i overwrite v_i in order to save storage.

The reader should compare Algorithms 7.1 and 8.1 until he or she is confident of their equivalence.

Operation Count

The Gram–Schmidt algorithm is the first algorithm we have presented in this book, and with any algorithm, it is important to assess its cost. To do so, throughout the book we follow the classical route and count the number of floating point operations—*"flops"*—that the algorithm requires. Each addition, subtraction, multiplication, division, or square root counts as one flop.

We make no distinction between real and complex arithmetic, although in practice on most computers there is a sizable difference.

In fact, there is much more to the cost of an algorithm than operation counts. On a single-processor computer, the execution time is affected by the movement of data between elements of the memory hierarchy and by competing jobs running on the same processor. On multiprocessor machines the situation becomes more complex, with communication between processors sometimes taking on an importance much greater than that of actual "computation." With some regret, we shall ignore these important considerations, because this book is deliberately classical in style, focusing on algorithmic foundations.

For both variants of the Gram Schmidt iteration, here is the classical result.

Theorem 8.1. *Algorithms* 7.1 *and* 8.1 *require* $\sim 2mn^2$ *flops to compute a QR factorization of an* $m \times n$ *matrix.*

Note that the theorem expresses only the leading term of the flop count. The symbol "$\sim$" has its usual asymptotic meaning:

$$\lim_{m,n\to\infty} \frac{\text{number of flops}}{2mn^2} = 1.$$

In discussing operation counts for algorithms, it is standard to discard lower-order terms as we have done here, since they are usually of little significance unless m and n are small.

Theorem 8.1 can be established as follows. To be definite, consider the modified Gram–Schmidt algorithm, Algorithm 8.1. When m and n are large, the work is dominated by the operations in the innermost loop:

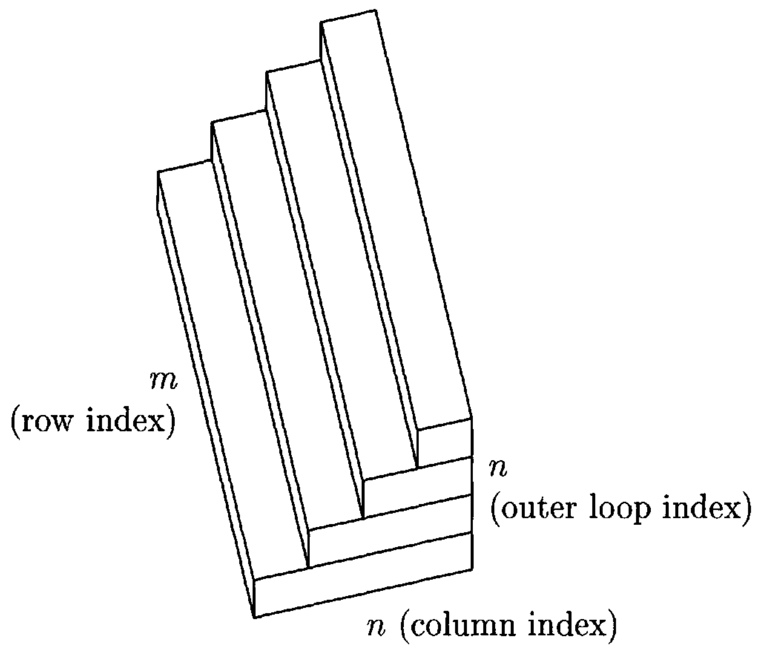
$$\begin{aligned} r_{ij} &= q_i^* v_j, \\ v_j &= v_j - r_{ij} q_i. \end{aligned}$$

The first line computes an inner product $q_i^* v_j$, requiring m multiplications and $m-1$ additions, and the second computes $v_j - r_{ij}q_i$, requiring m multiplications and m subtractions. The total work involved in a single inner iteration is consequently $\sim 4m$ flops, or 4 flops per column vector element. All together, the number of flops required by the algorithm is asymptotic to

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 4m \sim \sum_{i=1}^{n} (i) 4m \sim 2mn^2. \tag{8.8}$$

Counting Operations Geometrically

Operation counts can always be determined algebraically as in (8.8), and this is the standard procedure in the numerical analysis literature. However, it is also enlightening to take a different, geometrical route to the same conclusion. The argument goes like this. At the first step of the outer loop, Algorithm 8.1 operates on the whole matrix, subtracting a multiple of column 1 from the other columns. At the second step, it operates on a submatrix, subtracting a multiple of column 2 from columns $3, \ldots, n$. Continuing on in this way, at each step the column dimension shrinks by 1 until at the final step, only column n is modified. This process can be represented by the following diagram:



The $m \times n$ rectangle at the bottom corresponds to the first pass through the outer loop, the $m \times (n-1)$ rectangle above it to the second pass, and so on.

To leading order as $m, n \to \infty$, then, the operation count for Gram–Schmidt orthogonalization is proportional to the volume of the figure above. The constant of proportionality is four flops, because as noted above, the two steps of the inner loop correspond to four operations at each matrix location. Now as $m, n \to \infty$, the figure converges to a right triangular prism, with volume $mn^2/2$. Multiplying by four flops per unit volume gives, again,

$$\text{Work for Gram–Schmidt orthogonalization:} \quad \sim 2mn^2 \text{ flops.} \qquad (8.9)$$

In this book we generally record operation counts in the format (8.9), without stating them as theorems. We often derive these results via figures like the one above, although algebraic derivations are also possible. One reason we do this is that a figure of this kind, besides being a route to an operation count, also serves as a reminder of the structure of an algorithm. For pictures of algorithms with different structures, see pp. 75 and 176.

Gram–Schmidt as Triangular Orthogonalization

Each outer step of the modified Gram–Schmidt algorithm can be interpreted as a right-multiplication by a square upper-triangular matrix. For example, beginning with A, the first iteration multiplies the first column a_1 by $1/r_{11}$ and then subtracts r_{1j} times the result from each of the remaining columns a_j. This is equivalent to right-multiplication by a matrix R_1:

$$\left[\begin{array}{c|c|c|c} & & & \\ v_1 & v_2 & \cdots & v_n \\ & & & \end{array}\right] \begin{bmatrix} \frac{1}{r_{11}} & \frac{-r_{12}}{r_{11}} & \frac{-r_{13}}{r_{11}} & \cdots \\ & 1 & & \\ & & 1 & \\ & & & \ddots \end{bmatrix} = \left[\begin{array}{c|c|c|c} & & & \\ q_1 & v_2^{(2)} & \cdots & v_n^{(2)} \\ & & & \end{array}\right].$$

In general, step i of Algorithm 8.1 subtracts r_{ij}/r_{ii} times column i of the current A from columns $j > i$ and replaces column i by $1/r_{ii}$ times itself. This corresponds to multiplication by an upper-triangular matrix R_i:

$$R_2 = \begin{bmatrix} 1 & & & \\ & \frac{1}{r_{22}} & \frac{-r_{23}}{r_{22}} & \cdots \\ & & 1 & \\ & & & \ddots \end{bmatrix}, \quad R_3 = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \frac{1}{r_{33}} & \cdots \\ & & & \ddots \end{bmatrix}, \quad \ldots .$$

At the end of the iteration we have

$$A \underbrace{R_1 R_2 \cdots R_n}_{\hat{R}^{-1}} = \hat{Q}. \tag{8.10}$$

This formulation demonstrates that the Gram–Schmidt algorithm is a method of *triangular orthogonalization*. It applies triangular operations on the right of a matrix to reduce it to a matrix with orthonormal columns. Of course, in practice, we do not form the matrices R_i and multiply them together explicitly. The purpose of mentioning them is to give insight into the structure of the Gram–Schmidt algorithm. In Lecture 20 we shall see that it bears a close resemblance to the structure of Gaussian elimination.

Exercises

8.1. Let A be an $m \times n$ matrix. Determine the exact numbers of floating point additions, subtractions, multiplications, and divisions involved in computing the factorization $A = \hat{Q}\hat{R}$ by Algorithm 8.1.

8.2. Write a MATLAB function `[Q,R] = mgs(A)` (see next lecture) that computes a reduced QR factorization $A = \hat{Q}\hat{R}$ of an $m \times n$ matrix A with $m \geq n$ using modified Gram–Schmidt orthogonalization. The output variables are a matrix $Q \in \mathbb{C}^{m\times n}$ with orthonormal columns and a triangular matrix $R \in \mathbb{C}^{n\times n}$.

8.3. Each upper-triangular matrix R_j of p. 61 can be interpreted as the product of a diagonal matrix and a unit upper-triangular matrix (i.e., an upper-triangular matrix with 1 on the diagonal). Explain exactly what these factors are, and which line of Algorithm 8.1 corresponds to each.

Lecture 9. MATLAB

To learn numerical linear algebra, one must make a habit of experimenting on the computer. There is no better way to do this than by using the problem-solving environment known as MATLAB®.* In this lecture we illustrate MATLAB experimentation by three examples. Along the way, we make some observations about the stability of Gram–Schmidt orthogonalization.

MATLAB

MATLAB is a language for mathematical computations whose fundamental data types are vectors and matrices. It is distinguished from languages like Fortran and C by operating at a higher mathematical level, including hundreds of operations such as matrix inversion, the singular value decomposition, and the fast Fourier transform as built-in commands. It is also a problem-solving environment, processing top-level comments by an interpreter rather than a compiler and providing in-line access to 2D and 3D graphics.

Since the 1980s, MATLAB has become a widespread tool among numerical analysts and engineers around the world. For many problems of large-scale scientific computing, and for virtually all small- and medium-scale experimentation in numerical linear algebra, it is the language of choice.

*MATLAB is a registered trademark of The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, USA, tel. 508-647-7000, fax 508-647-7001, `info@mathworks.com`, `http://www.mathworks.com`.

In this book, we use MATLAB now and then to present certain numerical experiments, and in some exercises. We do not describe the language systematically, since the number of experiments we present is limited, and only a reading knowledge of MATLAB is needed to follow them.

Experiment 1: Discrete Legendre Polynomials

In Lecture 7 we considered the Vandermonde "matrix" with "columns" consisting of the monomials 1, x, x^2, and x^3 on the interval $[-1,1]$. Suppose we now make this a true Vandermonde matrix by discretizing $[-1,1]$ by 257 equally spaced points. The following lines of MATLAB construct this matrix and compute its reduced QR factorization.

```
x = (-128:128)'/128;        Set x to a discretization of [-1,1].
A=[x.^0 x.^1 x.^2 x.^3];    Construct Vandermonde matrix.
[Q,R] = qr(A,0);            Find its reduced QR factorization.
```

Here are a few remarks on these commands. In the first line, the prime ' converts (-128:128) from a row to a column vector. In the second line, the sequences .^ indicate *entrywise* powers. In the third line, qr is a built-in MATLAB function for computing QR factorizations; the argument 0 indicates that a reduced rather than full factorization is needed. The method used here is not Gram–Schmidt orthogonalization but Householder triangularization, discussed in the next lecture, but this is of no consequence for the present purpose. In all three lines, the semicolons at the end suppress the printed output that would otherwise be produced (x, A, Q, and R).

The columns of the matrix Q are essentially the first four Legendre polynomials of Figure 7.1. They differ slightly, by amounts close to plotting accuracy, because the continuous inner product on $[-1,1]$ that defines the Legendre polynomials has been replaced by a discrete analogue. They also differ in normalization, since a Legendre polynomial should satisfy $P_k(1) = 1$. We can fix this by dividing each column of Q by its final entry. The following lines of MATLAB do this by a right-multiplication by a 4×4 diagonal matrix.

```
scale = Q(257,:);         Select last row of Q.
Q = Q*diag(1 ./scale);    Rescale columns by these numbers.
plot(Q)                   Plot columns of rescaled Q.
```

The result of our computation is a plot that looks just like Figure 7.1 (not shown). In Fortran or C, this would have taken dozens of lines of code containing numerous loops and nested loops. In our six lines of MATLAB, not a single loop has appeared explicitly, though at least one loop is implicit in every line.

Experiment 2: Classical vs. Modified Gram–Schmidt

Our second example has more algorithmic substance. Its purpose is to explore the difference in numerical stability between the classical and modified Gram–Schmidt algorithms.

First, we construct a square matrix A with random singular vectors and widely varying singular values spaced by factors of 2 between 2^{-1} and 2^{-80}.

`[U,X] = qr(randn(80));`	Set U to a random orthogonal matrix.
`[V,X] = qr(randn(80));`	Set V to a random orthogonal matrix.
`S=diag(2.^(-1:-1:-80));`	Set S to a diagonal matrix with exponentially graded entries.
`A = U*S*V;`	Set A to a matrix with these entries as singular values.

Now, we use Algorithms 7.1 and 8.1 to compute QR factorizations of A. In the following code, the programs `clgs` and `mgs` are MATLAB implementations, not listed here, of Algorithms 7.1 and 8.1.

`[QC,RC] = clgs(A);`	Compute a factorization $Q^{(c)}R^{(c)}$ by classical Gram–Schmidt.
`[QM,RM] = mgs(A);`	Compute a factorization $Q^{(m)}R^{(m)}$ by modified Gram–Schmidt.

Finally, we plot the diagonal elements r_{jj} produced by both computations (MATLAB code not shown). Since $r_{jj} = \|P_j a_j\|$, this gives us a picture of the size of the projection at each step. The results are shown on a logarithmic scale in Figure 9.1.

The first thing one notices in the figure is a steady decrease of r_{jj} with j, closely matching the line 2^{-j}. Evidently r_{jj} is not exactly equal to the jth singular value of A, but it is a reasonably good approximation. This phenomenon can be roughly explained as follows. The SVD of A can be written in the form (5.3) as

$$A = 2^{-1}u_1v_1^* + 2^{-2}u_2v_2^* + 2^{-3}u_3v_3^* + \cdots + 2^{-80}u_{80}v_{80}^*,$$

where $\{u_j\}$ and $\{v_j\}$ are the left and right singular vectors of A, respectively. In particular, the jth column of A has the form

$$a_j = 2^{-1}\overline{v}_{j1}u_1 + 2^{-2}\overline{v}_{j2}u_2 + 2^{-3}\overline{v}_{j3}u_3 + \cdots + 2^{-80}\overline{v}_{j,80}u_{80}.$$

Since the singular vectors are random, we can expect that the numbers $\overline{v}_{ji}$ are all of a similar magnitude, on the order of $80^{-1/2} \approx 0.1$. Now, when we take the QR factorization, it is evident that the first vector q_1 is likely to be

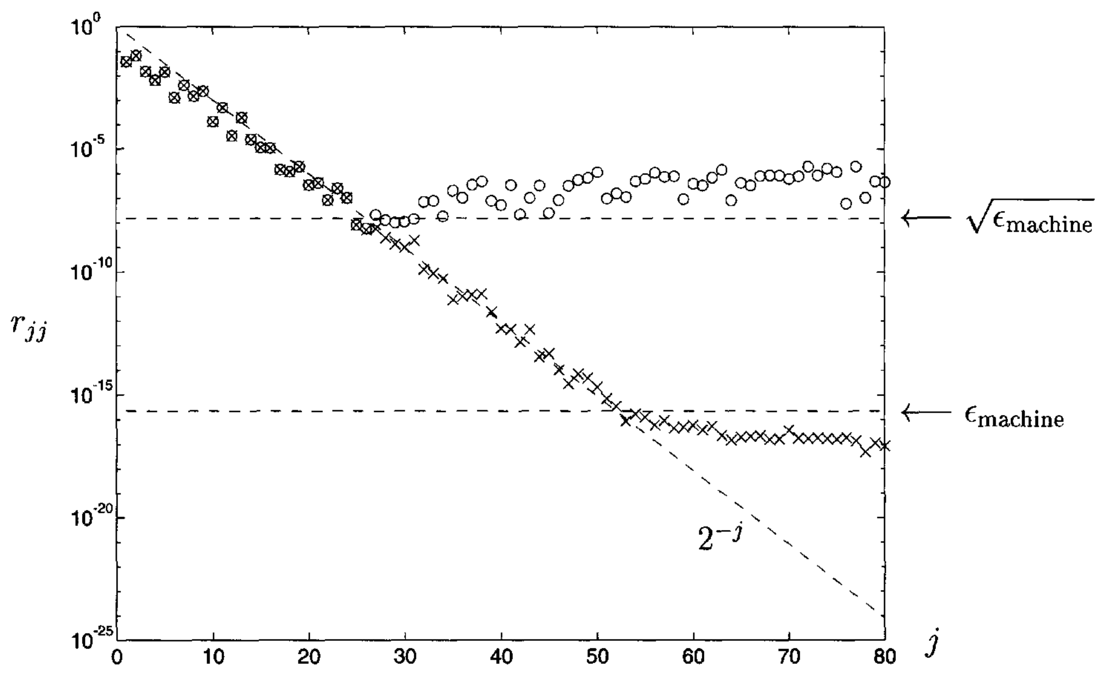


Figure 9.1. *Computed r_{jj} versus j for the QR factorization of a matrix with exponentially graded singular values. On this computer with about 16 digits of relative accuracy, the classical Gram–Schmidt algorithm produces the numbers represented by circles and the modified Gram–Schmidt algorithm produces the numbers represented by crosses.*

approximately equal to u_1, with r_{11} on the order of $2^{-1} \times 80^{-1/2}$. Orthogonalization at the next step will yield a second vector q_2 approximately equal to u_2, with r_{22} on the order of $2^{-2} \times 80^{-1/2}$—and so on.

The next thing one notices in Figure 9.1 is that the geometric decrease of r_{jj} does not continue all the way to $j = 80$. This is a consequence of rounding errors on the computer. With the classical Gram–Schmidt algorithm, the numbers never become smaller than about 10^{-8}. With the modified Gram–Schmidt algorithm, they shrink eight orders of magnitude further, down to the order of 10^{-16}, which is the level of *machine epsilon* for the computer used in this calculation. Machine epsilon is defined in Lecture 13.

Clearly, some algorithms are more stable than others. It is well established that the classical Gram–Schmidt process is one of the unstable ones. Consequently it is rarely used, except sometimes on parallel computers in situations where advantages related to communication may outweigh the disadvantage of instability.

Experiment 3: Numerical Loss of Orthogonality

At the risk of confusing the reader by presenting two instability phenomena in succession, we close this lecture by exhibiting another, different kind of instability that affects both the modified and classical Gram–Schmidt algorithms. In floating point arithmetic, these algorithms may produce vectors q_j that are far from orthogonal. The loss of orthogonality occurs when A is close to rank-deficient, and, like most instabilities, it can appear even in low dimensions.

Starting on paper rather than in MATLAB, consider the case of a matrix

$$A = \begin{bmatrix} 0.70000 & 0.70711 \\ 0.70001 & 0.70711 \end{bmatrix} \tag{9.1}$$

on a computer that rounds all computed results to five digits of relative accuracy (Lecture 13). The classical and modified algorithms are identical in the 2×2 case. At step $j = 1$, the first column is normalized, yielding

$$r_{11} = 0.98996, \qquad q_1 = a_1/r_{11} = \begin{bmatrix} 0.70000/0.98996 \\ 0.70001/0.98996 \end{bmatrix} = \begin{bmatrix} 0.70710 \\ 0.70711 \end{bmatrix}$$

in five-digit arithmetic. At step $j = 2$, the component of a_2 in the direction of q_1 is computed and subtracted out:

$$r_{12} = q_1^* a_2 = 0.70710 \times 0.70711 + 0.70711 \times 0.70711 = 1.0000,$$

$$v_2 = a_2 - r_{12} q_1 = \begin{bmatrix} 0.70711 \\ 0.70711 \end{bmatrix} - \begin{bmatrix} 0.70710 \\ 0.70711 \end{bmatrix} = \begin{bmatrix} 0.00001 \\ 0.00000 \end{bmatrix},$$

again with rounding to five digits. This computed v_2 is dominated by errors. The final computed Q is

$$Q = \begin{bmatrix} 0.70710 & 1.0000 \\ 0.70711 & 0.0000 \end{bmatrix},$$

which is not close to any orthogonal matrix.

On a computer with sixteen-digit precision, we still lose about five digits of orthogonality if we apply modified Gram–Schmidt to the matrix (9.1). Here is the MATLAB evidence. The "eye" function generates the identity of the indicated dimension.

```
A = [.70000 .70711; .70001 .70711];      Define A.
[Q,R] = qr(A);                           Compute factor Q by Householder.
   norm(Q'*Q-eye(2))                        Test orthogonality of Q.
[Q,R] = mgs(A);                          Compute factor Q by modified G–S.
   norm(Q'*Q-eye(2))                        Test orthogonality of Q.
```

The lines without semicolons produce the following printed output:

```
ans = 2.3515e-16,     ans = 2.3014e-11.
```

Exercises

9.1. (a) Run the six-line MATLAB program of Experiment 1 to produce a plot of approximate Legendre polynomials.

(b) For $k = 0, 1, 2, 3$, plot the difference on the 257-point grid between these approximations and the exact polynomials (7.11). How big are the errors, and how are they distributed?

(c) Compare these results with what you get with grid spacings $\Delta x = 2^{-\nu}$ for other values of ν. What power of Δx appears to control the convergence?

9.2. In Experiment 2, the singular values of A match the diagonal elements of a QR factor R approximately. Consider now a very different example. Suppose $Q = I$ and $A = R$, the $m \times m$ matrix (a *Toeplitz matrix*) with 1 on the main diagonal, 2 on the first superdiagonal, and 0 everywhere else.

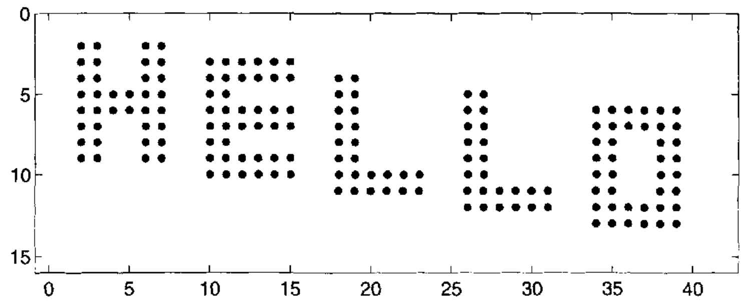
(a) What are the eigenvalues, determinant, and rank of A?

(b) What is A^{-1}?

(c) Give a nontrivial upper bound on σ_m, the mth singular value of A. You are welcome to use MATLAB for inspiration, but the bound you give should be justified analytically. (Hint: Use part (b).)

This problem illustrates that you cannot always infer much about the singular values of a matrix from its eigenvalues or from the diagonal entries of a QR factor R.

9.3. (a) Write a MATLAB program that sets up a 15×40 matrix with entries 0 everywhere except for the values 1 in the positions indicated in the picture below. The upper-leftmost 1 is in position $(2, 2)$, and the lower-rightmost 1 is in position $(13, 39)$. This picture was produced with the command `spy(A)`.



(b) Call `svd` to compute the singular values of A, and print the results. Plot these numbers using both `plot` and `semilogy`. What is the mathematically exact rank of A? How does this show up in the computed singular values?

(c) For each i from 1 to rank(A), construct the rank-i matrix B that is the best approximation to A in the 2-norm. Use the command `pcolor(B)` with `colormap(gray)` to create images of these various approximations.

Lecture 10. Householder Triangularization

The other principal method for computing QR factorizations is Householder triangularization, which is numerically more stable than Gram–Schmidt orthogonalization, though it lacks the latter's applicability as a basis for iterative methods. The Householder algorithm is a process of "orthogonal triangularization," making a matrix triangular by a sequence of unitary matrix operations.

Householder and Gram–Schmidt

As we saw in Lecture 8, the Gram–Schmidt iteration applies a succession of elementary triangular matrices R_k on the right of A, so that the resulting matrix

$$A\underbrace{R_1R_2\cdots R_n}_{\hat{R}^{-1}} = \hat{Q}$$

has orthonormal columns. The product $\hat{R} = R_n^{-1}\cdots R_2^{-1}R_1^{-1}$ is upper-triangular too, and thus $A = \hat{Q}\hat{R}$ is a reduced QR factorization of A.

In contrast, the Householder method applies a succession of elementary unitary matrices Q_k on the left of A, so that the resulting matrix

$$\underbrace{Q_n\cdots Q_2Q_1}_{Q^*}A = R$$

is upper-triangular. The product $Q = Q_1^*Q_2^*\cdots Q_n^*$ is unitary too, and therefore $A = QR$ is a full QR factorization of A.

The two methods can thus be summarized as follows:

Gram–Schmidt: triangular orthogonalization,
Householder: orthogonal triangularization.

Triangularizing by Introducing Zeros

At the heart of the Householder method is an idea originally proposed by Alston Householder in 1958. This is an ingenious way of designing the unitary matrices Q_k so that $Q_n \cdots Q_2Q_1A$ is upper-triangular.

The matrix Q_k is chosen to introduce zeros below the diagonal in the kth column while preserving all the zeros previously introduced. For example, in the 5×3 case, three operations Q_k are applied, as follows. In these matrices, the symbol $\times$ represents an entry that is not necessarily zero, and boldfacing indicates an entry that has just been changed. Blank entries are zero.

$$\underset{A}{\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}} \xrightarrow{Q_1} \underset{Q_1A}{\begin{bmatrix} \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}} \xrightarrow{Q_2} \underset{Q_2Q_1A}{\begin{bmatrix} \times & \times & \times \\ & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} \end{bmatrix}} \xrightarrow{Q_3} \underset{Q_3Q_2Q_1A}{\begin{bmatrix} \times & \times & \times \\ & \times & \times \\ & & \boldsymbol{\times} \\ & & \mathbf{0} \\ & & \mathbf{0} \end{bmatrix}} \qquad (10.1)$$

First, Q_1 operates on rows $1, \ldots, 5$, introducing zeros in positions $(2,1)$, $(3,1)$, $(4,1)$, and $(5,1)$. Next, Q_2 operates on rows $2, \ldots, 5$, introducing zeros in positions $(3,2)$, $(4,2)$, and $(5,2)$ but not destroying the zeros introduced by Q_1. Finally, Q_3 operates on rows $3, \ldots, 5$, introducing zeros in positions $(4,3)$ and $(5,3)$ without destroying any of the zeros introduced earlier.

In general, Q_k operates on rows $k, \ldots, m$. At the beginning of step k, there is a block of zeros in the first $k-1$ columns of these rows. The application of Q_k forms linear combinations of these rows, and the linear combinations of the zero entries remain zero. After n steps, all the entries below the diagonal have been eliminated and $Q_n \cdots Q_2Q_1A = R$ is upper-triangular.

Householder Reflectors

How can we construct unitary matrices Q_k to introduce zeros as indicated in (10.1)? The standard approach is as follows. Each Q_k is chosen to be a unitary matrix of the form

$$Q_k = \begin{bmatrix} I & 0 \\ 0 & F \end{bmatrix}, \qquad (10.2)$$

where I is the $(k-1) \times (k-1)$ identity and F is an $(m-k+1) \times (m-k+1)$ unitary matrix. Multiplication by F must introduce zeros into the

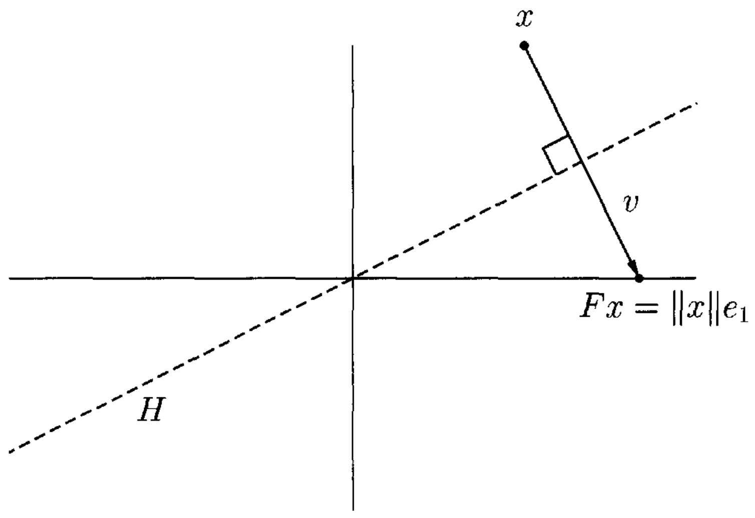


Figure 10.1. *A Householder reflection.*

kth column. The Householder algorithm chooses F to be a particular matrix called a *Householder reflector*.

Suppose, at the beginning of step k, the entries $k, \ldots, m$ of the kth column are given by the vector $x \in \mathbb{C}^{m-k+1}$. To introduce the correct zeros into the kth column, the Householder reflector F should effect the following map:

$$x = \begin{bmatrix} \times \\ \times \\ \times \\ \vdots \\ \times \end{bmatrix} \quad \xrightarrow{F} \quad Fx = \begin{bmatrix} \|x\| \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \|x\|e_1. \tag{10.3}$$

(We shall modify this idea by a $\pm$ sign in a moment.) The idea for accomplishing this is indicated in Figure 10.1. The reflector F will reflect the space $\mathbb{C}^{m-k+1}$ across the hyperplane H orthogonal to $v = \|x\|e_1 - x$. A *hyperplane* is the higher-dimensional generalization of a two-dimensional plane in three-space—a three-dimensional subspace of a four-dimensional space, a four-dimensional subspace of a five-dimensional space, and so on. In general, a hyperplane can be characterized as the set of points orthogonal to a fixed nonzero vector. In Figure 10.1, that vector is $v = \|x\|e_1 - x$, and one can think of the dashed line as a depiction of H viewed "edge on."

When the reflector is applied, every point on one side of the hyperplane H is mapped to its mirror image on the other side. In particular, x is mapped to $\|x\|e_1$. The formula for this reflection can be derived as follows. In (6.11) we have seen that for any $y \in \mathbb{C}^m$, the vector

$$Py = \left(I - \frac{vv^*}{v^*v}\right)y = y - v\left(\frac{v^*y}{v^*v}\right)$$

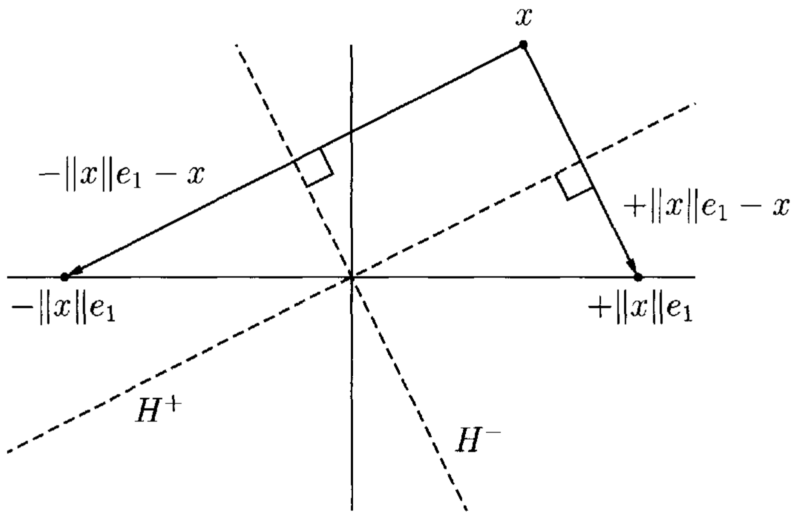


Figure 10.2. *Two possible reflections. For numerical stability, it is important to choose the one that moves x the larger distance.*

is the orthogonal projection of y onto the space H. To reflect y across H, we must not stop at this point; we must go exactly twice as far in the same direction. The reflection Fy should therefore be

$$Fy = \left(I - 2\frac{vv^*}{v^*v}\right)y = y - 2v\left(\frac{v^*y}{v^*v}\right).$$

Hence the matrix F is

$$F = I - 2\frac{vv^*}{v^*v}. \tag{10.4}$$

Note that the projector P (rank $m-1$) and the reflector F (full rank, unitary) differ only in the presence of a factor of 2.

The Better of Two Reflectors

In (10.3) and in Figure 10.1 we have simplified matters, for in fact, there are many Householder reflections that will introduce the zeros needed. The vector x can be reflected to $z\|x\|e_1$, where z is any scalar with $|z| = 1$. In the complex case, there is a circle of possible reflections, and even in the real case, there are two alternatives, represented by reflections across two different hyperplanes, H^+ and H^-, as illustrated in Figure 10.2.

Mathematically, either choice of sign is satisfactory. However, this is a case where the goal of numerical stability—insensitivity to rounding errors—dictates that one choice should be taken rather than the other. For numerical stability, it is desirable to reflect x to the vector $z\|x\|e_1$ that is not too close to x itself. To achieve this, we can choose $z = -\text{sign}(x_1)$, where x_1 denotes the first component of x, so that the reflection vector becomes $v = -\text{sign}(x_1)\|x\|e_1 - x$, or, upon clearing the factors -1,

$$v = \operatorname{sign}(x_1)\|x\|e_1 + x. \tag{10.5}$$

To make this a complete prescription, we may arbitrarily impose the convention that $\operatorname{sign}(x_1) = 1$ if $x_1 = 0$.

It is not hard to see why the choice of sign makes a difference for stability. Suppose that in Figure 10.2, the angle between H^+ and the e_1 axis is very small. Then the vector $v = \|x\|e_1 - x$ is much smaller than x or $\|x\|e_1$. Thus the calculation of v represents a subtraction of nearby quantities and will tend to suffer from cancellation errors. If we pick the sign as in (10.5), we avoid such effects by ensuring that $\|v\|$ is never smaller than $\|x\|$.

The Algorithm

We now formulate the whole Householder algorithm. To do this, it will be helpful to utilize a new (MATLAB-style) notation. If A is a matrix, we define $A_{i:i',j:j'}$ to be the $(i'-i+1)\times(j'-j+1)$ submatrix of A with upper-left corner a_{ij} and lower-right corner $a_{i',j'}$. In the special case where the submatrix reduces to a subvector of a single row or column, we write $A_{i,j:j'}$ or $A_{i:i',j}$, respectively.

The following algorithm computes the factor R of a QR factorization of an $m \times n$ matrix A with $m \geq n$, leaving the result in place of A. Along the way, n reflection vectors $v_1, \dots, v_n$ are stored for later use.

Algorithm 10.1. Householder QR Factorization

```
for $k = 1$ to $n$
    $x = A_{k:m,k}$
    $v_k = \operatorname{sign}(x_1)\|x\|_2 e_1 + x$
    $v_k = v_k/\|v_k\|_2$
    $A_{k:m,k:n} = A_{k:m,k:n} - 2v_k(v_k^* A_{k:m,k:n})$
```

Applying or Forming Q

Upon the completion of Algorithm 10.1, A has been reduced to upper-triangular form; this is the matrix R in the QR factorization $A = QR$. The unitary matrix Q has not, however, been constructed, nor has its n-column submatrix $\hat{Q}$ corresponding to a reduced QR factorization. There is a reason for this. Constructing Q or $\hat{Q}$ takes additional work, and in many applications, we can avoid this by working directly with the formula

$$Q^* = Q_n \cdots Q_2 Q_1 \tag{10.6}$$

or its conjugate

$$Q = Q_1 Q_2 \cdots Q_n. \tag{10.7}$$

(No asterisks have been forgotten here; recall that each Q_j is hermitian.)

For example, in Lecture 7 we saw that a square system of equations $Ax = b$ can be solved via QR factorization of A. The only way in which Q was used in this process was in the computation of the product Q^*b. By (10.6), we can calculate Q^*b by a sequence of n operations applied to b, the same operations that were applied to A to make it triangular. The algorithm is as follows.

Algorithm 10.2. Implicit Calculation of a Product Q^*b

```
for $k = 1$ to $n$
    $b_{k:m} = b_{k:m} - 2v_k(v_k^* b_{k:m})$
```

Similarly, the computation of a product Qx can be achieved by the same process executed in reverse order.

Algorithm 10.3. Implicit Calculation of a Product Qx

```
for $k = n$ downto 1
    $x_{k:m} = x_{k:m} - 2v_k(v_k^* x_{k:m})$
```

The work involved in either of these algorithms is of order $O(mn)$, not $O(mn^2)$ as in Algorithm 10.1 (see below).

Sometimes, of course, one may wish to construct the matrix Q explicitly. This can be achieved in various ways. We can construct QI via Algorithm 10.3 by computing its columns $Qe_1, Qe_2, \ldots, Qe_m$. Alternatively, we can construct Q^*I via Algorithm 10.2 and then conjugate the result. A variant of this idea is to conjugate each step rather than the final product, that is, to construct IQ by computing its rows $e_1^*Q, e_2^*Q, \ldots, e_m^*Q$ as suggested by (10.7). Of these various ideas, the best is the first one, based on Algorithm 10.3. The reason is that it begins with operations involving Q_n, Q_{n-1}, and so on that modify only a small part of the vector they are applied to; if advantage is taken of this sparsity property, a speed-up is achieved.

If only $\hat{Q}$ rather than Q is needed, it is enough to compute the columns $Qe_1, Qe_2, \ldots, Qe_n$.

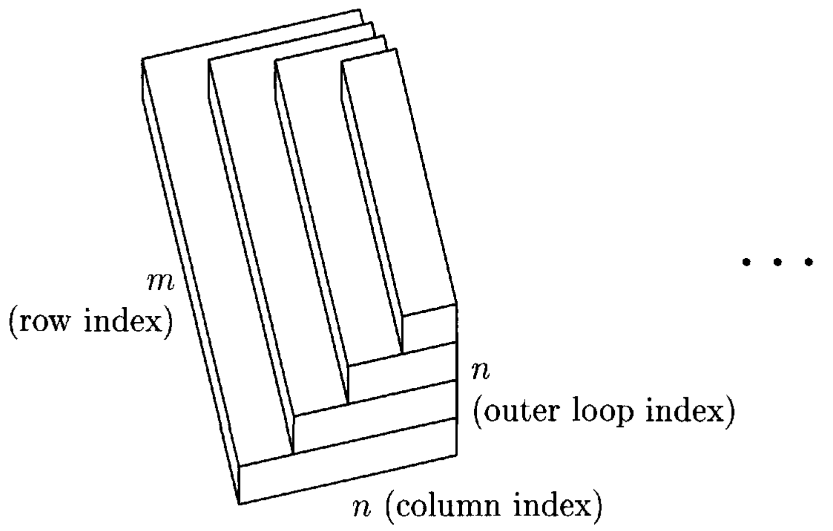
Operation Count

The work involved in Algorithm 10.1 is dominated by the innermost loop,

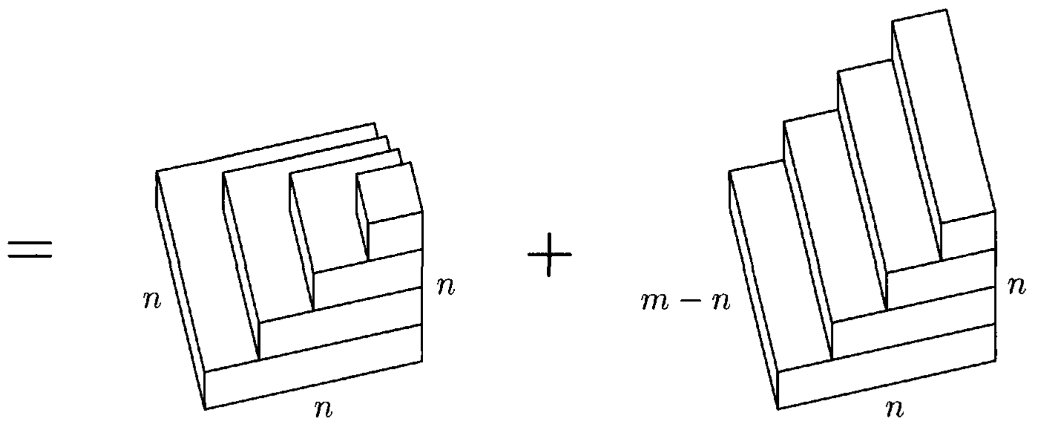
$$A_{k:m,j} - 2v_k(v_k^* A_{k:m,j}). \tag{10.8}$$

If the vector length is $l = m - k + 1$, this calculation requires $4l - 1 \sim 4l$ scalar operations: l for the subtraction, l for the scalar multiplication, and $2l - 1$ for the dot product. This is ~ 4 flops for each entry operated on.

We may add up these four flops per entry by geometric reasoning, as in Lecture 8. Each successive step of the outer loop operates on fewer rows, because during step k, rows $1, \ldots, k-1$ are not changed. Furthermore, each step operates on fewer columns, because columns $1, \ldots, k-1$ of the rows operated on are zero and are skipped. Thus the work done by one outer step can be represented by a single layer of the following solid:



The total number of operations corresponds to four times the volume of the solid. To determine the volume pictorially we may divide the solid into two pieces:



The solid on the left has the shape of a ziggurat and converges to a pyramid as $n \to \infty$, with volume $\frac{1}{3}n^3$. The solid on the right has the shape of a staircase and converges to a prism as $m, n \to \infty$, with volume $\frac{1}{2}(m-n)n^2$. Combined, the volume is $\sim \frac{1}{2}mn^2 - \frac{1}{6}n^3$. Multiplying by four flops per unit volume, we find

$$\text{Work for Householder orthogonalization:} \quad \sim 2mn^2 - \frac{2}{3}n^3 \text{ flops.} \tag{10.9}$$

Exercises

10.1. Determine the (a) eigenvalues, (b) determinant, and (c) singular values of a Householder reflector. For the eigenvalues, give a geometric argument as well as an algebraic proof.

10.2. (a) Write a MATLAB function `[W,R] = house(A)` that computes an implicit representation of a full QR factorization $A = QR$ of an $m \times n$ matrix A with $m \geq n$ using Householder reflections. The output variables are a lower-triangular matrix $W \in \mathbb{C}^{m\times n}$ whose columns are the vectors v_k defining the successive Householder reflections, and a triangular matrix $R \in \mathbb{C}^{n,n}$.

(b) Write a MATLAB function `Q = formQ(W)` that takes the matrix W produced by `house` as input and generates a corresponding $m \times m$ orthogonal matrix Q.

10.3. Let Z be the matrix

$$Z = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 7 \\ 4 & 2 & 3 \\ 4 & 2 & 2 \end{bmatrix}.$$

Compute three reduced QR factorizations of Z in MATLAB: by the Gram–Schmidt routine `mgs` of Exercise 8.2, by the Householder routines `house` and `formQ` of Exercise 10.2, and by MATLAB's built-in command `[Q,R] = qr(Z,0)`. Compare these three and comment on any differences you see.

10.4. Consider the 2×2 orthogonal matrices

$$F = \begin{bmatrix} -c & s \\ s & c \end{bmatrix}, \qquad J = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \tag{10.10}$$

where $s = \sin\theta$ and $c = \cos\theta$ for some θ. The first matrix has $\det F = -1$ and is a reflector—the special case of a Householder reflector in dimension 2. The second has $\det J = 1$ and effects a rotation instead of a reflection. Such a matrix is called a *Givens rotation*.

(a) Describe exactly what geometric effects left-multiplications by F and J have on the plane $\mathbb{R}^2$. (J rotates the plane by the angle θ, for example, but is the rotation clockwise or counterclockwise?)

(b) Describe an algorithm for QR factorization that is analogous to Algorithm 10.1 but based on Givens rotations instead of Householder reflections.

(c) Show that your algorithm involves six flops per entry operated on rather than four, so that the asymptotic operation count is 50% greater than (10.9).