# Numerical Linear Algebra

Notes By: Yuhang Cai

2023 Spring

# CONTENTS

# CHAPTER 1 MATLAB

MATLAB is a language for mathematical computations whose fundamental data types are vectors and matrices.

- Many built-in commands, SVD, FFT and matrix inversion;
- Built for large-scale scientific computing and small- and medium- scale experimentation in NLA.

## 1.1 Exp 1: Discrete Legendre Poly

The following lines of MATLAB construct this matrix and compute its reduced QR factorization.

```
x = (-128:128)'/128;
A= [x.^0 x.^1. x.^2 x.^3] ;
[Q,R] = qr(A,0);
```

The columns of the matrix $Q$ are essentially the first four Legendre polynomials. They differ slightly, by amounts close to plotting accuracy. They also differ in normalization, since a Legendre polynomial should satisfy $P_k(1) = 1$. We can fix this by diving each column of $Q$ by its final entry.

```
scale = Q(257,:);
Q = Q*diag(1 ./scale);
plot (Q)
```

The result of our computation is a plot that looks just like Figure 7.1.

## 1.2 Exp 2: Classical vs. Modified Gram-Schmidt

First, we construct a square matrix $A$ with random singular vbectors 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.
```

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 QR by classical GS
[QM, RM] = mgs(A);                           Compute a factorization QR by modified GS
```

Finally, we plot the diagonal entries $r_{jj}$.

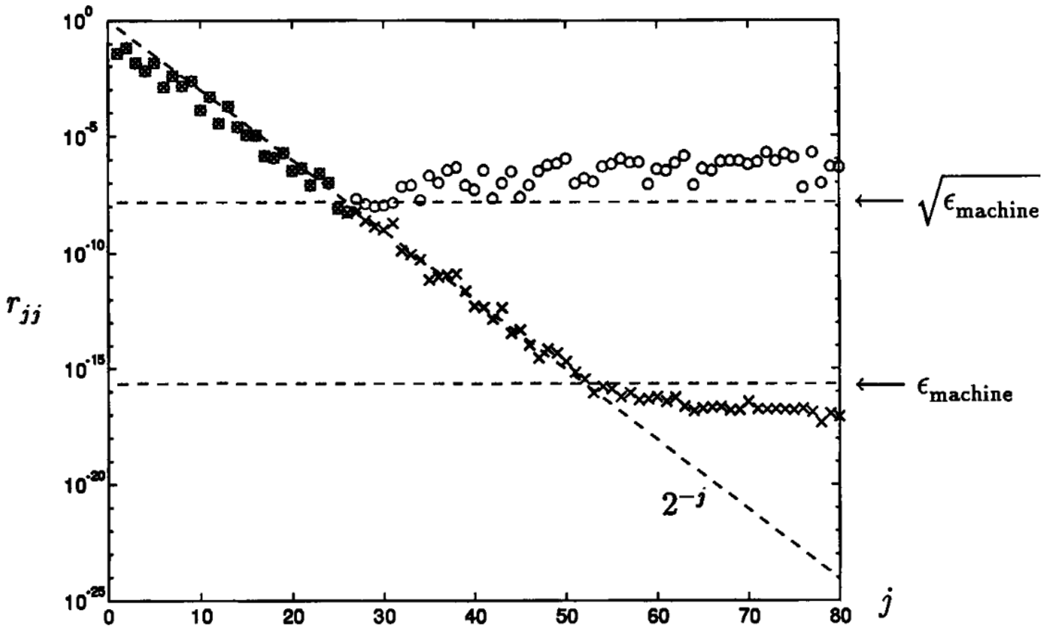


Figure 1.1: The classical GS is represented by circles while the modified GS is represented by crosses.

Notice that:

- $r_{jj}$ is close to $2^{-j}$. This is because
$$\begin{aligned} A &= 2^{-1}u_1v_1^* + 2^{-2}u_2v_2^* + 2^{-3}u_3v_3^* + \cdots + 2^{-80}u_{80}v_{80}^*, \\ a_j &= 2^{-1}\bar{v}_{j1}u_1 + 2^{-2}\bar{v}_{j2}u_2 + 2^{-3}\bar{v}_{j3}u_3 + \cdots + 2^{-80}\bar{v}_{j,80}u_{80}. \end{aligned}$$
Here $v_{ji}$ are almost constants $\approx 0.1$. In fact $u_i \approx q_i$ and $r_{ii} \approx 0.1 * 2^{-i}$.
- For classical GS, $r_{jj}$ stop at $10^{-8}$ while the modified GS is down to the order of $10^{-16}$, which is the level of machine epsilon.

Hence, the modified GS is more stable. Consequently the classical GS is rarely used, except sometimes on parallel computers in situations where advantages related to communication may outweigh the disadvantage of instability.

## 1.3 Exp 3: Numerical Loss of Orthogonality

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
$$A = \begin{bmatrix} 0.70000 & 0.70711 \\ 0.70001 & 0.70711 \end{bmatrix}$$
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 $2 \times 2$ case. At step $j = 1$, the first column is normalized, yielding
$$r_{11} = 0.98996, \quad q_1 = a_1/r_{11} = \begin{bmatrix} 0.70000/0.98996 \\ 0.70001/0.98996 \end{bmatrix} = \begin{bmatrix} 0.70710 \\ 0.70711 \end{bmatrix}$$
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

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

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

```
A = [.70000 .70711                                              Define A.
.70001 .70711] ;
[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 GS.
norm(Q'*Q - eye(2))                                    Test orthogonality of Q.
```

The results are:

$$\text{ans } = 2.3515e - 16, \quad \text{ans } = 2.3014e - 11.$$

# CHAPTER 2 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.

## 2.1 Householder and Gram-Schmidt

The GS iteration applies a succession of elementary triangular matrices $R_k$ on the right of $A$:

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

In contrast, the Householder method applies a succession of elementary unitary matrices $Q_k$ on the left of $A$,

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

The two methods can be summerized as the follows:

- GS: triangular orthogonalization.
- Householder: orthogonal triangularizaion.

## 2.2 Triangularization by Introducing Zeros

The idea of the Householder is to introduce zeros below the diagonal in the $k$th column while preserving all the zeros previously introduced.

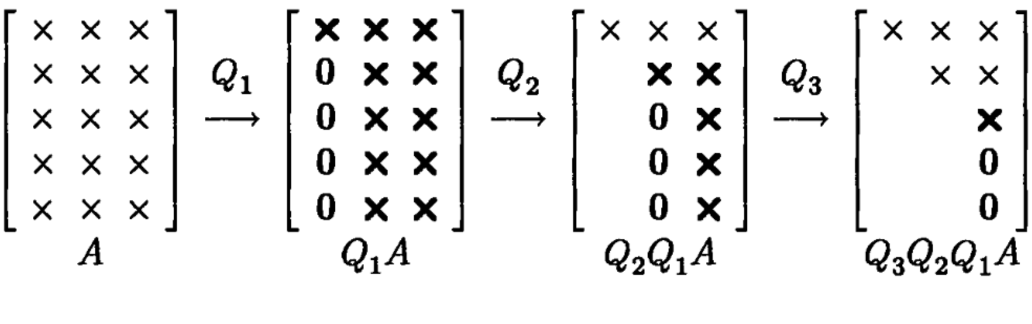


Figure 2.1: One example in $\mathbb{R}^{5\times 3}$

In fact, $Q_k$ 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.

## 2.3 Householder Reflectors

Each $Q_k$ is chosen to be a unitary matrix of the form:

$$Q_k = \begin{bmatrix} I & 0 \\ 0 & F \end{bmatrix},$$

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 $k$th column. The householder algorithm chooses $F$ to be a particular matrix called a Householder reflector. Suppose, at the beginning of step $k$, the entries $k, \dots, m$ of the $k$ th column are given by the vector $x \in \mathbb{C}^{m-k+1}$. To introduce the correct zeros into the $k$ th column, the Householder reflector $F$ should effect the following map:

$$x = \begin{bmatrix} \times \\ \times \\ \times \\ \vdots \\ \times \end{bmatrix} \quad \longrightarrow \quad Fx = \begin{bmatrix} \|x\| \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \|x\| e_1.$$

The idea for accomplishing this is indicated in Figure 10.2. The reflector $F$ will reflect the space $\mathbb{C}^{m-k+1}$ across the hyperplane $H$ orthogonal to $v = \|x\|e_1 - x$.

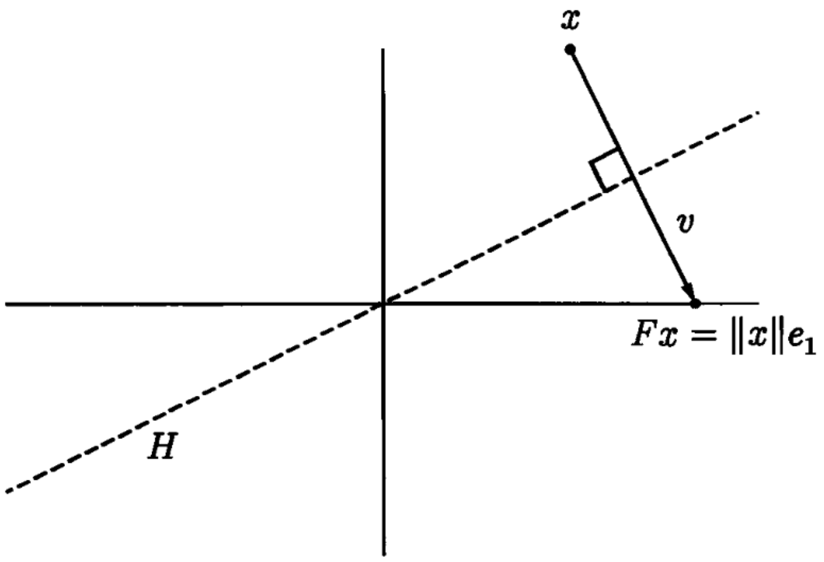


Figure 2.2: A Householder reflection

The formula for this reflection can be derived as follows. We know that for any $y \in \mathbb{C}^m$, the vector:

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

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

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

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.

## 2.4 The Better of Two Reflectors

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 10.3.

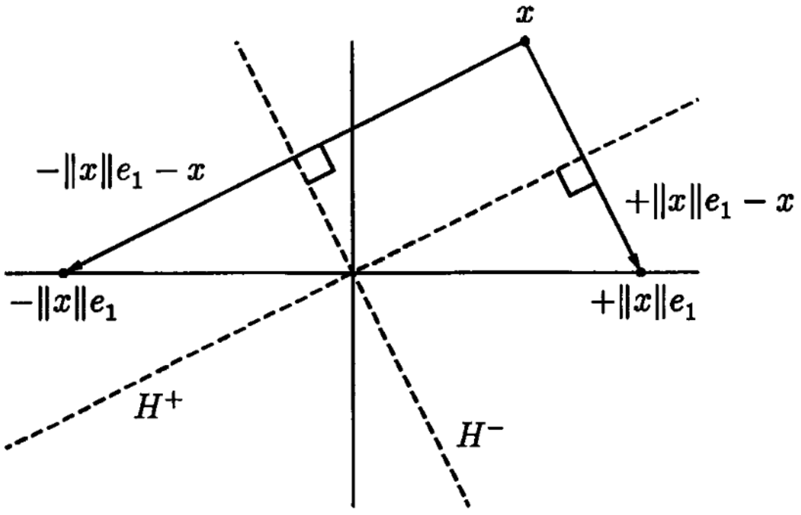


Figure 2.3: Two possible reflections

Mathematically, either choice of sign is satisfactory. However, this is a case where the goal of numerical stability-insensitivity to rounding errorsdictates 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 $z = -\operatorname{sign}(x_1)$, where $x_1$ denotes the first component of $x$, so that the reflection vector becomes $v = -\operatorname{sign}(x_1)\|x\|e_1 - x$, or

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

If $x_1 = 0$, we impose that $\operatorname{sign}(x_1) = 1$.

## 2.5 The Algorithm

We now formulate the whole Householder algorithm. 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.

**Algorithm 2.1:** Householder QR factorization

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

## 2.6 Applying or Forming $Q$

The algorithm 2.1 can get $R$ easily. However, we still have to form the matrix $Q$. Constructing $Q$ or $\hat{Q}$ takes additional work, and in many applications, we can avoid this by working directly with the formula

$$Q^* = Q_n \cdots Q_2 Q_1$$

or its conjugate

$$Q = Q_1 Q_2 \cdots Q_n.$$

In fact, we only need to the matvec $Q^* b$ or $Qx$. The algorithms are:

**Algorithm 2.2:** Implicit Calculation of a Product $Q^* b$

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

**Algorithm 2.3:** Implicit Calculation of a Product $Qx$

```
1 for $k = n$ to $1$ do
2     $x_{k:m} = x_{k:m} - 2v_k \left(v_k^* x_{k:m}\right)$
```

The work involved in either of these algorithms is of order $O(mn)$, not $O\left(mn^2\right)$ as in algorithm 2.1 (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 $e_1^* Q, e_2^* Q, \ldots, e_m^* Q$. Of these various ideas, the best is the first one, based on algorithm 2.3. 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 $Qe_1, Qe_2, \ldots, Qe_n$.

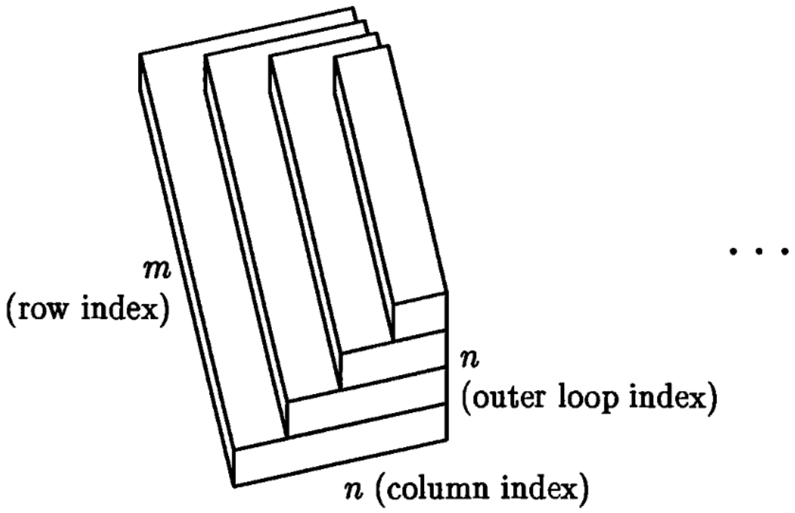
## 2.7 Operation Count

The work involved in algorithm 2.1 is dominated by the innermost loop,

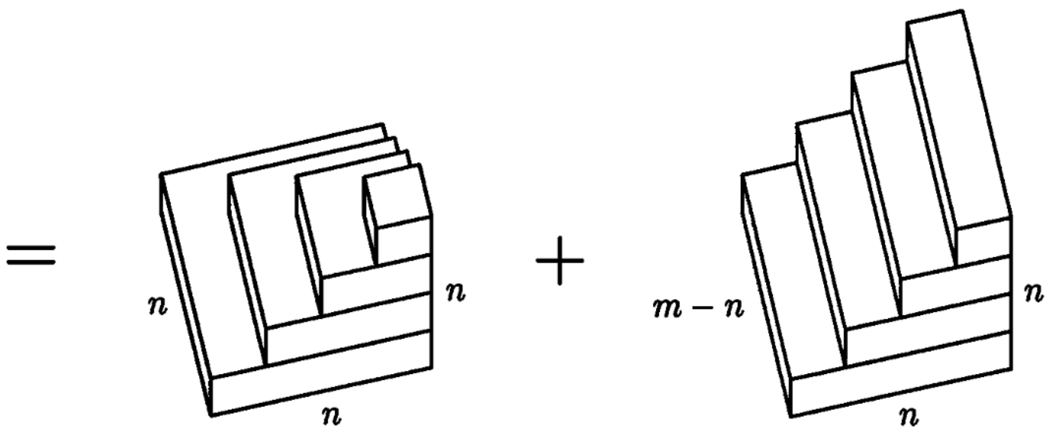
$$A_{k:m,j} - 2v_k \left(v_k^* A_{k:m,k}\right)$$

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 $\sim 4$ flops for each entry operated on.

We may add up these four flops per entry by geometric reasoning. 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:



This can be divide 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

**Corollary 2.1.**
Work for Householder orthogonalization: $\sim 2mn^2 - \frac{2}{3}n^3$ flops.

# CHAPTER 3
# LEAST SQUARES PROBLEMS

Least squares data-fitting has been an indispensable tool since its invention by Gauss and Legendre around 1800, with ramifications extending throughout the mathematical sciences. In the language of linear algebra, the problem here is the solution of an overdetermined system of equations $Ax = b$-rectangular, with more rows than columns. The least squares idea is to "solve" such a system by minimizing the 2 -norm of the residual $b - Ax$.

## 3.1 The Problem

Consider a linear system of equations having $n$ unknowns but $m > n$ equations. Symbolically, we wish to find a vector $x \in \mathbb{C}^n$ that satisfies $Ax = b$, where $A \in \mathbb{C}^{m\times n}$ and $b \in \mathbb{C}^m$.

In general, such a problem has no solution. A suitable vector $x$ exists only if $b$ lies in range $(A)$, and since $b$ is an $m$-vector, whereas range $(A)$ is of dimension at most $n$, this is true only for exceptional choices of $b$. We say that a rectangular system of equations with $m > n$ is **overdetermined**. The vector known as the **residual**,

$$r = b - Ax \in \mathbb{C}^m$$

can perhaps be made quite small by a suitable choice of $x$, but in general it cannot be made equal to zero.

However, we can try to minimize the norm of $r$. If we take the 2-norm, the problem is:

$$\min_{x\in\mathbb{C}^n} \quad \|b - Ax\|_2 \tag{3.1}$$

Geometrically, we seek a vector $x \in \mathbb{C}^n$ such that the vector $Ax \in \mathbb{C}^m$ is the closest point in range$(A)$ to $b$.

## 3.2 Orthogonal Projection and the Normal Equations

Our goal is to find the closest point $Ax$ in range $(A)$ to $b$, so that the norm of the residual $r = b - Ax$ is minimized.

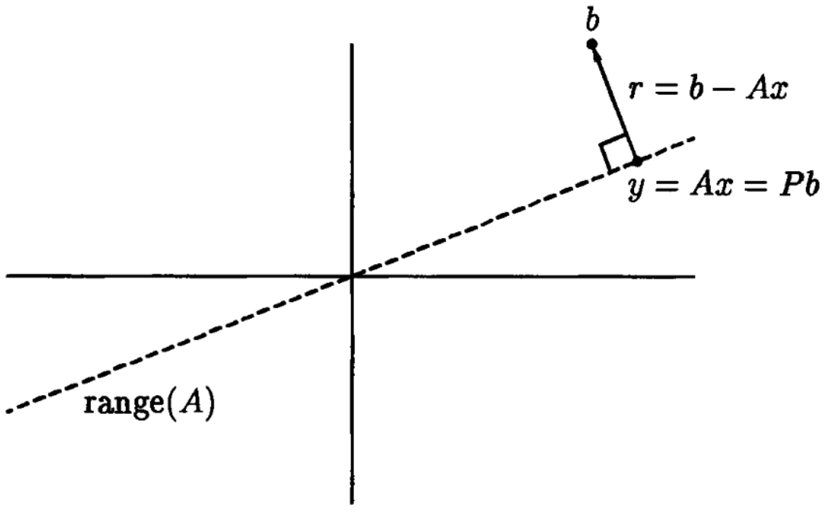


Figure 3.1: Formulation of LS in terms of orthogonal projection

It is clear geometrically that this will occur provided $Ax = Pb$, where $P \in \mathbb{C}^{m\times m}$ is the orthogonal projector (Lecture 6) that maps $\mathbb{C}^m$ onto range$(A)$. In other words, the residual $r = b - Ax$ must be orthogonal to range$(A)$. We formulate this condition as the following theorem.

**Theorem 3.1.**
Let $A \in \mathbb{C}^{m\times n}(m \geqslant n)$ and $b \in \mathbb{C}^m$ be given. A vector $x \in \mathbb{C}^n$ minimizes the residual norm $\|r\|_2 = \|b - Ax\|_2$, thereby solving the least squares problem Equation 3.1 if and only if $r \perp \text{range}(A)$, that is,

$$A^*r = 0$$

or equivalently,

$$A^*Ax = A^*b \tag{3.2}$$

or again equivalently,

$$Pb = Ax,$$

where $P \in \mathbb{C}^{m\times m}$ is the orthogonal projector onto range $(A)$. The $n \times n$ system of Equation 3.2, known as the **normal equations**, is nonsingular if and only if $A$ has full rank. Consequently the solution $x$ is unique if and only if $A$ has full rank.

## 3.3 Pseudoinverse

**Definition 3.2 (Pseudoinverse).**
Given a full rank matrix $A$, the pseudoinverse of $A$ is

$$A^\dagger = (A^*A)^{-1}A^* \in \mathbb{C}^{n\times m}.$$

Hence, the LS problem is:

$$x = A^\dagger b, \quad y = Pb.$$

## 3.4 Normal Equations

Since $A^*A$ is Hermitian, the standard method of solving normal equations is by **Cholesky factorization**. This method constructs a factorization $A^*A = R^*R$, where $R$ is upper-triangular. Hence, the equations are

$$R^*Rx = A^*b.$$

Here is the algorithm.

**Algorithm 3.1:** Least Squares via Normal Equations

```
1 Form the matrix $A^*A$ and the vector $A^*b$.;
2 Compute the Cholesky factorization $A^*A = R^*R$.;
3 Solve the lower-triangular system $R^*w = A^*b$ for $w$. ;
4 Solve the upper triangular system $Rx = w$ for $x$.
```

The steps that dominate the work for this computation are the first two (for steps 3 and 4, see Chapter 17). Because of symmetry, the computation of $A^*A$ requires only $mn^2$ flops, half what the cost would be if $A$ and $A^*$ were arbitrary matrices of the same dimensions. Cholesky factorization, which also exploits symmetry, requires $n^3/3$ flops. All together, solving least squares problems by the normal equations involves the following total operation count:

**Corollary 3.3.**
The work for algorithm 3.1: $\sim mn^2 + \frac{1}{3}n^3$ flops.

## 3.5 QR factorization

The "modern classical" method for solving least squares problems, popular since the 1960 s, is based upon reduced QR factorization. By Gram-Schmidt orthogonalization or, more usually, Householder triangularization, one constructs a factorization $A = \hat{Q}\hat{R}$. The orthogonal projector $P$ can then be written as $P = \hat{Q}\hat{Q}^*$. Hence,

$$y = Pb = \hat{Q}\hat{Q}^*b.$$

The system $Ax = y$ can be written as:

$$\hat{Q}\hat{R}x = \hat{Q}\hat{Q}^*b \Rightarrow \hat{R}x = \hat{Q}^*b.$$

**Algorithm 3.2:** Least Squares via QR Factorization

```
1 Compute the reduced QR factorization $A = \hat{Q}\hat{R}$;
2 Compute the vector $\hat{Q}^*b$;
3 Solve the upper-triangular system $\hat{R}x = \hat{Q}^*b$ for x.
```

The work for algorithm 3.2 is dominated by the cost of QR. If we use Householder reflections, we have

**Corollary 3.4.**
Work for algorithm 3.2: $\sim 2mn^2 - \frac{2}{3}n^3$ flops.

## 3.6 SVD

In Chapter 31 we shall describe an algorithm for computing the reduced singular value decomposition $A = \hat{U}\hat{\Sigma}V^*$. Then $P = \hat{U}\hat{U}^*$, giving

$$y = Pb = \hat{U}\hat{U}^*b,$$

and then

$$\hat{U}\hat{\Sigma}V^*x = \hat{U}\hat{U}^*b \Rightarrow \hat{\Sigma}V^*x = \hat{U}^*b.$$

The algorithm is:

**Algorithm 3.3:** Least Squares via SVD

```
1 Compute the reduced SVD $A = \hat{U}\hat{\Sigma}V^*$;
2 Compute the vector $\hat{U}^*b$;
3 Solve the diagonal system $\hat{\Sigma}w = \hat{U}^*b$ for $w$;
4 Set $x = Vw$.
```

AS we shall see in Chapter 31, for $m \gg n$ this cost is approximately the same as for QR, but for $m \approx n$ the SVD is more expensive.

**Corollary 3.5.**
Work for algorithm 3.3: $\sim 2mn^2 + 11n^3$ flops.

## 3.7 Comparison of Algorithms

Each of the methods we have described is advantageous in certain situations.

- algorithm 3.1 is the fastest but solving the normal equations might not be stable;
- Thus for many years, numerical analysts have recommended algorithm 3.2 instead as the standard method for least squares problems. This is indeed a natural and elegant algorithm, and we recommend it for "daily use."
- If A is close to rank-deficient, however, it turns out that algorithm 3.2 itself has less-than-ideal stability properties, and in such cases there are good reasons to turn to algorithm 3.3 based on the SVD.

# Part I

# Conditioning and Stability

# CHAPTER 4 CONDITIONING AND CONDITION NUMBERS

Conditioning pertains to the perturbation behavior of a mathematical problem. Stability pertains to the perturbation behavior of an algorithm used to solve that problem on a computer.

## 4.1 Condition of a Problem

In the abstract, we can give the following definition of mathematical problems.

**Definition 4.1** (Problem).
A mathematical problem is a function $f : X \to Y$ where $X$ is the vector space of data and $Y$ is the vector space of solutions.

This function $f$ is usually nonlinear (even in linear algebra), but most of the time it is at least continuous.

Typically we shall be concerned with the behavior of a problem $f$ at a particular data point $x \in X$ (the behavior may vary greatly from one point to another). The combination of a problem $f$ with prescribed data $x$ might be called a **problem instance**, but it is more usual, though occasionally confusing, to use the term **problem** for this notion too.

A **well-conditioned** problem (instance) is one with the property that all small perturbations of $x$ lead to only small changes in $f(x)$. An ill-conditioned problem is one with the property that some small perturbation of $x$ leads to a large change in $f(x)$.

## 4.2 Absolute Condition Number

**Definition 4.2** (Absolute condition number).
Let $\delta x$ denote a small perturbation of $x$, and write $\delta f = f(x + \delta x) - f(x)$. The absolute condition number $\hat{\kappa} = \hat{\kappa}(x)$ of the problem $f$ at $x$ is defined as

$$\hat{\kappa} = \lim_{\delta \to 0} \sup_{\|\delta x\| \leqslant \delta} \frac{\|\delta f\|}{\|\delta x\|}$$

We can also write this as:

$$\hat{\kappa} = \sup_{\delta x} \frac{\|\delta f\|}{\|\delta x\|}.$$

If $f$ is differentiable, we can evaluate the condition number by the Jacobian matrix. Let $J(x)$ be the Jacobian of $f$ at $x$. We have

$$\hat{\kappa} = \|J(x)\|,$$

where $\|J(x)\|$ is the induced norm by the norms on $X$ and $Y$.

## 4.3 Relative Condition Number

When we are concerned with relative changes, we need the notion of relative condition.

**Definition 4.3 (Relative condition number).**
The relative condition number $\kappa = \kappa(x)$ is defined by

$$\kappa = \lim_{\delta \to 0} \sup_{\|\delta x\| \leqslant \delta} \left( \frac{\|\delta f\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right).$$

If $f$ is differentiable, we can express this quantity in terms of the Jacobian:

$$\kappa = \frac{\|J(x)\|}{\|f(x)\|/\|x\|}.$$

Both absolute and relative condition numbers have their uses, but the latter are more important in numerical analysis. This is ultimately because the floating point arithmetic used by computers introduces relative errors rather than absolute ones; see the next lecture. A problem is **well-conditioned** if $\kappa$ is small (e.g., $1, 10, 10^2$), and **ill-conditioned** if $\kappa$ is large (e.g., 10 $10^6$).

**Example 4.4.**

- Consider the trivial problem of obtaining the scalar $x/2$ from $x \in \mathbb{C}$. The Jacobian of the function $f : x \mapsto x/2$ is just the derivative $J = f' = 1/2$, so by (12.6),

$$\kappa = \frac{\|J\|}{\|f(x)\|/\|x\|} = \frac{1/2}{(x/2)/x} = 1.$$

  This problem is well-conditioned by any standard.
- Consider the problem of computing $\sqrt{x}$ for $x > 0$. The Jacobian of $f : x \mapsto \sqrt{x}$ is the derivative $J = f' = 1/(2\sqrt{x})$, so we have

$$\kappa = \frac{\|J\|}{\|f(x)\|/\|x\|} = \frac{1/(2\sqrt{x})}{\sqrt{x}/x} = \frac{1}{2}.$$

  Again, this is a well-conditioned problem.
- Consider the problem of obtaining the scalar $f(x) = x_1 - x_2$ from the vector $x = (x_1, x_2)^* \in \mathbb{C}^2$. For simplicity, we use the $\infty$-norm on the data space $\mathbb{C}^2$. The Jacobian of $f$ is

$$J = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix}$$

  with $\|J\|_\infty = 2$. The condition number is thus

$$\kappa = \frac{\|J\|_\infty}{\|f(x)\|/\|x\|} = \frac{2}{|x_1 - x_2| \,/ \max\{|x_1|, |x_2|\}}.$$

  This quantity is large if $|x_1 - x_2| \approx 0$, so the problem is ill-conditioned when $x_1 \approx x_2$, matching our intuition of the hazards of "cancellation error."
- The problem of computing the eigenvalues of a nonsymmetric matrix is also often ill-conditioned. One can see this by comparing the two matrices

$$\begin{bmatrix} 1 & 1000 \\ 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 1000 \\ 0.001 & 1 \end{bmatrix},$$

  whose eigenvalues are $\{1, 1\}$ and $\{0, 2\}$, respectively. On the other hand, if a matrix $A$ is symmetric (more generally, if it is normal), then its eigenvalues are well-conditioned. It can be shown that if $\lambda$ and $\lambda + \delta\lambda$ are corresponding eigenvalues of $A$ and $A + \delta A$, then $|\delta\lambda| \leqslant \|\delta A\|_2$, with equality if $\delta A$ is a multiple of the identity. Thus the absolute condition number of the symmetric eigenvalue problem is $\hat{\kappa} = 1$, if perturbations are measured in the 2-norm, and the relative condition number is $\kappa = \|A\|_2/|\lambda|$.

**Example 4.5** (Root of Polynomial).
The determination of the roots of a polynomial, given the coefficients, is a classic example of an ill-conditioned problem. Assume we have a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$. If the ith coefficient $a_i$ is pertubed by infinitesimal quantity $\delta a_i$, the perturbation of the $j$th root $x_j$ satisfies:

$$\sum_{k\neq i} a_k(x_j + \delta x_j)^k + (a_i + \delta a_i)(x_j + \delta x_j)^i = 0 \Rightarrow \delta a_i x_j^i + \sum_k k a_k x_j^{k-1}\delta x_j = 0.$$

Hence, $\delta x_j = \frac{(\delta a_i)x_j^i}{p'(x_j)}$. So the condition number of $x_j$ w.r.t. perturbations of $a_i$ is:

$$\kappa = \frac{|\delta x_j|}{|x_j|} / \frac{|\delta a_i|}{|a_i|} = \frac{|a_i x_j^{i-1}|}{|p'(x_j)|}.$$

This number can be very large. If we consider the "Wilkinson polynomial",

$$p(x) = \prod_{i=1}^{2} 0(x - i) = a_0 + a_1 x + \cdots + a_{19}x^{19} + x^{20}.$$

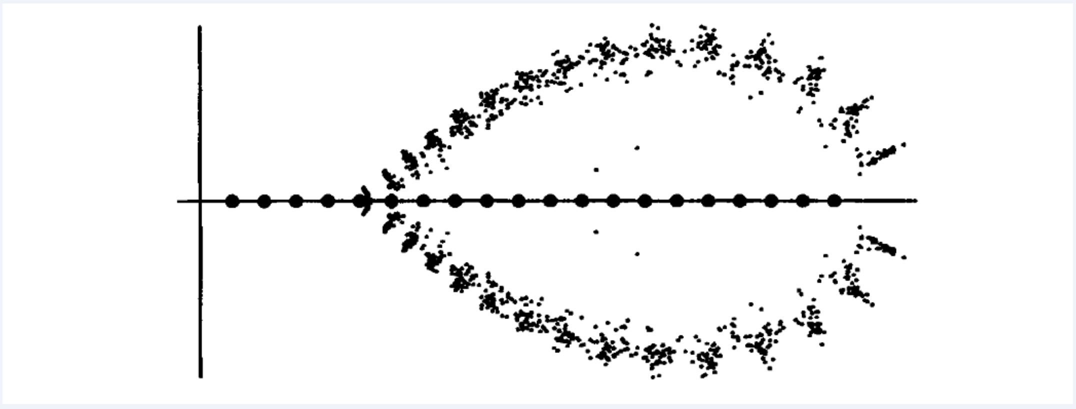


Figure 4.1: Wilkinson's classic example of ill-conditioning. The large dots are the roots of the unperturbed polynomial. The samll dots are the super imposed roots in the complex plane of 100 randomly perturbed polynomials with coefficients defined by $\overline{a_k} = a_k(1 + 10^{-10}r_k)$, where $r_k \sim \mathcal{N}(0, 1)$.

The most sensitive root of this polynomial is $x = 15$, and it's most sensitive in the coefficient $a_{15} \approx 1.67 \times 10^9$. The condition number is:

$$\kappa \approx \frac{1.67 \times 10^8 \cdot 15^14}{5!14!} \approx 5.1 \times 10^{13}.$$

## 4.4 Condition of Matrix-Vector Multiplication

Given $A \in \mathbb{C}^{m\times n}$ and consider the problem of computing $Ax$ is

$$\kappa = \sup_{\delta x}\left(\frac{\|A(x + \delta x) - Ax\|}{\|Ax\|} / \frac{\|\delta x\|}{\|x\|}\right) = \sup_{\delta x} \frac{\|A\delta x\|}{\|\delta x\|} / \frac{\|Ax\|}{\|x\|} = \|A\|\frac{\|x\|}{\|Ax\|}.$$

Then we can use the fact that $\|x\|/\|Ax\| \leqslant \|A^{-1}\|$ to loosen to a bound independent of $x$:

$$\kappa \leqslant \|A\| \left\|A^{-1}\right\|.$$

In fact, $A$ need not have been square. If $A \in \mathbb{C}^{m\times n}$ with $m \geqslant n$ has full rank, we can replace $A^{-1}$ by the pseudoinverse $A^{\dagger}$.

**Theorem 4.6.**
Let $A \in \mathbb{C}^{m\times m}$ be nonsingular and consider the equation $Ax = b$. The problem of computing $b$, given $x$, has condition number

$$\kappa = \|A\| \frac{\|x\|}{\|b\|} \leqslant \|A\| \left\|A^{-1}\right\| \tag{4.1}$$

with respect to perturbations of $x$. The problem of computing $x$, given $b$, has condition number

$$\kappa = \left\|A^{-1}\right\| \frac{\|b\|}{\|x\|} \leqslant \|A\| \left\|A^{-1}\right\| \tag{4.2}$$

with respect to perturbations of $b$. If $\|\cdot\| = \|\cdot\|_2$, then equality holds in the first inequality if $x$ is a multiple of a right singular vector of $A$ corresponding to the minimal singular value $\sigma_m$, and equality holds in the second inequality if $b$ is a multiple of a left singular vector of $A$ corresponding to the maximal singular value $\sigma_1$.

## 4.5 Condition Number of a Matrix

The product $\|A\| \|A^{-1}\|$ comes up so often that it has its own name:

**Definition 4.7** (Condition number of matrix).
The condition number of a mtrix $A \in \mathbb{C}^{n\times n}$ denoted by $\kappa(A)$:

$$\kappa(A) = \|A\| \left\|A^{-1}\right\|.$$

If $\kappa(A)$ is small, $A$ is said to be **well-conditioned**; if $\kappa(A)$ is large, $A$ is **ill-conditioned**. If $A$ is singular, it is customary to write $\kappa(A) = \infty$. Note that if $\|\cdot\| = \|\cdot\|_2$, then $\|A\| = \sigma_1$ and $\|A^{-1}\| = 1/\sigma_m$. Thus

$$\kappa(A) = \frac{\sigma_1}{\sigma_m}$$

in the 2-norm, and it is this formula that is generally used for computing 2−norm condition numbers of matrices.

For a rectangular matrix $A \in \mathbb{C}^{m\times n}$ of full rank, $m \geqslant n$, the condition number is defined in terms of the pseudoinverse: $\kappa(A) = \|A\| \|A^{+}\|$. Since $A^{+}$ is motivated by least squares problems, this definition is most useful in the case $\|\cdot\| = \|\cdot\|_2$, where we have

$$\kappa(A) = \frac{\sigma_1}{\sigma_n}.$$

## 4.6 Condition of a System of Equations

In Theorem 4.6 , we held $A$ fixed and perturbed $x$ or $b$. What happens if we perturb $A$ ? Specifically, let us hold $b$ fixed and consider the behavior of the problem $A \mapsto x = A^{-1}b$ when $A$ is perturbed by infinitesimal $\delta A$. Then $x$ must change by infinitesimal $\delta x$, where

$$(A + \delta A)(x + \delta x) = b.$$

Using the equality $Ax = b$ and dropping the doubly infinitesimal term $(\delta A)(\delta x)$, we obtain $(\delta A)x + A(\delta x) = 0$, that is, $\delta x = -A^{-1}(\delta A)x$. This equation implies $\|\delta x\| \leqslant \|A^{-1}\|\,\|\delta A\|\|x\|$, or equivalently,

$$\frac{\|\delta x\|}{\|x\|}\Big/\frac{\|\delta A\|}{\|A\|} \leqslant \left\|A^{-1}\right\|\|A\| = \kappa(A).$$

Equality in this bound will hold whenever $\delta A$ is such that

$$\left\|A^{-1}(\delta A)x\right\| = \left\|A^{-1}\right\|\|\delta A\|\|x\|,$$

and it can be shown by the use of dual norms that for any $A$ and $b$ and norm $\|\cdot\|$, such perturbations $\delta A$ exist. This leads us to the following result.

**Theorem 4.8.**
Let $b$ be fixed and consider the problem of computing $x = A^{-1}b$, where $A$ is square and nonsingular. The condition number of this problem with respect to perturbations in $A$ is

$$\kappa = \|A\|\left\|A^{-1}\right\| = \kappa(A). \tag{4.3}$$

**Note 4.9.**
Theorems 4.6 and 4.8 are of fundamental importance in numerical linear algebra, for they determine how accurately one can solve systems of equations. If a problem $Ax = b$ contains an ill-conditioned matrix $A$, one must always expect to "lose $\log_{10}\kappa(A)$ digits" in computing the solution, except under very special circumstances.

CHAPTER 5

# FLOATING POINT ARITHMETIC

It did not take long after the invention of computers for consensus to emerge on the right way to represent real numbers on a digital machine. The secret is floating point arithmetic, the hardware analogue of scientific notation. Before we can begin to study the accuracy of the algorithms of numerical linear algebra, we must examine this topic.

## 5.1 Limitations of Digital Representations

Using a finite number of bits to represent a number presents two problems:

- The represented numbers cannot be arbitrarily large or small;
- There must be gaps between them.

Modern computers represent numbers sufficiently large and small that the first constraint rarely poses difficulties. For example, the widely used IEEE double precision arithmetic permits numbers as large as $1.79 \times 10^{308}$ and as small as $2.23 \times 10^{-308}$, a range great enough for most of the problems considered in this book. In other words, **overflow** and **underflow** are usually not a serious hazard (but watch out if you are asked to evaluate a determinant!).

By contrast, the problem of gaps between represented numbers is a concern throughout scientific computing. For example, in IEEE double precision arithmetic, the interval $[1, 2]$ is represented by the discrete subset

$$1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, 1 + 3 \times 2^{-52}, \dots, 2. \tag{5.1}$$

The interval $[2, 4]$ is represented by the same numbers multiplied by 2 ,

$$2, 2 + 2^{-51}, 2 + 2 \times 2^{-51}, 2 + 3 \times 2^{-51}, \dots, 4,$$

and in general, the interval $[2^j, 2^{j+1}]$ is represented by Equation 5.1 times $2^j$. Thus in IEEE double precision arithmetic, the gaps between adjacent numbers are in a relative sense never larger than $2^{-52} \approx 2.22 \times 10^{-16}$. This may seem negligible, and so it is for most purposes if one uses stable algorithms (see the next chapter). But it is surprising how many carelessly constructed algorithms turn out to be unstable!

## 5.2 Floating Point Numbers

IEEE arithmetic is an example of an arithmetic system based on a floating point representation of the real numbers. This is the universal practice on general purpose computers nowadays. In a floating point number system, the position of the decimal (or binary) point is stored separately from the digits, and the gaps between adjacent represented numbers scale in proportion to the size of the numbers. This is distinguished from a **fixed point** representation, where the gaps are all of the same size.

Specifically, let us consider an idealized floating point number system defined as follows. The system consists of a discrete subset $\mathbf{F}$ of the real numbers $\mathbb{R}$ determined by an integer $\beta \geqslant 2$ known as the base or radix (typically 2 ) and an integer $t \geqslant 1$ known as the precision (24 and 53 for IEEE single and double precision, respectively). The elements of $\mathbf{F}$ are the number 0 together with all numbers of the form

$$x = \pm\left(m/\beta^t\right)\beta^e$$

where $m$ is an integer in the range $1 \leqslant m \leqslant \beta^t$ and $e$ is an arbitrary integer. Equivalently, we can restrict the range to $\beta^{t-1} \leqslant m \leqslant \beta^t - 1$ and thereby make the choice of $m$ unique. The quantity $\pm\left(m/\beta^t\right)$ is then known as the fraction or mantissa of $x$, and $e$ is the exponent. In other words:

$$\mathbf{F} = \{\pm\left(m/\beta^t\right)\beta^e | e \in \mathbb{Z}, 1 \leqslant m \leqslant \beta^t\}.$$

Our floating point number system is idealized in that it ignores over- and underflow. As a result, F is a countably infinite set, and it is self-similar: $\mathbf{F} = \beta\mathbf{F}$.

## 5.3 Machine Epsilon

The resolution of $\mathbf{F}$ is traditionally summarized by a number known as machine epsilon. Provisionally, let us define this number by

$$\epsilon_{\text{machine}} = \frac{1}{2}\beta^{1-t}. \tag{5.2}$$

This number is half the distance between 1 and the next larger floating point number. In a relative sense, this is as large as the gaps between floating point numbers get. That is, $\epsilon_{\text{machine}}$ has the following property:

**Proposition 5.1 (Property of machine epsilon).**
For all $x \in \mathbb{R}$, there exists $x' \in \mathbf{F}$ such that $|x - x'| \leqslant \epsilon_{\text{machine}}|x|$.

For the values of $\beta$ and $t$ common on various computers, $\epsilon_{\text{machine}}$ usually lies between $10^{-6}$ and $10^{-35}$. In IEEE single and double precision arithmetic, $\epsilon_{\text{machine}}$ is specified to be $2^{-24} \approx 5.96 \times 10^{-8}$ and $2^{-53} \approx 1.11 \times 10^{-16}$, respectively.

Let $\text{fl} : \mathbb{R} \to \mathbf{F}$ be a function giving the closest floating point approximation to a real number, its rounded equivalent in the floating point system. Then, the Proposition 5.1 can be rewritten as:

**Proposition 5.2.**
For all $x \in \mathbb{R}$, there exists $\epsilon$ with $|\epsilon| \leqslant \epsilon_{\text{machine}}$ such that

$$\text{fl}(x) = x(1+\epsilon). \tag{5.3}$$

## 5.4 Floating Point Arithmetic

It is not enough to represent real numbers, of course; one must compute with them. On a computer, all mathematical computations are reduced to certain elementary arithmetic operations, of which the classical set is,,$+ - \times$, and $\div$. Mathematically, these symbols represent operations on $\mathbb{R}$. On a computer, they have analogues that are operations on F. It is common practice to denote these floating point operations by $\oplus, \ominus, \otimes$, and $\oslash$.

A computer might be built on the following design principle. Let $x$ and $y$ be arbitrary floating point numbers, that is, $x, y \in \mathbf{F}$. Let $*$ be one of the operations,,$+ - x$, or $\div$, and let $\circledast$ be its floating point analogue. Then $x \circledast y$ must be given exactly by

$$x \circledast y = \text{fl}(x * y).$$

Hence, we have

**Theorem 5.3** (Fundamental Axiom of FPA).
For all $x, y \in \mathbf{F}$, there exists $\epsilon$ with $|\epsilon| \leqslant \epsilon_{\text{machine}}$ such that

$$x \circledast y = (x * y)(1 + \epsilon). \tag{5.4}$$

## 5.5 Complex Floating Point Arithmetic

Floating point complex numbers are generally represented as pairs of floating point real numbers, and the elementary operations upon them are computed by reduction to real and imaginary parts. The result is that the axiom is valid for complex as well as real floating point numbers, except that for $\otimes$ and $\Theta$, $\epsilon_{\text{machine}}$ must be enlarged from Equation 5.2 by factors on the order of $2^{3/2}$ and $2^{5/2}$, respectively. Once $\epsilon_{\text{machine}}$ is adjusted in this manner, rounding error analysis for complex numbers can proceed just as for real numbers.

# CHAPTER 6 STABILITY

It would be a fine thing if numerical algorithms could provide exact solutions to numerical problems. Since the problems are continuous while digital computers are discrete, however, this is generally not possible. The notion of stability is the standard way of characterizing what is possible-numerical analysts' idea of what it means to get the "right answer," even if it is not exact.

## 6.1 Algorithms

Recall Definition 4.1, we can define algorithms.

**Definition 6.1** (Algorithm).
An algorithm can be viewed as another map $\tilde{f} : X \to Y$ between the same two spaces. In detail, we should notice that $f$ is a composition of fl and another function $g$ i.e., $f(x) = g(\mathrm{fl}(x))$. And $f$ maps $X$ to $\mathrm{fl}(Y)$.

Hence, $\tilde{f}$ will be affected by rounding errors. Depending on the circumstances, it may also be affected by all kinds of other complications such as convergence tolerances or even the other jobs running on the computer, in cases where the assignment of computations to processors is not determined until runtime.

## 6.2 Accuracy

Except in trivial cases, $\tilde{f}$ cannot be continuous. Nevertheless, a good algorithm should approximate the associated problem $f$. To make this idea quantitative, we may consider the **absolute error** of a computation, $\|\tilde{f}(x) - f(x)\|$, or the **relative error**,

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \tag{6.1}$$

In this book we mainly utilize relative quantities, and thus (14.1) will be our standard error measure.

If $\tilde{f}$ is a good algorithm, one might naturally expect the relative error to be small, of order $\epsilon_{\text{machine}}$. One might say that an algorithm $\tilde{f}$ for a problem $f$ is **accurate** if for each $x \in X$,

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = O\left(\epsilon_{\text{machine}}\right) \tag{6.2}$$

## 6.3 Stability

If the problem $f$ is ill-conditioned, however, the goal of accuracy as defined by Equation 6.2 is unreasonably ambitious. Rounding of the input data is unavoidable on a digital computer, and even if all the subsequent computations could be carried out perfectly, this perturbation alone might lead to a significant change in the result. Instead of aiming for accuracy in all cases, the most it is appropriate to aim for in general is stability.

**Definition 6.2 (Stability).**
We say that an algorithm $\tilde{f}$ for a problem $f$ is stable if for each $x \in X$,

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O\left(\epsilon_{\text{machine}}\right), \tag{6.3}$$

for some $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O\left(\epsilon_{\text{machine}}\right). \tag{6.4}$$

In words,

A stable algorithm gives nearly the right answer to nearly the right question.

The motivation for this definition will become clear in the next chapter. We caution the reader that whereas the definitions of stability given here are useful in many parts of numerical linear algebra, the condition $O\left(\epsilon_{\text{machine}}\right)$ is probably too strict to be appropriate for all numerical problems in other areas such as differential equations.

## 6.4 Backward Stability

More algorithms of numerical linear algebra satisfy a condition that is both stronger and simpler than stability.

**Definition 6.3 (Backward stable).**
We say that an algorithm $\tilde{f}$ for a problem $f$ is backward stable if for each $x \in X$,

$$\tilde{f}(x) = f(\tilde{x}) \text{ for some } \tilde{x} \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = O\left(\epsilon_{\text{machine}}\right). \tag{6.5}$$

In words,

A backward stable algorithm gives exactly the right answer to nearly the right question.

Examples are given in the next chapter.

## 6.5 Independence of Norm

Our definitions involving $O\left(\epsilon_{\text{machine}}\right)$ have the convenient property that, provided $X$ and $Y$ are finite-dimensional, they are norm-independent.

**Theorem 6.4.**
For problems $f$ and algorithms $\tilde{f}$ defined on finite-dimensional spaces $X$ and $Y$, the properties of accuracy, stability, and backward stability all hold or fail to hold independently of the choice of norms in $X$ and $Y$.

# CHAPTER 7 MORE ON STABILITY

We continue the discussion of stability by considering examples of stable and unstable algorithms. Then we discuss a fundamental idea linking conditioning and stability, whose power has been proved in innumerable applications since the 1950s: backward error analysis.

## 7.1 Stability of Floating Point Arithmetic

The four simplest computational problems are,,$+ - x$, and $\div$ There is not much to say about choice of algorithms! Of course, we shall normally use the floating point operations $\oplus, \ominus, \otimes$, and $\oplus$ provided with the computer. As it happens, the axioms (5.4) and (5.3) imply that these four canonical examples of algorithms are all backward stable.

Now we consider $\ominus$ here. For data $x = (x_1, x_2)^* \in X$, the subtraction is $f(x_1, x_2) = x_1 - x_2$, and the algorithm is:

$$\tilde{f}(x_1, x_2) = \mathrm{fl}(x_1) \ominus \mathrm{fl}(x_2).$$

Hence, we have

$$\mathrm{fl}(x_1) = x_1(1 + \epsilon_1), \quad \mathrm{fl}(x_2) = x_2(1 + \epsilon_2),$$

for some $|\epsilon_1|, |\epsilon_2| \leqslant \epsilon_{\text{epsilon}}$. By Equation 5.4, we have

$$\mathrm{fl}(x_1) \ominus \mathrm{fl}(x_2) = (\mathrm{fl}(x_1) - \mathrm{fl}(x_2))\,(1 + \epsilon_3),$$

for some $|\epsilon_3| \leqslant \epsilon_{\text{machine}}$. Combine these, we have

$$\begin{aligned}
\mathrm{fl}(x_1) \ominus \mathrm{fl}(x_2) &= [x_1(1 + \epsilon_1) - x_2(1 + \epsilon_2)]\,(1 + \epsilon_3) \\
&= x_1(1 + \epsilon_1)(1 + \epsilon_3) - x_2(1 + \epsilon_2)(1 + \epsilon_3) \\
&= x_1(1 + \epsilon_4) - x_2(1 + \epsilon_5)
\end{aligned}$$

for some $|\epsilon_4|, |\epsilon_4| \leqslant 2\epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2)$. Hence, $\tilde{f}(x) = \mathrm{fl}(x_1) \ominus \mathrm{fl}(x_2)$ is exactly equal to $\tilde{x}_1 - \tilde{x}_2$, where $\tilde{x}_1$ and $\tilde{x}_2$ satisfy

$$\frac{|\tilde{x}_1 - x_1|}{|x_1|} = O\left(\epsilon_{\text{machine}}\right), \quad \frac{|\tilde{x}_2 - x_2|}{|x_2|} = O\left(\epsilon_{\text{machine}}\right).$$

**Example 7.1.**
- Inner product is back stable.
- Outer product is stable but not back stable.
- $x + 1$ with $\oplus$ is stable but not back stable.
- sin and cos are stable but not back stable.

- If the derivate of function is equal to zero at certain points, then it's not back stable. This is because a small change in the value will lead to a big change in the variables.

## 7.2 An Unstable Algorithm

Here is a more substantial example: the use of the characteristic polynomial to find the eigenvalues of a matrix. Given a matrix $A$, one method to compute the eigenvalues is:

- Find the coefficients of the characteristic polynomial,
- Find the roots of the characteristic polynomial.

This scheme is not only backward unstable but unstable, and it should not be used. The instability is revealed in the rootfinding of the second step. As we saw in Example 4.5, the problem of finding the roots of a polynomial is generally ill-conditioned. It follows that small errors in the coefficients of the characteristic polynomial will tend to be amplified when finding roots, even if the rootfinding is done to perfect accuracy.

For example, suppose $A = I$, the $2 \times 2$ identity matrix. The eigenvalues of $A$ are insensitive to perturbations of the entries, and a stable algorithm should be able to compute them with errors $O(\epsilon_{\text{machine}})$. However, the algorithm described above produces errors on the order of $\sqrt{\epsilon_{\text{machine}}}$. To explain this, we note that the characteristic polynomial is $x^2 - 2x + 1$, just as in Example 4.5. When the coefficients of this polynomial are computed, they can be expected to have errors on the order of $\epsilon_{\text{machine}}$, and these can cause the roots to change by order $\sqrt{\epsilon_{\text{machine}}}$. For example, if $\epsilon_{\text{machine}} = 10^{-16}$, the roots of the computed characteristic polynomial can be perturbed from the actual eigenvalues by approximately $10^{-8}$, a loss of eight digits of accuracy.

Before you try this computation for yourself, we must be a little more honest. If you use the algorithm just described to compute the eigenvalues of the $2 \times 2$ identity matrix, you will probably find that there are no errors at all, because the coefficients and roots of $x^2 - 2x + 1$ are small integers that will be represented exactly on your computer. However, if the experiment is done on a slightly perturbed matrix, such as

$$A = \begin{bmatrix} 1 + 10^{-14} & 0 \\ 0 & 1 \end{bmatrix},$$

the computed eigenvalues will differ from the actual ones by the expected order $\sqrt{\epsilon_{\text{machine}}}$. Try it!

## 7.3 Accuracy of a Backward Stable Algo

Suppose we have backward stable algorithm $\tilde{f}$ for a problem $f : X \to Y$, the accuracy depends on the condition number $\kappa = \kappa(x)$ of $f$. If $\kappa(x)$ is small, the result will be accurate in a relative sense.

**Theorem 7.2 (Accuracy of a back stap algo).**
Suppose a backward stable algorithm is applied to solve a problem $f : X \to Y$ with condition number $\kappa$ on a computer satisfying the axioms (5.3) and (5.4). Then the relative errors satisfy

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = O(\kappa(x)\epsilon_{\text{machine}}). \tag{7.1}$$

*Proof.*
By the Definition 6.3 of backward stability, we have $\tilde{f}(x) = f(\tilde{x})$ for some $\tilde{x} \in X$ satisfying

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{\text{machine}}).$$

By the Definition 4.3, this implies

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \leqslant (\kappa(x) + o(1))\frac{\|\tilde{x} - x\|}{\|x\|}.$$

Combine this, we can prove the theorem. □

## 7.4 Backward Error Analysis

The process we have just carried out in proving Theorem 7.2 is known as **backward error analysis**. We obtained an accuracy estimate by two steps. One step was to investigate the condition of the problem. The other was to investigate the stability of the algorithm. Our conclusion was that if the algorithm is stable, then the final accuracy reflects that condition number.

Mathematically, this is straightforward, but it is certainly not the first idea an unprepared person would think of if called upon to analyze a numerical algorithm. The first idea would be **forward error analysis**. Here, the rounding errors introduced at each step of the calculation are estimated, and somehow, a total is maintained of how they may compound from step to step.

Experience has shown that for most of the algorithms of numerical linear algebra, forward error analysis is harder to carry out than backward error analysis. With the benefit of hindsight, it is not hard to explain why this is so. Suppose a tried-and-true algorithm is used, say, to solve $Ax = b$ on a computer. It is an established fact (see Chapter 22) that the results obtained will be consistently less accurate when $A$ is ill-conditioned. Now, how could a forward error analysis capture this phenomenon? The condition number of $A$ is so global a property as to be more or less invisible at the level of the individual floating point operations involved in solving $Ax = b$. (We dramatize this by an example in the next lecture.) Yet one way or another, the forward analysis will have to detect that condition number if it is to end up with a correct result.

In short, it is an established fact that the best algorithms for most problems do no better, in general, than to compute exact solutions for slightly perturbed data. Backward error analysis is a method of reasoning fitted neatly to this backward reality.

# CHAPTER 8 STABILITY OF HOUSEHOLDER TRAINGULARIZATION

In this lecture we see backward error analysis in action. First we observe in a MATLAB experiment the remarkable phenomenon of backward stability of Householder triangularization. We then consider how the triangularization step can be combined with other backward stable pieces to obtain a stable algorithm for solving $Ax = b$.

## 8.1 Experiment

Householder factorization is a backward stable algorithm for computing QR factorizations. We can illustrate this by a MATLAB experiment carried out in IEEE double precision arithmetic, $\epsilon_{\text{machine}} \approx 1.11 \times 10^{-16}$.

```
R = triu(randn(50));           Set R to a 50 × 50 upper-triangular matrix with normal random entries.
[Q,X] = qr(randn(50));        Set Q to a random orthogonal matrix by orthogonalizing a random matrix.
A=Q*R;                                                  Set A to the product QR, up to rounding errors.
[Q2,R2] = qr(A);                 Compute QR factorization A ≈ Q2R2 by Householder traingularization.
```

The purpose of these four lines of MATLAB is to construct a matrix with a known QR factorization, $A = QR$, which can then be compared with the QR factorization $A = Q_2R_2$ computed by Householder triangularization. Actually, because of rounding errors, the QR factors of the computed matrix $A$ are not exactly $Q$ and $R$. However, for the purposes of this experiment they are close enough. The results about to be presented would not be significantly different if $A$ were exactly equal to $QR$ (which we could achieve, in effect, by calculating $A = QR$ in higher precision arithmetic on the computer).

For $Q_2$ and $R_2$, as it happens, are very far from exact:

```
norm(Q2-Q)                                                                  How accurate is Q2?
   ans = 0.00889
norm(R2-R)/norm(R)                                                          How accurate is R2?
   ans = 0.00071
```

These errors are huge! Our calculations have been done with sixteen digits of accuracy, yet the final results are accurate to only two or three digits. The individual rounding errors have been amplified by factors on the order of $10^{13}$. (Note that the computed $Q_2$ is close enough to $Q$ to indicate that changes in the signs of the columns cannot be responsible for any of the errors. If you try this experiment and get entirely different results, it may be that you need to multiply the columns of $Q$ and rows of $R$ by appropriate factors $\pm 1$ .) We seem to have lost twelve digits of accuracy. But now, an astonishing thing happens when we multiply these inaccurate matrices $Q_2$ and $R_2$:

```
norm(A-Q2*R2)/norm(A)                                                     How accurate is Q2R2?
   ans = 1.432e-15
```

The product $Q_2R_2$ is accurate to a full fifteen digits! The errors in $Q_2$ and $R_2$ must be "diabolically correlated," as Wilkinson used to say. To one part in $10^{12}$, they cancel out in the product $Q_2R_2$.

To highlight how special this accuracy of $Q_2R_2$ is, let us construct another pair of matrices $Q_3$ and $R_3$ that are equally accurate approximations to $Q$ and $R$, and multiply them.

```
Q3 = Q+1e-4*randn(50);          Set Q_3 to a random perturbation of Q
                                that is closer to Q than Q_2 is.
R3 = R+1e-4*randn(50);          Set R_3 to a random perturbation of R
                                that is closer to R than R_2 is.
norm(A-Q3*R3)/norm(A)                         How accurate is Q_3R_3?
   ans = 0.00088
```

This time, the error in the product is huge. $Q_2$ is no better than $Q_3$, and $R_2$ is no better than $R_3$, but $Q_2R_2$ is twelve orders of magnitude better than $Q_3R_3$.

In this experiment, we did not take the trouble to make $R_3$ upper-triangular or $Q_3$ orthogonal, but there would have been little difference had we done so.

The errors in $Q_2$ and $R_2$ are **forward errors**. In general, a large forward error can be the result of an ill-conditioned problem or an unstable algorithm (Theorem 7.2). In our experiment, they are due to the former. As a rule, the sequences of column spaces of a random triangular matrix are exceedingly ill-conditioned as a function of the entries of the matrix.

The error in $Q_2R_2$ is the **backward error** or **residual**. The smallness of this error suggests that Householder triangularization is backward stable.

## 8.2 Theorem

In fact, Householder triangularization is backward stable for all matrices $A$ and all computers satisfying (5.3) and (5.4). We shall now state a theorem to this effect. Our result will take the form

$$\tilde{Q}\tilde{R} = A + \delta A,$$

where $\delta A$ is small. In words, the computed $Q$ times the computed $R$ equals a small perturbation of the given matrix $A$. Note that

- By $\tilde{R}$, we mean the upper-triangular matrix that is constructed by Householder traingularization in floating point arithmetic.
- By $\tilde{Q}$, we mean a special unitary matrix. Recall that $Q = Q_1Q_2\cdots Q_n$, where $Q_k$ is defined by vector $v_k$. In the floating point computation, we obtain a sequence of vectors $\tilde{v}_k$. Let $\tilde{Q}_k$ be the exactly unitary reflector defined by the $\tilde{v}_k$. We define $\tilde{Q} = \tilde{Q}_1\tilde{Q}_2\cdots\tilde{Q}_n$.

Then we have the following result:

**Theorem 8.1 (Back stap of Householder).**
Let the QR factorization $A = QR$ of a matrix $A \in \mathbb{C}^{m\times n}$ be computed by Householder triangularization on a computer satisfying the axioms (5.3) and (5.4), and let the computed factors $\tilde{Q}$ and $\tilde{R}$ be defined as indicated above. Then, we have

$$\tilde{Q}\tilde{R} = A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}) \tag{8.1}$$

for some $\delta A \in \mathbb{C}^{m\times n}$.

## 8.3 Analyzing an Algorithm to Solve $Ax = b$

We have seen that Householder is backward stable but not always accurate in the forward sense. Now, QR factorization is generally not an end in itself, but a means to other ends such as solution of a system of equations, a least square problem, or an eigenvalue problem. The happy fact is that accuracy of $QR$ is indeed enough for most of purposes. We can show this by surprisingly simple arguments.

The example we shall consider is the use of Householder triangularization to solve a nonsingular $m \times m$ linear system $Ax = b$. The idea was discussed at the end of Chapter 7 in algorithm 3.2.

This algorithm is backward stable, proving this is straightforward, given that each of the three steps is itself backward stable. Here, we shall state backward stability results for the three steps, without proof, and then give the details of how they can be combined.

The first step of algorithm 3.2 is QR factorization of $A$, leading to computed matrices $\tilde{R}$ and $\tilde{Q}$. The backward stability of this process has already been expressed by Equation 8.1.

The second step is computation of $\tilde{Q}^* b$ by algorithm 2.2. When $\tilde{Q}^* b$ is computed, rounding errors will be made, so the result will not be exactly $\tilde{Q}^* b$. Instead it will be some vector $\tilde{y}$. It can be shown that this vector satisfies the following backward stability estimate:

$$(\tilde{Q} + \delta Q)\tilde{y} = b, \quad \|\delta Q\| = O(\epsilon_{\text{machine}}). \tag{8.2}$$

Hence, the result of applying the Householder reflectors in floating point arithmetic is exactly equivalent to multiplying $b$ by a slightly perturbed matrix, $(\tilde{Q} + \delta Q)^{-1}$.

The final step is back substitution to compute $\tilde{R}^{-1}\tilde{y}$. In this step new rounding errors will be introduced, but once more, the computation is backward stable. This time the estimate takes the form:

$$(\tilde{R} + \delta R)\tilde{x} = \tilde{y}, \quad \frac{\|\delta R\|}{\|\tilde{R}\|} = O(\epsilon_{\text{machine}}). \tag{8.3}$$

As always, the equality on the left is exact. We will derive this in full detail in the next chapter. Now, combine (8.1) (8.2) and (8.3), we get the following theorem.

**Theorem 8.2 (Backstap of linear system).**
algorithm 3.2 is backward stable, satisfying

$$(A + \Delta A)\tilde{x} = b, \quad \frac{\|\Delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}) \tag{8.4}$$

for some $\Delta A \in \mathbb{C}^{m \times m}$.

*Proof.*
Combine (8.3) and (8.2), we have

$$b = (\tilde{Q} + \delta Q)(\tilde{R} + \delta R)\tilde{x} = [\tilde{Q}\tilde{R} + (\delta Q)\tilde{R} + \tilde{Q}(\delta R) + (\delta Q)(\delta R)]\tilde{x}.$$

Hence, by (8.1),

$$b = [A + \delta A + (\delta Q)\tilde{R} + \tilde{Q}(\delta R) + (\delta Q)(\delta R)]\tilde{x}.$$

So $\Delta A = \delta A + (\delta Q)\tilde{R} + \tilde{Q}(\delta R) + (\delta Q)(\delta R)$.

Since $\tilde{Q}\tilde{R} = A + \delta A$ and $\tilde{Q}$ is unitary, we have

$$\frac{\|\tilde{R}\|}{\|A\|} \leqslant \left\|\tilde{Q}^*\right\| \frac{\|A + \delta A\|}{\|A\|} = O(1)$$

as $\epsilon_{\text{machine}} \to 0$, by (8.1). (It is $1 + O(\epsilon_{\text{machine}})$ if $\|\cdot\| = \|\cdot\|_2$, but we have made no assumptions about $\|\cdot\|$.) This gives us

$$\frac{\|(\delta Q)\tilde{R}\|}{\|A\|} \leqslant \|\delta Q\| \frac{\|\tilde{R}\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

by (8.2). Similarly,

$$\frac{\|\tilde{Q}(\delta R)\|}{\|A\|} \leqslant \|\tilde{Q}\| \frac{\|\delta R\|}{\|\tilde{R}\|} \frac{\|\tilde{R}\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

by (8.3). Finally,

$$\frac{\|(\delta Q)(\delta R)\|}{\|A\|} \leqslant \|\delta Q\| \frac{\|\delta R\|}{\|A\|} = O\left(\epsilon_{\text{machine}}^2\right)$$

The total perturbation $\Delta A$ thus satisfies

$$\frac{\|\Delta A\|}{\|A\|} \leqslant \frac{\|\delta A\|}{\|A\|} + \frac{\|(\delta Q)\tilde{R}\|}{\|A\|} + \frac{\|\tilde{Q}(\delta R)\|}{\|A\|} + \frac{\|(\delta Q)(\delta R)\|}{\|A\|} = O\left(\epsilon_{\text{machine}}\right),$$

as claimed. □

Combining Theorem 4.8, Theorem 7.2 and Theorem 8.2 gives the following result about accuracy of solutions of $Ax = b$.

**Theorem 8.3** (Error of LS).
The solution $\tilde{x}$ computed by algorithm 3.2 satisfies

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\kappa(A)\epsilon_{\text{machine}}). \tag{8.5}$$

# CHAPTER 9 STABILITY OF BACK SUBSTITUTION

One of the easiest problems of numerical linear algebra is the solution of a triangular system of equations. The standard algorithm is successive substitution, called back substitution when the system is upper-triangular. Here we show in full detail that this algorithm is backward stable, obtaining quantitative bounds on the effects of rounding errors, with no "$O(\epsilon_{\text{machine}})$".

## 9.1 Triangular System

We have seen that a general system of equations $Ax = b$ can be reduced to an upper-triangular system $Rx = y$ by QR factorization. Lower- and upper- triangular systems also arise in Gaussian elimination, in Cholesky factorization, and in numerous other computations of numerical linear algebra.

These systems are easily solved by a process of successive substitution, called **forward substitution** if the system is lower-triangular and **back substitution** if it's upper-triangular.

Suppose we wish to solve $Rx = b$, that is,

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ & r_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & r_{mm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}, \tag{9.1}$$

where $b \in \mathbb{C}^m$ and $R \in \mathbb{C}^{m\times m}$ and $x \in \mathbb{C}^m$ is unknown. We can solve this one after another, beginning with $x_m$ and finishing with $x_1$.

**Algorithm 9.1:** Back Substitution

```
1 $x_m = b_m / r_{mm}$;
2 $x_{m-1} = (b_{m-1} - x_m r_{m-1,m}) / r_{m-1,m-1}$;
3 $x_{m-2} = (b_{m-2} - x_{m-1} r_{m-2,m-1} - x_m r_{m-2,m}) / r_{m-2,m-2}$;
4 $\vdots$ ;
5 $x_j = (b_j - \sum_{k=j+1}^{m} x_k r_{jk}) / r_{jj}$
```

The structure is triangular, with a subtraction and multiplication at each position. The operation count is accordingly twice the area of an $m \times m$ triangle.

**Corollary 9.1.**
The work for back substitution: $\sim m^2$ flops.

## 9.2 Backward Stability Theorem

Now we will try to show (8.3), and this will be the only case in this book in which we give all the details of such a proof.

Before the proof, we must pin down one detail of the algorithm. Let us decide, arbitrarily, that in the expressions in parentheses above, the subtractions will be carried out from left to right.

**Theorem 9.2 (Backstap of backsub).**
Let algorithm 9.1 be applied to a problem (9.1) consisting of floating point numbers on a computer satisfying (5.4). This algorithm is backward stable in the sense that the computed solution $\tilde{x} \in \mathbb{C}^m$ satisfies

$$(R + \delta R)\tilde{x} = b \tag{9.2}$$

for some upper-triangular $\delta R \in \mathbb{C}^{m\times m}$ with

$$\frac{\|\delta R\|}{\|R\|} = O(\epsilon_{\text{machine}}). \tag{9.3}$$

Specifically, for each $i, j$,

$$\frac{|\delta r_{ij}|}{|r_{ij}|} \leqslant m\epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}). \tag{9.4}$$

To keep the ideas clear and interesting, our proof will be most leisurely.

## 9.3 $m = 1$

When $d = 1$, the problem is:

$$\tilde{x}_1 = b_1 \oplus r_{11}.$$

The axiom (5.4) for $\oplus$ guarantees that:

$$\tilde{x}_1 = \frac{b_1}{r_{11}}(1+\epsilon_1), \quad |\epsilon_1| \leqslant \epsilon_{\text{machine}}.$$

However, we would like to express the error from a perturbation in $R$. To this end, we set $\epsilon'_1 = -\epsilon_1/(1+\epsilon_1)$, where the formula becomes,

$$\tilde{x}_1 = \frac{b_1}{r_{11}(1+\epsilon'_1)}, \quad |\epsilon'_1| \leqslant \epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}). \tag{9.5}$$

Hence, we have

$$(r_{11} + \delta r_{11})\tilde{x}_1 = b_1,$$

with $\Delta r_{11} = \epsilon'_1 r_{11}$. In other words,

$$\frac{|\delta r_{11}|}{|r_{11}|} \leqslant \epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}).$$

## 9.4 $m = 2$

The $2 \times 2$ case is slightly less trivial. Suppose we have an upper-triangular matrix $R \in \mathbb{C}^{2\times 2}$ and a vector $b \in \mathbb{C}^2$. We firstly, solve $\tilde{x}_2$ by:

$$\tilde{x}_2 = b_2 \oplus r_{22} = \frac{b_2}{r_{22}(1+\epsilon_1)}, \quad |\epsilon_1| \leqslant \epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}). \tag{9.6}$$

The second step is defined by the formula

$$\tilde{x}_1 = (b_1 \ominus (\tilde{x}_2 \otimes r_{12})) \oplus r_{11}.$$

In fact, if we apply (5.4), we have

$$\tilde{x}_1 = \frac{(b_1 - \tilde{x}_2 r_{12}(1+\epsilon_2))(1+\epsilon_3)}{r_{11}}(1+\epsilon_4),$$

where $|\epsilon_2|, |\epsilon_3|, |\epsilon_4| \leqslant \epsilon_{\text{machine}}$. Now we shift the $\epsilon_3$ and $\epsilon_4$ terms from the number to the denominator, we have,

$$\tilde{x}_1 = \frac{b_1 - \tilde{x}_2 r_{12}(1+\epsilon_2)}{r_{11}(1+\epsilon_3')(1+\epsilon_4')},$$

where $|\epsilon_3'|, |\epsilon_4'| \leqslant \epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2)$, or equivalently,

$$\tilde{x}_1 = \frac{b_1 - \tilde{x}_2 r_{12}(1+\epsilon_2)}{r_{11}(1+2\epsilon_5)}, \tag{9.7}$$

where $|\epsilon_5| \leqslant \epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2)$. Hence,

$$(R + \delta R)\tilde{x} = b,$$

where the entries $\delta_{ij}$ of $\delta R$ satisfy

$$\begin{pmatrix} |\delta r_{11}|/|r_{11}| & |\delta r_{12}|/|r_{12}| \\ & |\delta r_{22}|/|r_{22}| \end{pmatrix} = \begin{pmatrix} 2|\epsilon_5| & |\epsilon_2| \\ & |\epsilon_1| \end{pmatrix} \leqslant \begin{pmatrix} 2 & 1 \\ & 1 \end{pmatrix} \epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2).$$

Hence, the $2 \times 2$ back substitution is stable.

## 9.5 $m = 3$

The analysis for a $3 \times 3$ matrix includes all the reasoning necessary for the general case. The first two steps are the same as before:

$$\begin{aligned} \tilde{x}_3 &= b_3 \odot r_{33} = \frac{b_3}{r_{33}(1+\epsilon_1)} \\ \tilde{x}_2 &= (b_2 \ominus (\tilde{x}_3 \otimes r_{23})) \odot r_{22} = \frac{b_2 - \tilde{x}_3 r_{23}(1+\epsilon_2)}{r_{22}(1+2\epsilon_3)} \end{aligned}$$

where

$$\begin{bmatrix} 2|\epsilon_3| & |\epsilon_2| \\ & |\epsilon_1| \end{bmatrix} \leqslant \begin{bmatrix} 2 & 1 \\ & 1 \end{bmatrix} \epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2).$$

The third step involves the computation

$$\tilde{x}_1 = [(b_1 \ominus (\tilde{x}_2 \otimes r_{12})) \ominus (\tilde{x}_3 \otimes r_{13})] \odot r_{11}.$$

Firstly, we have

$$\tilde{x}_1 = [(b_1 - \tilde{x}_2 r_{12}(1+\epsilon_4))(1+\epsilon_6) - \tilde{x}_3 r_{13}(1+\epsilon_5)](1+\epsilon_7) \odot r_{11}.$$

The $\oplus$ is eliminated using $\epsilon_8$; let us immediately replace this by $\epsilon_8'$ with $|\epsilon_8| \leqslant \epsilon_{\text{machine}} + O(\epsilon_{\text{machine}}^2)$ and put the result in the denominator:

$$\tilde{x}_1 = \frac{[(b_1 - \tilde{x}_2 r_{12}(1+\epsilon_4))(1+\epsilon_6) - \tilde{x}_3 r_{13}(1+\epsilon_5)](1+\epsilon_7)}{r_{11}(1+\epsilon_8')}.$$

Now, the expression above has everything as we need it except the terms involving $\epsilon_6$ and $\epsilon_7$, which originated from operations $\Theta$. If these are distributed, they will affect the number $b_1$, whereas our aim is to perturb only the entries $r_{ij}$. The term involving $\epsilon_7$ is easily dispatched: we change $\epsilon_7$ to $\epsilon_7'$ and move it to the denominator as usual. The term involving $\epsilon_6$ requires a new trick. We move it to the denominator too, but to keep the equality valid, we compensate by putting the new factor $(1+\epsilon_6')$ into the $r_{13}$ term as well. Thus

$$\tilde{x}_1 = \frac{b_1 - \tilde{x}_2 r_{12}(1+\epsilon_4) - \tilde{x}_3 r_{13}(1+\epsilon_5)(1+\epsilon_6')}{r_{11}(1+\epsilon_6')(1+\epsilon_7')(1+\epsilon_8')}.$$

Now $r_{13}$ has two perturbations of size at most $\epsilon_{\text{machine}}$, and $r_{11}$ has three. In this formula, all the errors in the computation have been expressed as perturbations in the entries of $R$.

The result can be summarized as

$$(R+\delta R)\tilde{x} = b,$$

where the entries $\delta r_{ij}$ satisfy

$$\begin{bmatrix} |\delta r_{11}|/|r_{11}| & |\delta r_{12}|/|r_{12}| & |\delta r_{13}|/|r_{13}| \\ & |\delta r_{22}|/|r_{22}| & |\delta r_{23}|/|r_{23}| \\ & & |\delta r_{33}|/|r_{33}| \end{bmatrix} \leqslant \begin{bmatrix} 3 & 1 & 2 \\ & 2 & 1 \\ & & 1 \end{bmatrix} \epsilon_{\text{machine}} + O\left(\epsilon_{\text{machine}}^2\right).$$

## 9.6 General $m$

The analysis in higher-dimensional cases is similar. For example, in the $5\times 5$ case we obtain the componentwise bound

$$\frac{|\delta R|}{|R|} \leqslant \begin{bmatrix} 5 & 1 & 2 & 3 & 4 \\ & 4 & 1 & 2 & 3 \\ & & 3 & 1 & 2 \\ & & & 2 & 1 \\ & & & & 1 \end{bmatrix} \epsilon_{\text{machine}} + O\left(\epsilon_{\text{machine}}^2\right).$$

The entries of the matrix in this formula are obtained from three components. The multiplications $\tilde{x}_k r_{jk}$ introduce $\epsilon_{\text{machine}}$ perturbations in the pattern

$$\otimes : \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ & 0 & 1 & 1 & 1 \\ & & 0 & 1 & 1 \\ & & & 0 & 1 \\ & & & & 0 \end{bmatrix}.$$

The divisions by $r_{kk}$ introduce perturbations in the pattern

$$\oplus : \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}.$$

Finally, the subtractions also occur in the pattern (17.15), and, due to the decision to compute from left to right, each one introduces a perturbation on the diagonal and at each position to the right. This adds up to the pattern

$$\ominus : \begin{bmatrix} 4 & 0 & 1 & 2 & 3 \\ & 3 & 0 & 1 & 2 \\ & & 2 & 0 & 1 \\ & & & 1 & 0 \\ & & & & 0 \end{bmatrix}.$$

Adding them produces the result we want. This completes the proof of Theorem 9.2.

# CHAPTER 10 CONDITIONING OF LEAST SQUARES PROBLEMS

The conditioning of least squares problems is a subtle topic, combining the conditioning of square systems of equations with the geometry of orthogonal projection. It is important because it has nontrivial implications for the stability of least squares algorithms.

## 10.1 Four Conditioning Problems

We return to the linear least squares problems. We assume $\|\cdot\| = \|\cdot\|_2$ in this chapter and the matrix is of full rank:

$$\begin{gathered}\text{Given } A \in \mathbb{C}^{m\times n} \text{ of full rank, } m \geqslant n, b \in \mathbb{C}^m, \\ \text{find } x \in \mathbb{C}^n \text{ such that } \|b - Ax\| \text{ is minimized.}\end{gathered} \tag{10.1}$$

The solution is given by,

$$x = A^\dagger b, \quad y = Pb. \tag{10.2}$$

We consider the conditioning of (10.1) w.r.t. perturbations. Conditioning pertains to the sensitivity of solutions to perturbations in data. The data for the problem are matrix $A$ and the vector $b$. The solution can be $x$ or $y$. Thus,

$$\text{Data: } A, b, \qquad \text{Solution: } x, y.$$

Together, these two paris of choices defined four conditioning questions.

## 10.2 Theorem

The results are expressed in terms of three dimensionless parameters:

- Given a matrix $A$, the condition number $\kappa(A) = \|A\|\|A^\dagger\| = \frac{\sigma_1}{\sigma_n}$.
- The angle between $y$ and $b$: $\theta = \cos^{-1}\frac{\|y\|}{\|b\|}$ .
- How much $\|y\|$ falls short of its maximum possible value: $\eta = \frac{\|A\|\|x\|}{\|y\|} = \frac{\|A\|\|x\|}{\|Ax\|}$.

These parameters lie in the ranges:

$$1 \leqslant \kappa(A) < \infty, \quad 0 \leqslant \theta \leqslant \pi/2, \quad 1 \leqslant \eta \leqslant \kappa(A).$$

**Theorem 10.1 (Conditioning of LS).**
Let $b \in \mathbb{C}^m$ and $A \in \mathbb{C}^{m\times n}$ of full rank be fixed. The LS problem (10.1) has the following 2-norm condition numbers describing the sensitivities of $y$ and $x$ to perturbations in $b$ and $A$:

|   | $y$ | $x$ |
|---|---|---|
| $b$ | $\frac{1}{\cos\theta}$ | $\frac{\kappa(A)}{\eta\cos\theta}$ |
| $A$ | $\frac{\kappa(A)}{\cos\theta}$ | $\kappa(A)+\frac{\kappa(A)^2\tan\theta}{\eta}$ |

The results in the first row are exact, being attained for certain perturbations $\delta b$, and the results in the second row are upper bounds.

**Note 10.2.**
In the special case $m = n$, (10.1) reduces to a square, nonsingular system of equations, with $\theta = 0$. In this case, the numbers in the second column of the theorem reduces $\kappa(A)/\eta$ and $\kappa(A)$, which are the results (4.2) and (4.3) derived earlier.

## 10.3 Transformation to a Diagonal Matrix

Assume $A = U\Sigma V^*$, since perturbations are measures in the 2-norm, we can assume $A = \Sigma$ and write

$$A = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \\ & & & \\ & & & \end{bmatrix} = \begin{bmatrix} A_1 \\ 0 \end{bmatrix}.$$

Assume $b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, then the projection $y = Pb$ is $y = \begin{bmatrix} b_1 \\ 0 \end{bmatrix}$. Hence, $Ax = y$ is

$$\begin{bmatrix} A_1 \\ 0 \end{bmatrix} x = \begin{bmatrix} b_1 \\ 0 \end{bmatrix},$$

which implies $x = A_1^{-1}b_1$. It's easy to find that

$$P = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}, \quad A^\dagger = \begin{bmatrix} A_1^{-1} & 0 \end{bmatrix}.$$

## 10.4 Sensitivity of $y$ to Perturbations in $b$

Note that $y = Pb$ and the Jacobian is $P$ itself. Hence, the condition number of $y$ w.r.t. perturbations in $b$ is:

$$\kappa_{bmapstoy} = \frac{\|P\|}{\|y\|/\|b\|} = \frac{1}{\cos\theta}.$$

The condition number is realized when $\delta b$ is zero except in the first $n$ entries.

## 10.5 Sensitivity of $x$ to Perturbations in $b$

The relationship between $b$ and $x$ is $x = A^\dagger b$. Hence,

$$\kappa_{b\mapsto x} = \frac{\|A^\dagger\|}{\|x\|/\|b\|} = \|A^\dagger\|\frac{\|b\|}{\|y\|}\frac{\|y\|}{\|A\|\|x\|}\|A\| = \frac{\kappa(A)}{\eta\cos\theta}.$$

Here the condition number is realized by perturbation $\delta b$ sastisfying $\|A^\dagger(\delta b)\| = \|A^\dagger\|\|\delta b\| = \|\delta b\|/\sigma_n$, which means $\delta b = e_n$.

## 10.6 Tilting the range of $A$

The analysis of perturbations in $A$ is a nonlinear problem and more subtle. We could proceed by calculating Jacobians algebraically, but instead, we shall take a geometric view. Our starting point is the observation that perturbations in $A$ affect the least squares problem in two ways:

- They distort the mapping of $\mathbb{C}^n$ onto range$(A)$;
- They alter range$(A)$ itself.

Let us consider this latter effect now.

We can visualize slight changes in range$(A)$ as small "tiltings" of this space. The question is, **What is the maximum angle of tilt $\delta\alpha$ that can be imparted by a small perturbation $\delta A$?** The answer can be determined as follows. The image under $A$ of the unit $n$-sphere is a hyperellipse that lies flat in range$(A)$. To change range$(A)$ as efficiently as possible, we grasp a point $p = Av$ on the hyperellipse and nudge it in a direction $\delta p$ orthogonal to range$(A)$. A matrix perturbation that achieves this most efficient is $\delta A = (\delta p)v^*$, which gives $(\delta A)v = \delta p$ with $\|\delta A\| = \|\delta p\|$.

Now it's clear that to obtain the maximum tilt with a given $\|\delta p\|$, we should take $p$ to be as close to the origin as possible. That is, we want $p = \sigma_n u_n$, where $\sigma_n$ is the smallest singular value of $A$ and $u_n$ is the corresponding left singular vector. With $A$ in the diagonal form, $p$ is equal to the last column of $A$ and $v^*$ is the $n$-vector $(0, 0, \ldots, 0, 1)$, and $\delta A$ is a perturbation of the entries of $A$ below the diagonal in this column. Such a perturbation tilts range$(A)$ by the angle $\delta\alpha$ given by $\tan(\delta\alpha) = \|\delta p\|/\sigma_n$. Since $\|\delta p\| = \|\delta A\|$ and $\delta\alpha \leqslant \tan(\delta\alpha)$, we have

$$\delta\alpha \leqslant \frac{\|\delta A\|}{\sigma_n} = \frac{\|\delta A\|}{\|A\|}\kappa(A), \tag{10.3}$$

wit equality attained for choices $\delta A$ of this kind just described.

## 10.7 Sensitivity of $y$ to Perturbations in $A$

We begin with the left-hand entry in the second row of the table in Theorem 10.1. Since $y$ is the orthogonal projection of $b$ onto range$(A)$, it's determined by $b$ and range$(A)$ alone. Therefore, to analyze the sensitivity of $y$ to perturbations in $A$, we can simply study the effect on $y$ of tilting range$(A)$ by some angle $\delta\alpha$.

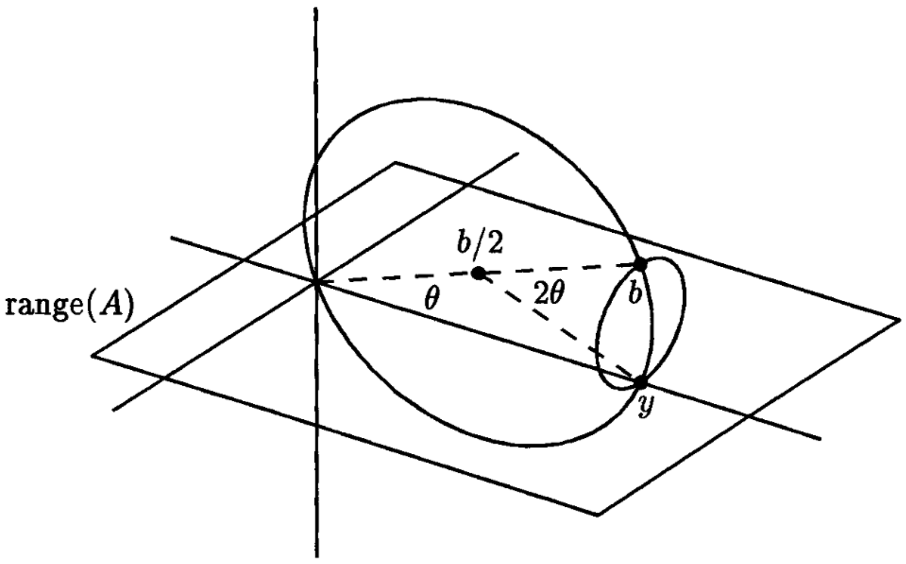


Figure 10.1: Two circles on the sphere along which $y$ moves as range$(A)$ varies. The large circle, of radius $\|b\|/2$, corresponds to tilting range$(A)$ in the plane $0 - b - y$, and the small circle of radius $(\|b\|/2)\sin\theta$, corresponds to tilting it in an orthogonal direction. However range$(A)$ is tilted, $y$ remains on the sphere of radius $\|b\|/2$ centered at $\frac{b}{2}$.

An elegant geometrical property becomes apparent when we imagine fixing $b$ and watching $y$ vary as range$(A)$ is tilted. No matter how range$(A)$ is tilted, the vector $y \in$ range$(A)$ must always be orthogonal to $y - b$. That is, the line $b - y$ must lie at right angles to the line $0 - y$. In other words, as range$(A)$ is adjusted, $y$ moves along the sphere of radius $\|b\|/2$ centered at the point $b/2$.

Tilting range$(A)$ in the plane $0-b-y$ by an angle $\delta\alpha$ changes the angle $2\theta$ at the central point $\frac{b}{2}$ by $2\delta\alpha$. Thus the corresponding perturbation $\delta y$ is the base of an isosceles triangle with central angle $2\delta\alpha$ and edge length $\|b\|/2$. This implies $\|\delta y\| = \|b\| \sin(\delta\alpha)$. Tilting range$(A)$ in any other direction results in a similar geometry in a different plane and perturbations smaller by a factor as small as $\sin\theta$. Thus for arbitrary perturbations by an angle $\delta\alpha$ we have

$$\|\delta y\| \leqslant \|b\| \sin(\delta\alpha) \leqslant \|b\|\delta\alpha. \tag{10.4}$$

By the definition of $\theta$ and (10.3), we have

$$\|\delta y\| \leqslant \frac{\|\delta A\|}{\|A\|}\kappa(A) \cdot \frac{\|y\|}{\cos\theta} \Rightarrow \frac{\|\delta y\|}{\|y\|} \Big/ \frac{\|\delta A\|}{\|A\|} \leqslant \frac{\kappa(A)}{\cos\theta}. \tag{10.5}$$

## 10.8 Sensitivity of $x$ to Perturbations in $A$

We are now ready to analyze the most interesting relationship of Theorem 10.1: the sensitivity of $x$ to perturbations in $A$. A perturbation $\delta A$ splits naturally into two parts: one part $\delta A_1$ in the fist $n$ rows of $A$, and another part $\delta A_2$ in the remaining $m - n$ rows:

$$\delta A = \begin{bmatrix} \delta A_1 \\ \delta A_2 \end{bmatrix} = \begin{bmatrix} \delta A_1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \delta A_2 \end{bmatrix}.$$

First, let us consider the effect of perturbations $\delta A_1$. Such a perturbation changes the mapping of $A$ in its range, but not range$(A)$ itself or $y$. It perturbs $A_1$ by $\delta A_1$ in the square system (10.1) without changing $b_1$. The condition for such perturbations is given by (4.3), which here takes the form

$$\frac{\|\delta x\|}{\|x\|} \Big/ \frac{\|\delta A_1\|}{\|A\|} \leqslant \kappa\left(A_1\right) = \kappa(A). \tag{10.6}$$

Next we consider the effect of perturbations $\delta A_2$. Such a perturbation tilts range$(A)$ without changing the mapping of $A$ within this space. The point $y$ and thus the vector $b_1$ are perturbed, but not $A_1$. This corresponding to perturbing $b_1$ without changing $A_1$. The condition number for such perturbation is given by (4.2), which takes the form

$$\frac{\|\delta x\|}{\|x\|} \Big/ \frac{\|\delta b_1\|}{\|b_1\|} \leqslant \frac{\kappa\left(A_1\right)}{\eta\left(A_1; x\right)} = \frac{\kappa(A)}{\eta}. \tag{10.7}$$

To finish the argument, we need to relate $\delta b_1$ to $\delta A_2$. Now the vector $b_1$ is $y$ expressed in the coordinates of range$(A)$. Therefore, the only changes in $y$ that realized as changes in $b_1$ are those that lie parallel to range$(A)$; orthogonal changes have no effect. In particular, if range$(A)$ is tilted by an angle $\delta\alpha$ in the plane $0 - b - y$, the resulting perturbation $\delta y$ lies not parallel to range $(A)$ but at an angle $\pi/2 - \theta$. Consequently, the change in $b_1$ satisfies $\|\delta b_1\| = \sin\theta\|\delta y\|$. By (10.4), we therefore have

$$\|\delta b_1\| \leqslant (\|b\|\delta\alpha)\sin\theta$$

Curiously, if range $(A)$ is tilted in a direction orthogonal to the plane $0-b-y$, we obtain the same bound, but for a different reason. Now $\delta y$ is parallel to range $(A)$, but it is a factor of $\sin\theta$ smaller, as discussed above in connection with Figure 18.2. Thus we have $\|\delta y\| \leqslant (\|b\|\delta\alpha)\sin\theta$, and since $\|\delta b_1\| \leqslant \|\delta y\|$, we again arrive at the same result.

All the pieces are now in place. Since $\|b_1\| = \|b\|\cos\theta$, we can rewrite before as

$$\frac{\|\delta b_1\|}{\|b_1\|} \leqslant (\delta\alpha)\tan\theta$$

Relating $\delta\alpha$ to $\|\delta A_2\|$ by (10.3) and combining the previous results, we obtain

$$\frac{\|\delta x\|}{\|x\|}\Big/\frac{\|\delta A_2\|}{\|A\|} \leqslant \frac{\kappa(A)^2 \tan\theta}{\eta}.$$

Adding this to (10.6) establishes the lower-right result of Theorem 10.1.

# CHAPTER 11 STABILITY OF LEAST SQUARES ALGORITHMS

Least squares problems can be solved by various methods, as described in Chapter 11, including the normal equations, Householder triangularization, GramSchmidt orthogonalization, and the SVD. Here we compare these methods and show that the use of the normal equations is in general unstable.

## 11.1 Example

We consider the following example. We will fit $\exp(\sin(4\tau))$ on the interval $[0, 1]$ by a polynomial of degree 14.

```
m=100; n=15;
t= (0:m-1)'/(m-1);                          Set t to a discretization of [0,1].
A = []; for i=1:n;                                 Construct Vandermonde matrix.
   A = [A t.^(i-1)]; end
b = exp(sin(4*t));                                             Right-hand side
b = b/2006.787453080206;                           Make the coefficient c_15 = 1
```

Now we check the quantities:

```
x = A\b; y = A*x;                                            Solve LS problem.
kappa = cond(A)
   kappa = kappa = 2.2718e+10                                             κ(A)
theta = asin(norm(b-y)/norm(b))
   theta = 3.7461e-06                                                        θ
eta = norm(A)*norm(x)/norm(y)
   eta = 2.1036e+05                                                          η
```

The result $\kappa(A) \approx 10^{10}$ indicates that the monomials $1, t, \ldots, t^{14}$ form a highly ill-conditioned basis. The result $\theta \approx 10^{-6}$ indicates that $\exp(\sin(4t))$ can be fitted very closely by a polynomial of degree 14 . (The fit is so close that we computed $\theta$ with the formula $\theta = \sin^{-1}(\|b - y\|/\|b\|)$, to avoid cancellation error.) As for $\eta$, its value of about $10^5$ is about midway between the extremes 1 and $\kappa(A)$.

We can get the following table:

| | $y$ | $x$ |
|---|---|---|
| $b$ | 1.0 | $1.1 \times 10^5$ |
| $A$ | $2.3 \times 10^{10}$ | $3.2 \times 10^{10}$ |

## 11.2 Householder Triangularization

The code is

```
[Q,R] = qr(A,0);                                      Householder triang. of A.
x = R\(Q'*b);                                                      Solve for x.
x(15)
   ans = 1.00000031528723
```

The relative error is $3 \times 10^{-7}$ and $\epsilon_{\text{machine}} = 10^{-16}$ and the rounding errors have been amplified by a factor of order $10^9$. It can be entirely explained by ill-conditioning. Hence, Algo 3.2 appears to be backward stable.

Note that we don't really have to store $\hat{Q}$ but the vectors $v_k$. Then, we can compute $\hat{Q}^* b$ by Algo 2.2. In MATLAB, we can achieve this effect by computing a QR factorization not just of $A$ but of the $m \times (n+1)$ "augmented" matrix $\begin{bmatrix} A & b \end{bmatrix}$. In the course of this factorization, the $n$ Householder reflectors that make $A$ upper-triangular are applied to $b$ also, leaving the vector $\hat{Q}^* b$ in the first $n$ positions of column $n+1$. An additional $(n+1)$ st reflector is then applied to make entries $n+2, \ldots, m$ of column $n+1$ zero, but this does not change the first $n$ entries of that column, which are the ones we care about.

```
[Q2, R2] = qr([A b], 0);                               Householder triang. of [Ab]
R2 = R2(1:n, 1:n);
Qb = R2(1:n, n+1);
x = R2\Qb;
x(15)
   ans = 1.00000031529465
```

The answer is the same.

A third way is to use the built-in operator \ via Householder triangularization.

```
x = A\b;
x(15)
   ans = 0.99999994311087
```

The result is better due to column pivoting!

We have the following theorem:

**Theorem 11.1** (Backstap of Householder for LS).
Let the full-rank least squares problem (3.1) be solved by Householder triangularization (Algorithm 3.2) on a computer satisfying (5.3) and (5.4). This algorithm is backward stable in the sense that the computed solution $\tilde{x}$ has the property

$$\|(A + \delta A)\tilde{x} - b\| = \min, \quad \frac{\|\delta A\|}{\|A\|} = O\left(\epsilon_{\text{machine}}\right) \tag{11.1}$$

for some $\delta A \in \mathbb{C}^{m \times n}$. This is true whether $\hat{Q}^* b$ is computed via explicit formation of $\hat{Q}$ or implicitly by Algorithm 2.2. It also holds for Householder triangularization with arbitrary column pivoting.

## 11.3 GS Orthogonalization

Another way to solve a least squares problem is by modified Gram-Schmidt orthogonalization (Algorithm ??). For $m \approx n$, this takes somewhat more operations than the Householder approach, but for $m \gg n$, the flop counts for both algorithms are asymptotic to $2mn^2$. We have the following code:

```
[Q, R] = mgs(A);
x = R \ (Q'*b);
x(15)
   ans = 1.02926594532672
```

This result is very poor. Rounding errors have been amplified by a factor on the order of $10^{14}$, far greater than the condition number of the problem. In fact, this algorithm is unstable, and the reason is easily identified. As mentioned at the end of Lecture 9, Gram-Schmidt orthogonalization produces matrices $\hat{Q}$, in general, whose columns are not accurately orthonormal. Since the algorithm above depends on that orthonormality, it suffers accordingly.

A better method of stabilizing the Gram-Schmidt method is to make use of an augmented system of equations, just as in the second of our two Householder experiments above:

```
[Q2,R2] = mgs([A b]);
R2 = R2(1:n,1:n);
Qb = R2(1:n,n+1);
x = R2\Qb;
x(15)
   ans = 1.00000005653399
```

Now the result looks as good as with Householder triangularization. It can be proved that this is always the case.

**Theorem 11.2** (Backstap of GS for LS).
The solution of the full-rank least squares problem (3.1) by Gram-Schmidt orthogonalization is also backward stable, satisfying (11.1), provided that $\hat{Q}^*b$ is formed implicitly as indicated in the code segment above.

## 11.4 Normal Equations

A fundamentally different approach to least squares problems is the solution of the normal equations (Algorithm 3.1), typically by Cholesky factorization (Chapter 23). For $m \gg n$, this method is twice as fast as methods depending on explicit orthogonalization, requiring asymptotically only $mn^2$ flops. In the following experiment, the problem is solved in a single line of MATLAB by the \ operator:

```
x = (A'*A)\(A'*b);                                    Form and solve normal equations.
x(15)
   ans = 0.39339069870283
```

This result is terrible! It is the worst we have obtained, with not even a single digit of accuracy. The use of the normal equations is clearly an unstable method for solving least squares problems.

Suppose we have a backward stable algorithm for the full-rank LS problem that delivers a solution $\tilde{x}$ satisfying $\|(A+\delta A)\tilde{x}-b\| = \min$ for some $\delta A$ with $\|\delta A\|/\|A\| = O(\epsilon_{\text{machine}})$. (Allowing perturbations in $b$ as well as $A$, or considering stability instead of backward stability, does not change our main points.) By Thm 7.2 of and Thm 10.1 we have

$$\frac{\|\tilde{x}-x\|}{\|x\|} = O\left(\left(\kappa + \frac{\kappa^2 \tan\theta}{\eta}\right)\epsilon_{\text{machine}}\right),$$

where $\kappa = \kappa(A)$. Now suppose $A$ is ill-conditioned, i.e., $\kappa \gg 1$, and $\theta$ is bounded away from $\pi/2$. Depending on the values of the various parameters, two very different situations may arise. If $\tan\theta$ is of order 1 (that is, the least squares fit is not especially close) and $\eta \ll \kappa$, the right-hand side is $O(\kappa^2\epsilon_{\text{machine}})$. On the other hand, if $\tan\theta$ is close to zero (a very close fit) or $\eta$ is close to $\kappa$, the bound is $O(\kappa\epsilon_{\text{machine}})$. The condition number of the least squares problem may lie anywhere in the range $\kappa$ to $\kappa^2$.

Now consider what happens when we solve LS by the normal equations, $(A^*A)\,x = A^*b$. Cholesky factorization is a stable algorithm for this system of equations in the sense that it produces a solution $\tilde{x}$ satisfying $(A^*A+\delta H)\,\tilde{x} = A^*b$ for some $\delta H$ with $\|\delta H\|/\|A^*A\| = O(\epsilon_{\text{machine}})$ (Theorem 23.3). However, the matrix $A^*A$ has condition number $\kappa^2$, not $\kappa$. Thus the best we can expect from the normal equations is

$$\frac{\|\tilde{x}-x\|}{\|x\|} = O\left(\kappa^2\epsilon_{\text{machine}}\right).$$

The behavior of the normal equations is governed by $\kappa^2$, not $\kappa$.

The conclusion is now clear. If $\tan\theta$ is of order 1 and $\eta \ll \kappa$, or if $\kappa$ is of order 1, then two bounds are of the same order and the normal equations are stable. If $\kappa$ is large and either $\tan\theta$ is close to zero or $\eta$ is close to $\kappa$, however, then the previous one is much bigger than the good bound and the normal equations are unstable. The normal equations are typically unstable for ill-conditioned problems involving close fits. In our example problem, with $\kappa^2 \approx 10^{20}$, it is hardly surprising that Cholesky factorization yielded no correct digits.

According to our definitions, an algorithm is stable only if it has satisfactory behavior uniformly across all the problems under consideration. The following result is thus a natural formalization of the observations just made.

**Theorem 11.3** (Stability of normal equations).
The solution of the full-rank least squares problem (3.1) via the normal equations (Algorithm 3.1) is unstable. Stability can be achieved, however, by restriction to a class of problems in which $\kappa(A)$ is uniformly bounded above or $(\tan\theta)/\eta$ is uniformly bounded below.

## 11.5 SVD

SVD is stable.

```
[U,S,V] = svd(A,0);
x = V*(S\(U'*b));
x(15)
   ans = 0.99999998230471
```

In fact, this is the most accurate of all the results obtained in our experiments, beating Householder triangularization with column pivoting (MATLAB's \ ) by a factor of about 3. A theorem in the usual form can be proved.

**Theorem 11.4** (Backstap of SVD).
The solution of the full-rank least squares problem (3.1) by the SVD (Algorithm 3.3) is backward stable, satisfying the estimate (11.1).

## 11.6 Rank-Deficient LS Problems

In this chapter we have identified four backward stable algorithms for linear least squares problems: Householder triangularization, Householder triangularization with column pivoting, modified Gram-Schmidt with implicit calculation of $\hat{Q}^*b$, and the SVD. From the point of view of classical normwise stability analysis of the full-rank problem 3.1 the differences among these algorithms are minor, so one might as well make use of the simplest and cheapest, Householder triangularization without pivoting.

However, there are other kinds of least squares problems where column pivoting and the SVD take on a special importance. These are problems where $A$ has rank $< n$, possibly with $m < n$, so that the system of equations is underdetermined. Such problems do not have a unique solution unless one adds an additional condition, typically that $x$ itself should have as small a norm as possible. A further complication is that the correct solution depends on the rank of $A$, and determining ranks numerically in the presence of rounding errors is never a trivial matter.

Thus rank-deficient least squares problems are not a challenging subclass of least squares problems, but fundamentally different. Since the definition of a solution is new, there is no reason that an algorithm that is stable for full-rank problems must be stable also in the rank-deficient case. In fact, the only fully stable algorithms for rank-deficient problems are those based on the SVD. An alternative is Householder triangularization with column pivoting, which is stable for almost all problems. We shall not give details.

# Part II

# Systems of Equations

# CHAPTER 12 GAUSSIAN ELIMINATION

Gaussian elimination is undoubtedly familiar to the reader. It's the simplest way to solve linear systems of equations by hand, and also the standard method for solving them on computers. We first describe Gaussian elimination in its pure form, and then, in the next lecture, add the feature of row pivoting that is essential to stability.

## 12.1 LU Factorization

Gaussian elimination transform a full linear system into an upper-triangular one by applying linear transformation on the left, which is quite close to householder triangularization. However, the difference is that the operations of Gaussian elimination are not unitary.

Let $A \in \mathbb{C}^{m\times m}$ be a square matrix. THe ideal is to transform $A$ into an $m \times m$ upper traingular matrix $U$ by introducing zeros below the diagonal. This is done by subtracting multiples of each row from subsequent rows, which is equivalent to left multiplying $A$ by a sequence of lower-triangular matrices $L_k$:

$$\underbrace{L_{m-1}\cdots L_2 L_1}_{L^{-1}} A = U. \tag{12.1}$$

Setting $L = L_1^{-1}L_2^{-1}\cdots L_{m-1}^{-1}$ gives $A = LU$. Then, we obtain an **LU factorization** of $A$,

$$A = LU, \tag{12.2}$$

where $U$ is upper-triangular and $L$ is lower-triangular. It turns out $L$ is **unit lower-triangular**, which means the diagonal entries are all 1.

**Note 12.1.**
Gaussian elimination augments our taxonomy of algorithms for factoring a matrix:

- GS: $A = QR$ by triangular orthogonalization,
- Householder: $A = QR$ by orthogonal triangularization,
- Gaussian elimination: $A = LU$ by triangular triangularization.

## 12.2 Example

Consider:

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

We have

$$L_1A = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}.$$

Then

$$L_2L_1A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}.$$

Finally,

$$L_3L_2L_1A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = U.$$

Note that we can compute inverses of $L_1, L_2, L_3$ easily.

$$L_1^{-1} = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix}.$$

Hence, we can get the LU factorization:

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix}.$$

## 12.3 General Formulas and Two Strokes of Luck

Assume $A \in \mathbb{C}^{m\times m}$ and $x_k$ is the $k$th column of $A$ at the beginning of step $k$. Then $L_k$ must be chosen so that:

$$x_k = \begin{bmatrix} x_{1k} \\ \vdots \\ x_{kk} \\ x_{k+1,k} \\ \vdots \\ x_{mk} \end{bmatrix} \xrightarrow{L_k} L_kx_k = \begin{bmatrix} x_{1k} \\ \vdots \\ x_{kk} \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

To do this we subtract $l_{jk}$ times row $k$ from row $j$, where $l_{jk}$ is the **multiplier**

$$l_{jk} = \frac{x_{jk}}{x_{kk}} \quad (k < j \leqslant m).$$

The matrix $L_k$ takes the form

$$L_k = \begin{bmatrix} 1 & & & & \\ & \vdots & & & \\ & & 1 & & \\ & & -l_{k+1,k} & 1 & \\ & & \vdots & & \ddots \\ & & -l_{m,k} & & & 1 \end{bmatrix},$$

with the nonzero subdiagonal entries situated in column $k$. In the numerical example above, w get two strokes of luck:

- $L_k$ can be inverted by negating its subdiagonal entries;
- $L$ can be formed be collecting the entries $l_{jk}$.

We can explain these bits of good fortune as follows. Let's define

$$l_k = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ l_{k+1,k} \\ \vdots \\ l_{m,k} \end{bmatrix}.$$

Then $L_k$ can be written $L_k = I - l_k e_k^*$. Note that $\langle e_k, l_k \rangle = e_k^* l_k = 0$. Hence,

$$(I - l_k e_k^*)(I + l_k e_k^*) = I - l_k e_k^* l_k e_k^* = I.$$

For the second stroke of luck, we just consider $L_k^{-1} L_{k+1}^{-1}$. Note that $e_k^* l_{k+1} = 0$, and therefore,

$$L_k^{-1} L_{k+1}^{-1} = (I + l_k e_k^*)(I + l_{k+1} e_{k+1}^*) = I + l_k e_k^* + l_{k+1} e_{k+1}^*.$$

From this, we can observe that:

$$L = L_1^{-1} L_2^{-1} \cdots L_{m-1}^{-1} = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ l_{m1} & l_{m2} & \cdots & l_{m,m-1} & 1 \end{bmatrix}.$$

Note that the modified GS process also have a similar result. In practice, we won't form $L_k$ but store $l_{jk}$ and form $L$.

**Algorithm 12.1:** Gaussian Elimination without Pivoting

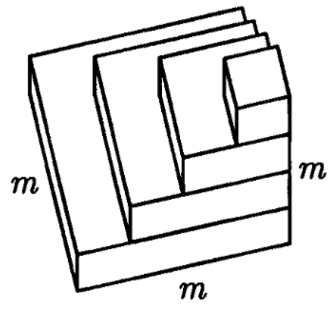
```
1 $U = A, L = I$;
2 for $k = 1$ to $m - 1$ do
3     for $j = k + 1$ to $m$ do
4         $l_{jk} = u_{jk}/u_{kk}$;
5         $u_{j,k:m} = u_{j,k:m} - l_{jk} u_{k,k,:m}$
```

**Note 12.2.**
Three matrices $A, L, U$ are not really needed; to minimize memory use on the computer, both $L$ and $U$ can be written into the same array as $A$.

## 12.4 Operation Count

As usual, the asymptotic operation count of this algorithm can be derived geometrically. The work is dominated by the vector operation in the inner loop, $u_{j,k:m} = u_{j,k:m} - \ell_{jk} u_{k,k:m}$, which executes one scalar-vector multiplication and one vector subtraction. If $l = m - k + 1$ denotes the length of the row vectors being manipulated, the number of flops is $2l$ : two flops per entry.

For each value of $k$, the inner loop is repeated for row $k + 1, \dots, m$. The work involved corresponds to one layer of the following solid:



Note that the solid converges as $m \to \infty$ to a pyramid, with volume $\frac{1}{3}m^3$. At two flops per unit of volume, we have

**Corollary 12.3.**
Work fo Gaussian elimination: $\sim \frac{2}{3}m^3$ flops.

## 12.5 Solution of $Ax = b$ by LU

We can solve $Ax = b$ with three steps:

- Decompose $A = LU$,
- Solve $Ly = b$,
- Solve $Ux = y$.

The total works is $\frac{2}{3}m^3 + 2m^2 \sim \frac{2}{3}m^3$. Which is half of the flops of Householder triangularization (??).

**Note 12.4.**
**Why is Gaussian elimination usually used rather than QR factorization to solve square systems of equations?**
The factor of 2 is certainly one reason. Also important, however, may be the historical fact that the elimination idea has been known for centuries, whereas QR factorization of matrices did not come along until after the invention of computers. To supplant Gaussian elimination as the method of choice, QR factorization would have to have had a compelling advantage.

## 12.6 Instability of Gaussian Elimination without Pivoting

Unfortunately, Gaussian elimination as presented so far is unusable for solving general linear systems, for it is not backward stable. The instability is related to another, more obvious difficulty. For certain matrices, Gaussian elimination fails entirely, because it attempts division by zero.

For example, consider

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

This matrix has full rank and is well-conditioned, with $\kappa(A) = (3 + \sqrt{5})/2 \approx 2.618$ in the 2-norm. Nevertheless, Gaussian elimination fails at the first step.

A slight perturbation of the same matrix reveals the more general problem. Suppose we apply Gaussian elimination to

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}. \tag{12.3}$$

If we apply the Gaussian elimination, we will get

$$L = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}.$$

Since $\epsilon_{\text{machine}} \approx 10^{-16}$, the number $1 - 10^{20}$ cannot be represented exactly. Assume it's rounded by $-10^{20}$. Then,

$$\tilde{L} = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix}, \quad \tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}.$$

Then, we have

$$\tilde{L}\tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix}.$$

If we solve $\tilde{L}\tilde{U}x = b$, the answer will be completely different.

A careful consideration of what has occurred in this example reveals the following. Gaussian elimination has computed the LU factorization stably: $\tilde{L}$ and $\tilde{U}$ are close to the exact factors for a matrix close to $A$ (in fact, $A$ itself). Yet it has not solved $Ax = b$ stably. The explanation is that the LU factorization, though stable, was **not backward stable**.

**Note 12.5.**
As a rule, if one step of an algorithm is a stable but not backward stable algorithm for solving a subproblem, the stability of the overall calculation may be in jeopardy.

In fact, for general $m \times m$ matrices $A$, the situation is worse than this. Gaussian elimination without pivoting is neither backward stable nor stable as a general algorithm for LU factorization. Additionally, the triangular matrices it generates have condition numbers that may be arbitrarily greater than those of $A$ itself, leading to additional sources of instability in the forward and back substitution phases of the solution of $Ax = b$.

# CHAPTER 13 PIVOTING

In the last lecture, we saw that Gaussian elimination in its pure form is unstable. The instability can be controlled by permuting the order of the rows of the matrix being operated on, an operation called pivoting. Pivoting has been a standard feature of Gaussian elimination computations since the 1950s.

## 13.1 Pivots

At step $k$ of Gaussian elimination, multiples of row $k$ are subtracted from rows $k+1, \ldots, m$ of the working matrix $X$ in order to introduce zeros in entry $k$ of these rows. In this operation row $k$, column $k$, and especially the entry $x_{kk}$ play special roles. We call $x_{kk}$ the **pivot**. From every entry in the submatrix $X_{k+1:m,k:m}$ is subtracted the product of a number in row $k$ and a number in column $k$, divided by $x_{kk}$:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \boldsymbol{x_{kk}} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ & x_{kk} & \times & \times & \times \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}.$$

However, we don't really have to choose $x_{kk}$ as the pivot. We can also choose $x_{ik}$:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \boldsymbol{x_{ik}} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \times & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \\ & x_{ik} & \times & \times & \times \\ & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}.$$

Similarly, we can also introduce zeros in column $j$ rather than column $k$:

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \boldsymbol{\times} & \boldsymbol{x_{ij}} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \times & \times & \times & \times \end{bmatrix} \longrightarrow \begin{bmatrix} \times & \times & \times & \times & \times \\ & \boldsymbol{\times} & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \boldsymbol{\times} & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \\ & \times & x_{ij} & \times & \times \\ & \boldsymbol{\times} & \mathbf{0} & \boldsymbol{\times} & \boldsymbol{\times} \end{bmatrix}.$$

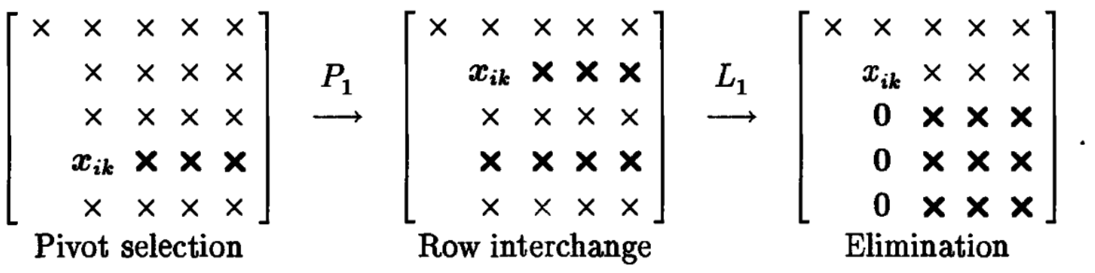
All in all, we are free to choose any entry of $X_{k:m,k:m}$ as the pivot, as long as it's nonzero. For numerical stability, it's desirable to pivot even when $x_{kk}$ is nonzero if there is a larger element available. In practice, we just pick as pivot the largest number among a set of entries. Now at step $k$, we will permute the rows and columns to make $x_{kk}$ is the largest number. This interchange of rows and perhaps columns is what is usually thought of as **pivoting**.

**Note 13.1.**
The ideal that rows and columns are interchanged is indispensable conceptually. Whether it's a good idea to interchange them physically on the computer is less clear. In some implementations, the data in computer memory are indeed swapped at each pivot step. In others, an equivalent effect is achieved by indirect addressing with permuted index vectors. Which is best varies from machine to machine and depends on many factors.

## 13.2 Partial Pivoting

If every entry of $X_{k:m,k:m}$ is considered as a possible pivot at step $k$, there are $O((m-k)^2)$ entries to be examined to determine the largest. Summing over $m$ steps, the total cost of selecting pivots is $O(m^3)$, adding significant to the cost of Gaussian elimination. This expensive strategy is called **complete pivoting**.

In practice, we only interchange rows and this standard method is called the **partial pivoting**. The pivot at each step is chosen as the largest of the $m-k+1$ subdiagonal entries in column $k$, incurring a total cost of only $O(m^2)$ operations.



This algorithm can be expressed as a matrix product. Now we need some permutation matrices $P_k$. After $m-1$ steps, $A$ becomes an upper-triangular matrix $U$:

$$L_{m-1}P_{m-1}\cdots L_2P_2L_1P_1A = U. \tag{13.1}$$

## 13.3 Example

Let's return to the example we already checked:

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

The first pivoting $P_1$:

$$\begin{bmatrix} & & 1 & \\ & 1 & & \\ 1 & & & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

The first elimination step now is $L_1$:

$$\begin{bmatrix} 1 & & & \\ -\frac{1}{2} & 1 & & \\ -\frac{1}{4} & & 1 & \\ -\frac{3}{4} & & & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & -\frac{7}{4} & -\frac{9}{4} & \frac{17}{4} \end{bmatrix}.$$

Now the second the forth rows are interchanged $P_2$:

$$\begin{bmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \end{bmatrix}.$$

Then we do the second elimination step $L_2$.

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{3}{7} & 1 & \\ & \frac{2}{7} & & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}.$$

Now we change the third and the fourth rows $P_3$:

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}. = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & -\frac{2}{7} & \frac{4}{7} \end{bmatrix}.$$

The final step ($L_3$) is:

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{1}{3} & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & -\frac{2}{7} & \frac{4}{7} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}.$$

## 13.4 $PA = LU$ Factorization and a Third Stroke of Luck

However, we are not doing LU but get an LU of $PA$, where $P$ is a permutation matrix. Combine all of them, it looks like:

$$\begin{bmatrix} & & 1 & \\ & & & 1 \\ & 1 & & \\ 1 & & & \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 & & & \\ -\frac{3}{4} & 1 & & \\ \frac{1}{2} & -\frac{2}{7} & 1 & \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix} \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}.$$

Note that, all the subdiagonal entries of $L$ are smaller than 1. Our elimination process took the form:

$$L_3P_3L_2P_2L_1P_1A = U,$$

which doesn't look lower-traingular at all. However, a third stroke of good fortune has come to our aid. The sec elementary operations can be reordered in the form

$$L_3P_3L_2P_2L_1P_1 = L_3'L_2'L_1'P_3P_2P_1,$$

where $L_k'$ is equal to $L_k$ but with the subdiagonal entries permuted. To be precise, define

$$L_3' = L_3, \quad L_2' = P_3L_2P_3^{-1}, \quad L_1' = P_3P_2L_1P_2^{-1}P_3^{-1}$$

Since of these definitions applies only permutations $P_j$ with $j > k$ to $L_k$, it's easily verified that $L_k'$ has the same structure as $L_k$.

In general, for an $m \times m$ matrix, the factorization provided by Gaussian elimination with partial pivoting can be written in the form

$$\left(L_{m-1}' \cdots L_2'L_1'\right)\left(P_{m-1} \cdots P_2P_1\right)A = U,$$

where $L_k'$ is defined by

$$L_k' = P_{m-1} \cdots P_{k+1}L_kP_{k+1}^{-1} \cdots P_{m-1}^{-1}$$

The product of the matrices $L_k'$ is unit lower-triangular and easily invertible by negating the subdiagonal entries, just as in Gaussian elimination without pivoting. Writing $L = (L_{m-1}' \cdots L_2'L_1')^{-1}$ and $P = P_{m-1} \cdots P_2P_1$, we have

$$PA = LU. \tag{13.2}$$

In general, any square matrix $A$, singular or nonsingular, has a factorization (13.2), where $P$ is a permutation matrix, $L$ is unit lower-triangular with lower-triangular entries $\leqslant 1$ in magnitude, and $U$ is upper-triangular. Partial pivoting is such a universal practice that this factorization is usually known simply as an **LU factorization** of $A$. This formula has a simple interpretation.

- Permute the rows of $A$ according to $P$,
- Apply Gaussian elimination without pivoting to $PA$.

The algorithm of LU with partial pivoting is:

**Algorithm 13.1:** Gaussian Elimination with Partial Pivoting

```
1 $U = A, L = I, P = I$;
2 for $k = 1$ to $m-1$ do
3     Select $i \geqslant k$ to maximize $|u_{ik}|$;
4     $u_{k,k:m} \leftrightarrow u_{i,k:m}$ (interchange two rows);
5     $l_{k,1:k-1} \leftrightarrow l_{i,1:k-1}$ ;
6     $p_{k,:} \leftrightarrow p_{i,:}$ ;
7     for $j = k+1$ to $m$ do
8         $l_{jk} = u_{jk}/u_{kk}$;
9         $u_{j,k:m} = u_{j,k:m} - l_{jk}u_{k,k:m}$
```

Note that the $L_k = I - v_ke_k^*$, hence

$$\begin{aligned} L_k' = I + P_{m-1} \cdots P_{k+1}(I + v_ke_k^*)P_{k+1}^{-1} \cdots P_{m-1}^{-1} &= I - P_{m-1} \cdots P_{k+1}v_ke_k^*P_{k+1}^{-1} \cdots P_{m-1}^{-1} \\ &= I - P_{m-1} \cdots P_{k+1}v_ke_k^*. \end{aligned}$$

Hence, we have

$$\left(L_k'\right)^{-1} = I - P_{m-1} \cdots P_{k+1}v_ke_k^*.$$

This explains why we can update $L$ like this in the algorithm.

To leading order, this algorithm requires the same number of floating point operations as Gaussian elimination without pivoting, namely, $\frac{2}{3}m^3$. As with algorithm 12.1, the use of computer memory can be minimized if desired by overwriting $U$ and $L$ into the same array used to store $A$.

In practice, of course, $P$ is not represented explicitly as a matrix. The rows are swapped at each step, or an equivalent effect is achieved via a permutation vector, as indicated earlier.

## 13.5 Complete Pivoting

In complete pivoting, the selection of pivots takes a significant amount of time. In practice this is rarely done, because the improvement in stability is marginal. However, we shall outline how the algebra changes in this case.

In matrix form, complete pivoting precedes each elimination step with a permutation $P_k$ of the rows applied on the left and also a permutation $Q_k$ of the columns applied on the right:

$$L_{m-1}P_{m-1}\cdots L_2P_2L_1P_1AQ_1Q_2\cdots Q_{m-1}=U.$$

Once again, this is not quite an LU factorization of $A$, but it is close. If the $L'_k$ are defined as the partial pivoting (the column permutations are not involved), then

$$\left(L'_{m-1}\cdots L'_2L'_1\right)\left(P_{m-1}\cdots P_2P_1\right)A\left(Q_1Q_2\cdots Q_{m-1}\right)=U.$$

Setting $L=\left(L'_{m-1}\cdots L'_2L'_1\right)^{-1}$, $P=P_{m-1}\cdots P_2P_1$, and $Q=Q_1Q_2\cdots Q_{m-1}$, we obtain

$$PAQ=LU.$$

# CHAPTER 14

# STABILITY OF GAUSSIAN ELIMINATION

Gaussian elimination with partial pivoting is explosively unstable for certain matrices, yet stable in practice. This apparent paradox has a statistical explanation.

## 14.1 Stability and the Size of $L$ and $U$

The stability analysis of most algorithms of numerical linear algebra, including virtually all of those based on unitary operations, is straightforward. The stability analysis of Gaussian elimination with partial pivoting, however, is complicated and has been a point of difficulty in numerical analysis since the 1950s.

In (12.3), we gave an example of a $2 \times 2$ matrix for which Gaussian elimination without pivoting was unstable. In that example, the factor $L$ had an entry of size $10^{20}$. An attempt to solve a system of equations based on $L$ introduced rounding errors of relative order $\epsilon_{\text{machine}}$, hence absolute order $\epsilon_{\text{machine}} \times 10^{20}$. Not surprisingly, this destroyed the accuracy of the result. Thus the purpose of pivoting, from the point of view of stability, is to ensure that $L$ and $U$ are not too large. As long as all the intermediate quantities that arise during the elimination are of manageable size, the rounding errors they emit are very small, and the algorithm is backward stable.

In fact, we have the following theorem for Gaussian elimination without pivoting.

**Theorem 14.1.**
Theorem 22.1. Let the factorization $A = LU$ of a nonsingular matrix $A \in \mathbb{C}^{m\times m}$ be computed by Gaussian elimination without pivoting (algorithm 12.1) on a computer satisfying the axioms (5.3) and (5.4). If $A$ has an LU factorization, then for all sufficiently small $\epsilon_{\text{machine}}$, the factorization completes successfully in floating point arithmetic (no zero pivots are encountered), and the computed matrices $\tilde{L}$ and $\tilde{U}$ satisfy

$$\tilde{L}\tilde{U} = A + \delta A, \qquad \frac{\|\delta A\|}{\|L\|\|U\|} = O\left(\epsilon_{\text{machine}}\right) \tag{14.1}$$

for some $\delta A \in \mathbb{C}^{m\times n}$.

Note that the difference is that the denominator is $\|L\|\|U\|$, not $\|A\|$. If $\|L\|\|U\| = O(\|A\|)$, then the Gaussian elimination is backward stable. The problem is that $\|L\|\|U\| \neq O(\|A\|)$.

For Gaussian elimination without pivoting, both $L$ and $U$ can be unboundedly large. That algorithm is unstable by any standard, and we shall not discuss it further. Instead, we should confine our attention to Gaussian elimination with partial pivoting.

## 14.2 Growth Factors

Consider Gaussian elimination with partial pivoting. Because each pivot selection involves maximization over a column, this algorithm produces a matrix $L$ with entries of absolute value $\leqslant 1$ everywhere below the diagonal. This implies $\|L\| = O(1)$ in any norm. Therefore, for Gaussian elimination with partial pivoting, (14.1) reduces to the condition $\|\delta A\|/\|U\| = O(\epsilon_{\text{machine}})$. We conclude that the algorithm is backward stable provided $\|U\| = O(\|A\|)$.

There is a standard reformulation of this that is perhaps more vivid. The key question for stability in terms of the two dimensional examples is whether amplification of the entries takes takes place during this reduction. In particular, we define the **growth factor** for $A$ to be defined as the ratio

$$\rho = \frac{\max_{i,j} |u_{ij}|}{\max_{i,j} |a_{ij}|}. \tag{14.2}$$

In fact, we should notice that $\|U\| = O(\rho\|A\|)$ due to the equivalence of norms.

**Corollary 14.2.**
Let the factorization $PA = LU$ of a matrix $A \in \mathbb{C}^{m\times m}$ be computed by Gaussian elimination with partial pivoting (algorithm 13.1) on a computer satisfying the axioms (5.3) and (5.4). Then the computed matrices $\tilde{P}$, $\tilde{L}$ and $\tilde{U}$ satisfy

$$\tilde{L}\tilde{U} = \tilde{P}A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = O(\rho\epsilon_{\text{machine}})$$

for some $\delta A \in \mathbb{C}^{m\times m}$, where $\rho$ is the growth factor for $A$. If $|l_{ij}| < 1$ for each $i > j$, implying that there are no ties in the selection of pivots in exact arithmetic, then $\tilde{P} = P$ for all sufficient small $\epsilon_{\text{machine}}$.

**Note 14.3.**
Is Gaussian elimination backward stable? According to Corollary 14.2 and our definition (6.5) of backward stability, the answer is yes if $\rho = O(1)$ uniformly for all matrices of a given dimension $m$, and otherwise no.

This makes things complicated.

## 14.3 Worst-Case Instability

For certain matrices $A$, despite the beneficial effects of pivoting, $\rho$ turns out to be huge. For example, suppose $A$ is the matrix

$$A = \begin{bmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & 1 & & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}. \tag{14.3}$$

If we do Gaussian elimination, we don't need pivoting at all. However, the entries $2, 3, \ldots, m$ in the final column are doubled. At the end we have

$$U = \begin{bmatrix} 1 & & & & 1 \\ & 1 & & & 2 \\ & & 1 & & 4 \\ & & & 1 & 8 \\ & & & & 16 \end{bmatrix}. \tag{14.4}$$

The final $PA = LU$ looks like:

$$\begin{bmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & 1 & & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ -1 & -1 & 1 & & \\ -1 & -1 & -1 & 1 & \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & 1 \\ & 1 & & & 2 \\ & & 1 & & 4 \\ & & & 1 & 8 \\ & & & & 16 \end{bmatrix}.$$

Hence the growth factor is $\rho = 16$. For an $m \times m$ matrix of the same form, it's $\rho = 2^{m-1}$. In fact, this is as large as $\rho$ can get.

A growth factor of order $2^m$ corresponds to a loss of on the order of $m$ bits of precision, which is catastrophic for a practical computation. Since a typical computer represents floating point numbers with just sixty-four bits, whereas matrix problems of dimensions in the hundreds or thousands are solved all the time, a loss of $m$ bits of precision is intolerable for real computations.

This brings us to an awkward point. Here, in the discussion of Gaussian elimination with pivoting, the definitions of stability presented in Chapter 14 fail us. According to the definitions, all that matters in determining stability or backward stability is the existence of a certain bound and applicable uniformly to all matrices for each fixed dimension $m$. Uniformity with respect to $m$ is not required. Here, for each $m$, we have a uniform bound involving the constant $2^{m-1}$. Thus, according to our definitions, Gaussian elimination is backward stable.

**Theorem 14.4.**
Gaussian elimination with partial pivoting is backward stable.

This conclusion is absurd, however, in view of the vastness of $2^{m-1}$ for practical values of $m$.

For the remainder of this lecture, we ask the reader to put aside our formal definitions of stability and accept a more informal (and more standard) use of words. Gaussian elimination for certain matrices is explosively unstable, as can be confirmed by numerical experiments with Matlab, LINPACK, LAPACK, or other software packages.

## 14.4 Stability in Practice

However, despite examples like (14.3), Gaussian elimination with partial pivoting is utterly stable in practice. Large factors $U$ like (14.4) never seem to appear in real applications. In fifty years of computing, no matrix problems that excite an explosive instability are known to have arisen under natural circumstances.

This is a curious situation indeed. How can an algorithm that fails for certain matrices be entirely trustworthy in practice? The answer seems to be that although some matrices cause instability, these represent such an extraordinarily small proportion of the set of all matrices that they "never" arise in practice simply for statistical reasons.

One can learn more about this phenomenon by considering random matrices. Of course, the matrices that arise in applications are not random in any ordinary sense. They have all kinds of special properties, and if one tried to describe them as random samples from some distribution, it would have to be a curious distribution indeed. It would certainly be unreasonable to expect that any particular distribution of random matrices should match the behavior of the matrices arising in practice in a close quantitative way.

However, the phenomenon to be explained is not a matter of precise quantities. Matrices with large growth factors are vanishingly rare in applications. If we can show that they are vanishingly rare among random matrices in some well-defined class, the mechanisms involved must surely be the same. The argument does not depend on one measure of "vanishingly" agreeing with the other to any particular factor such as 2 or 10 or 100.

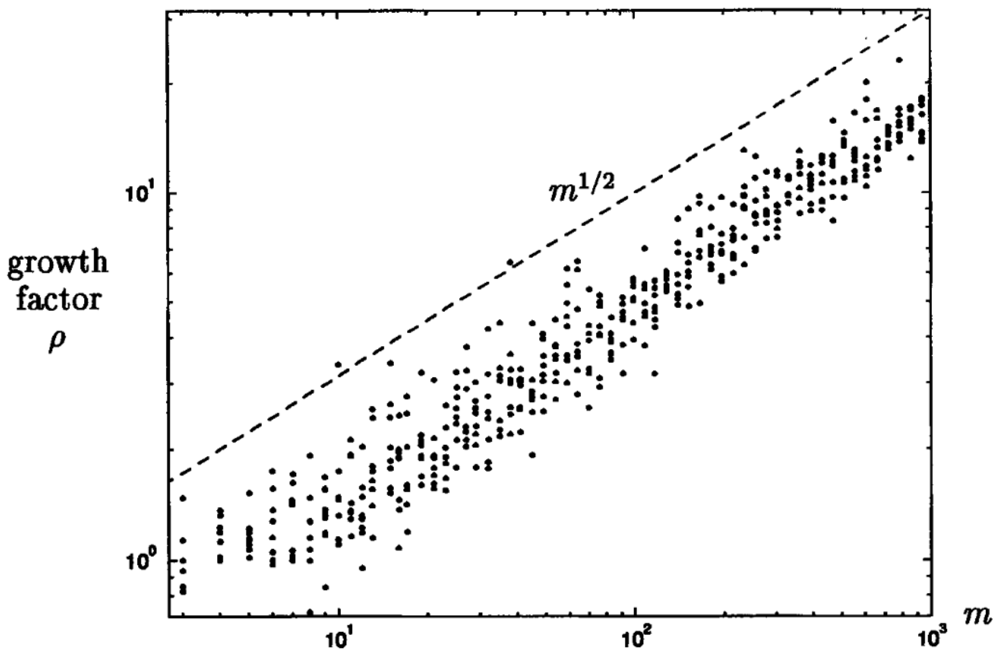


Figure 14.1: Growth factors for Gaussian elimination with partial pivoting applied to 496 random matrices of various dimensions. The typical size of $\rho$ is of order $m^{\frac{1}{2}}$, much less than the maximal possible value $2^{m-1}$.

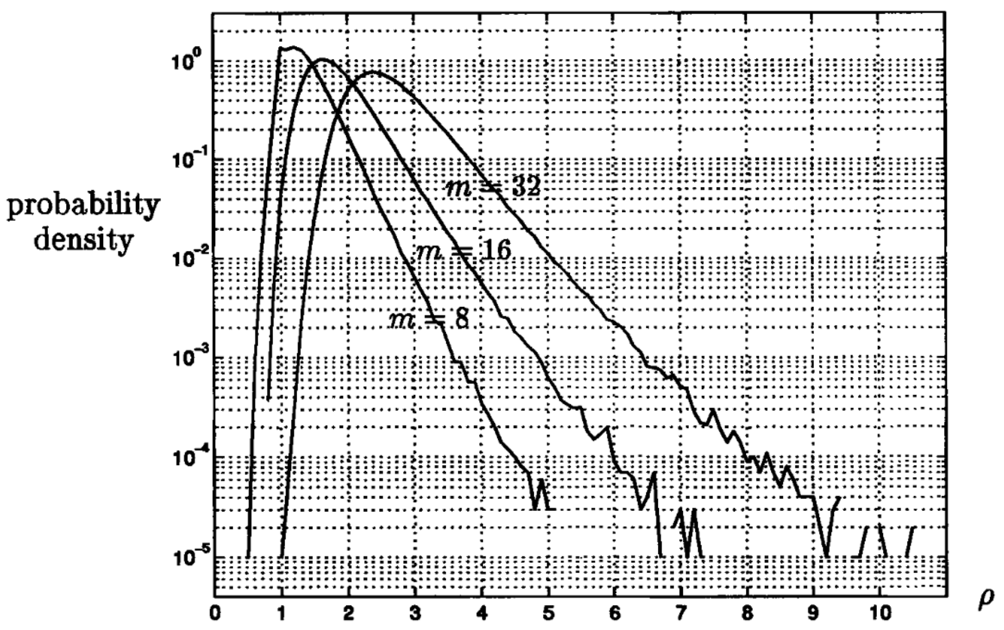


Figure 14.2: Probability distributions for growth factors of random matrices of dimensions $m = 8, 16, 32$, based on sample sizes of one million for each dimension. The density appears to decrease exponentially with $\rho$.

Figures 14.1 and 14.2 present experiments with random matrices where each entry is an independent sample from the real normal distribution of mean 0 and standard deviation $m^{\frac{1}{2}}$. In Figure 14.1, a collection of random matrices of various dimensions have been factored and the growth factor presented as a scatter plot. Only two of the matrices gave a growth factor as large as $m^{\frac{1}{2}}$. In Figure 14.2, the growth factors have been collected in bins of width 0.2 and the resulting data plotted as a probability density distribution. Among these three million matrices, though the maximum growth factor in principle might have been $2,147,483,648$, the maximum actually encountered was 11.99.

Similar results are obtained with random matrices defined by other probability distributions, such as uniformly distributed entries in $[-1, 1]$. If you pick a billion matrices at random, you will almost certainly not find one for which Gaussian elimination is unstable.

## 14.5 Explanation

We shall not attempt to give a full explanation.

If $PA = LU$, then $U = L^{-1}PA$. It follows that if Gaussian elimination is unstable when applied to the matrix $A$, implying that $\rho$ is large, then $L^{-1}$ must be large too. Now, as it happens, random triangular matrices tend to have huge inverses, exponentially large as a function of the dimension $m$. In particular, this is true for random triangular matrices of the form delivered by Gaussian elimination with partial pivoting, with 1 on the diagonal and entries $\leqslant 1$ in absolute value below.

When Gaussian elimination is applied to random matrices $A$, however, the resulting factors $L$ are anything but random. Correlations appear among the signs of the entries of $L$ that render these matrices extraordinarily wellconditioned. A typical entry of $L^{-1}$, far from being exponentially large, is usually less than 1 in absolute value.

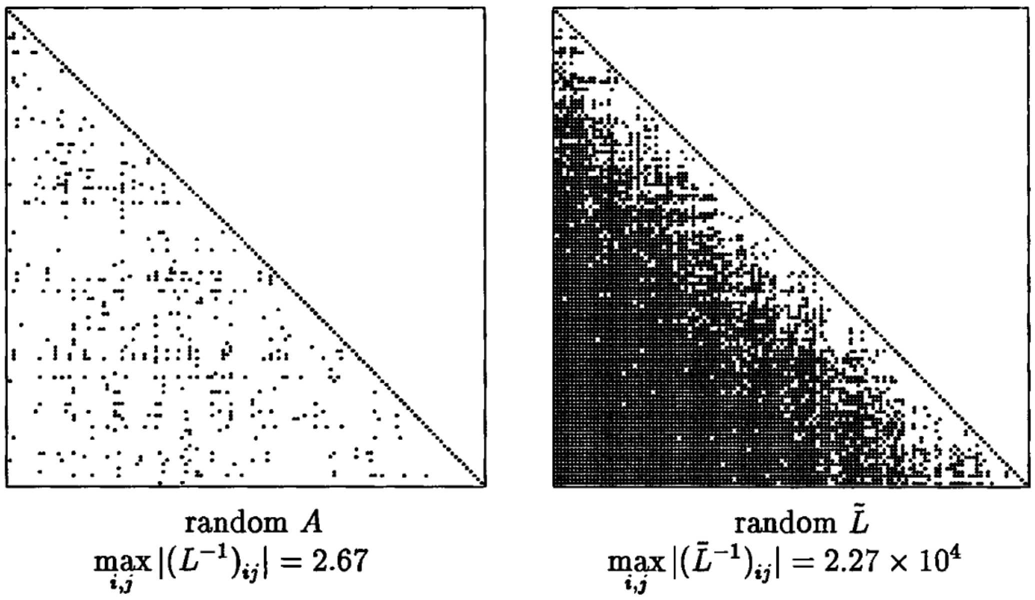


Figure 14.3: Let $A$ be a random $128 \times 128$ matrix with factorization $PA = LU$. On the left, $L^{-1}$ is shown: the dots represent entries with magnitude $\geqslant 1$. On the right, a similar picture for $\tilde{L}^{-1}$, where $\bar{L}$ is the same as $L$ except that the signs of its subdiagonal entries have been randomized. Gaussian elimination tends to produce matrices $L$ that are extraordinarily well-conditioned.

Then the question is: Why do the matrices $L$ delivered by Gaussian elimination almost never have large inverses?

The answer lies in the consideration of column spaces. Since $U$ is uppertriangular and $PA = LU$, the column spaces of $PA$ and $L$ are the same. By this we mean that the first column of $PA$ spans the same space as the first column of $L$, the first two columns of $PA$ span the same space as the first two columns of $L$, and so on. If $A$ is random, its column spaces are randomly oriented, and it follows that the same must be true of the column spaces of $P^{-1}L$. However, this condition is incompatible with $L^{-1}$ being large. It can be shown that **if $L^{-1}$ is large, then the column spaces of $L$, or of any permutation $P^{-1}L$, must be skewed in a fashion that is very far from random.**

Figure 14.4 gives evidence of this. We use the Q portrait, defined by the Matlab commands:

```
[Q, R] = qr(A);
spy(abs(Q) > 1/sqrt(m))
```

These commands first compute a QR factorization of the matrix $A$, then plot a dot at each position of $Q$ corresponding to an entry larger than the standard deviation, $m^{-1/2}$. The figure illustrates that for a random $A$, even after row interchanges to the form $PA$, the column spaces are oriented nearly randomly, whereas for a matrix $A$ that gives a large growth factor, the orientations are very far from random. It is likely that by quantifying this argument, it can be proved that growth factors larger than order $m^{1/2}$ are exponentially rare among random matrices in the sense that for any $\alpha > 1/2$ and $M > 0$, the probability of the event $\rho > m^{\alpha}$ is smaller than $m^{-M}$ for all sufficiently large $m$. As of this writing, however, such a theorem has not yet been proved.

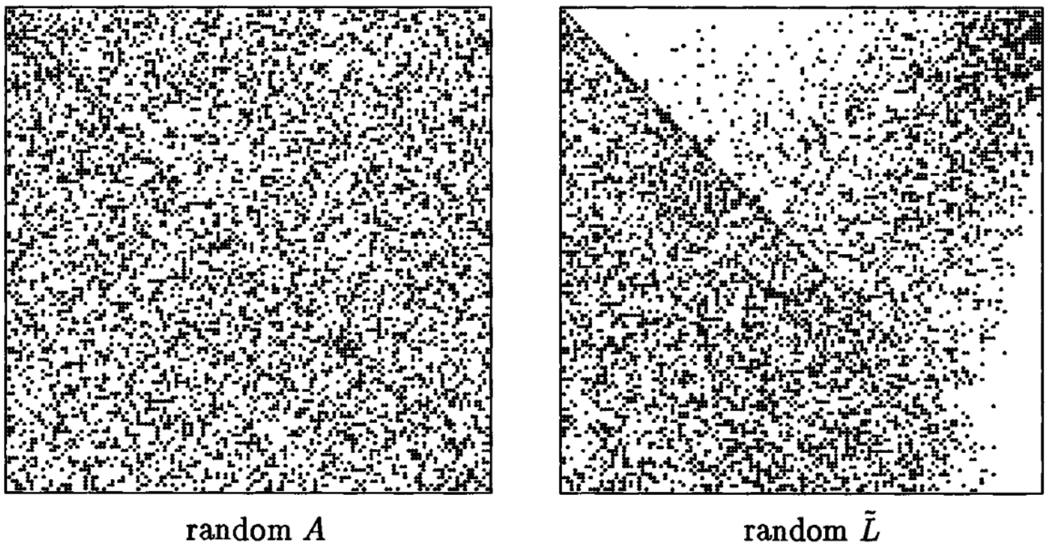


Figure 14.4: Q portraits of the same two matrices. On the left, the random matrix $A$ after permutation to the form $PA$, or equivalently, the factor $L$. On the right, the matrix $\tilde{L}$ with randomized signs. The column spaces of $\tilde{L}$ are skewed in a manner exponentially unlikely to arise in typical classes of random matrices.

**Note 14.5.**

Let us summarize the stability of Gaussian elimination with partial pivoting. This algorithm is highly unstable for certain matrices $A$. For instability to occur, however, the column spaces of $A$ must be skewed in a very special fashion, one that is exponentially rare in at least one class of random matrices. Decades of computational experience suggest that matrices whose column spaces are skewed in this fashion arise very rarely in applications.

# CHAPTER 15

# CHOLESKY FACTORIZATION

Hermitian positive definite matrices can be decomposed into triangular factors twice as quickly as general matrices. The standard algorithm for this, Cholesky factorization, is a variant of Gaussian elimination that operates on both the left and the right of the matrix at once, preserving and exploiting symmetry.

## 15.1 Symmetric Gaussian Elimination

First we recall the definition of Hermitian positive definite matrices.

**Definition 15.1 (Hermitian positive definite matrix).**
A matrix $A \in \mathbb{C}^{m\times m}$ is Hermitian positive definite if

- $A^* = A$,
- $\forall x \neq 0 \in \mathbb{C}^m, x^*Ax > 0$.

Note that all the eigenvalues of a Hermitian positive definite (PD) matrix are positive real numbers. Besides, eigenvectors corresponding to distinct eigenvalues are orthogonal.

We turn now to the problem of decomposing a PD matrix. To begin, we apply one step of GE to a PD matrix $A$ with a 1 in the upper-left position:

$$A = \begin{bmatrix} 1 & w^* \\ w & K \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ w & I \end{bmatrix} \begin{bmatrix} 1 & w^* \\ 0 & K - ww^* \end{bmatrix}.$$

However, to obtain symmetry, Cholesky factorization first introduces zeros in the first row to match the zeros just introduced in the first column. We can do this by a right upper-triangular operation:

$$\begin{bmatrix} 1 & w^* \\ 0 & K - ww^* \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & K - ww^* \end{bmatrix} \begin{bmatrix} 1 & w^* \\ 0 & I \end{bmatrix}.$$

Combine them, we get

$$A = \begin{bmatrix} 1 & w^* \\ w & K \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ w & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & K - ww^* \end{bmatrix} \begin{bmatrix} 1 & w^* \\ 0 & I \end{bmatrix}. \tag{15.1}$$

The idea of Cholesky factorization is to continue this process, zeroing one column and one row of $A$ symmetrically until it is reduced to the identity.

## 15.2 Cholesky Factorization

In order for the symmetric triangular reduction to work in general, we need a factorization that works for any $a_{11} > 0$, not just $a_{11} = 1$. The generalization of (15.1) is accomplished by adjusting some of the elements of $R_1$ by a factor of $\sqrt{a_{11}}$. Let $\alpha = \sqrt{a_{11}}$ and observe:

$$\begin{aligned} A &= \begin{bmatrix} a_{11} & w^* \\ w & K \end{bmatrix} \\ &= \begin{bmatrix} \alpha & 0 \\ w/\alpha & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & K - ww^*/a_{11} \end{bmatrix} \begin{bmatrix} \alpha & w^*/\alpha \\ 0 & I \end{bmatrix} = R_1^* A_1 R_1 \end{aligned}$$

This is the basic step that is applied repeatedly in Cholesky factorization. If the upper-left entry of the submatrix $K - ww^*/a_{11}$ is positive, the same formula can be used to factor it; we then have $A_1 = R_2^* A_2 R_2$ and thus $A = R_1^* R_2^* A_2 R_2 R_1$. The process is continued down to the bottom-right corner, giving us eventually a factorization

$$A = \underbrace{R_1^* R_2^* \cdots R_m^*}_{R^*} \underbrace{R_m \cdots R_2 R_1}_{R}. \tag{15.2}$$

This equation has the form

$$A = R^* R, \quad r_{jj} > 0, \tag{15.3}$$

where $R$ is upper-triangular. A reduction of this kind of a hermitian positive definite matrix is known as a **Cholesky factorization**.

The description above left one item dangling. How do we know that the upper-left entry of the submatrix $K - ww^*/a_{11}$ is positive? The answer is that it must be positive because $K - ww^*/a_{11}$ is positive definite, since it is the $(m-1) \times (m-1)$ lower-right principal submatrix of the positive definite matrix $R_1^{-*} A R_1^{-1}$.

**Theorem 15.2** (Cholesky factorization).
Every Hermitian positive definite matrix $A \in \mathbb{C}^{m \times m}$ has a unique Cholesky factorization (15.3).

Note that the uniquenessss comes from the form $RR^*$. Check the first row of $A$ will lead to uniqueness of first row of $R$. Then, we can use induction.

## 15.3 The Algorithm

When Cholesky factorization is implemented, only half of the matrix being operated on needs to be represented explicitly. This simplification allows half of the arithmetic to be avoided. A formal statement of the algorithm (only one of many possibilities) is given below. The input matrix $A$ represents the superdiagonal half of the $m \times m$ hermitian positive definite matrix to be factored. (In practical software, a compressed storage scheme may be used to avoid wasting half the entries of a square array.) The output matrix $R$ represents the upper-triangular factor for which $A = R^* R$. Each outer iteration corresponds to a single elementary factorization: the upper-triangular part of the submatrix $R^*_{k:m,k:m}$ represents the superdiagonal part of the hermitian matrix being factored at step $k$.

**Algorithm 15.1:** Cholesky Factorization

```
R = A;
for $k = 1$ to $m$ do
    for $j = k+1$ to $m$ do
        $R_{j,j:m} = R_{j,j:m} - R_{k,j:m} R_{kj} / R_{kk}$
    $R_{k,k:m} = R_{k,k:m} / \sqrt{R_{kk}}$
```

## 15.4 Operation Count

The arithmetic done in Cholesky factorization is dominated by the inner loop. A single execution of the line

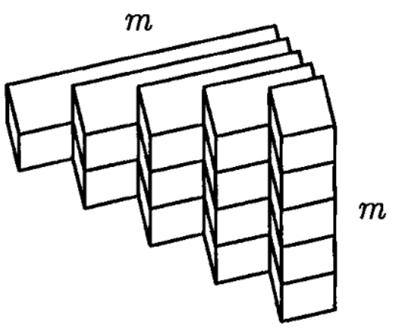
$$R_{j,j:m} = R_{j,j:m} - R_{k,j:m} R_{kj}/R_{kk}$$

requires one division, $m - j + 1$ multiplications, and $m - j + 1$ subtractions, for a total of $\sim 2(m - j)$ flops. This calculation is repeated once for each $j$ from $k + 1$ to $m$, and that loop is repeated for each $k$ from 1 to $m$. The sum is straightforward to evaluate:

$$\sum_{k=1}^{m} \sum_{j=k+1}^{m} 2(m-j) \sim 2 \sum_{k=1}^{m} \sum_{j=1}^{k} j \sim \sum_{k=1}^{m} k^2 \sim \frac{1}{3} m^3 \text{ flops.}$$

Thus, Cholesky factorization involves only half as many operations as Gaussian elimination, which would require $\sim \frac{2}{3}m^3$ flops to factor the same matrix.

As usual, the operation count can also be determined graphically. For each $k$, two floating point operations are carried out (one multiplication and one subtraction) at each position of a triangular layer. The entire algorithm corresponds to stacking $m$ layers:



It's obvious that the volume is $\frac{1}{6}m^3$.

**Corollary 15.3.**
Work for Cholesky factorization: $\sim \frac{1}{3}m^3$ flops.

## 15.5 Stability

All of the subtleties of the stability analysis of Gaussian elimination vanish for Cholesky factorization. This algorithm is always stable. Intuitively, the reason is that the factors $R$ can never grow large. In the 2-norm, for example, we have $\|R\| = \|R^*\| = \|A\|^{1/2}$ (proof: SVD), and in other $p$-norms with $1 \leqslant p \leqslant \infty$, $\|R\|$ cannot differ from $\|A\|^{1/2}$ by more than a factor of $\sqrt{m}$. Thus, numbers much larger than the entries of $A$ can never arise.

Note that the stability of Cholesky factorization is achieved without the need for any pivoting. Intuitively, one may observe that this is related to the fact that most of the weight of a hermitian positive definite matrix is on the diagonal.

**Theorem 15.4 (Backstap of Cholesky).**
Let $A \in \mathbb{C}^{m\times m}$ be hermitian positive definite, and let a Cholesky factorization of $A$ be computed by algorithm 15.1 on a computer satisfying (5.3) and (5.4). For all sufficiently small $\epsilon_{\text{machine}}$, this process is guaranteed to run to completion (i.e., no zero or negative corner entries $r_{kk}$ will arise), generating a computed factor $\tilde{R}$ that satisfies

$$\tilde{R}^* \tilde{R} = A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = O\left(\epsilon_{\text{machine}}\right) \tag{15.4}$$

for some $\delta A \in \mathbb{C}^{m\times m}$.

**Note 15.5.**
Like so many algorithms of numerical linear algebra, this one would look much worse if we tried to carry out a forward error analysis rather than a backward one. If $A$ is ill-conditioned, $\tilde{R}$ will not generally be close to $R$; the best we can say is $\|\tilde{R} - R\|/\|R\| = O(\kappa(A)\epsilon_{\text{machine}})$. (In other words, Cholesky factorization is in general an ill-conditioned problem.) It is only the product $\tilde{R}^*R$ that satisfies the much better error bound (15.4). Thus the errors introduced in $\tilde{R}$ by rounding are large but "diabolically correlated," just as we saw in Chapter 16 for QR factorization.

## 15.6 Solution of $Ax = b$

If $A$ is hermitian positive definite, the standard way to solve a system of equations $Ax = b$ is by Cholesky factorization. algorithm 15.1 reduces the system to $R^*Rx = b$, and we then solve two triangular systems in succession: first $R^*y = b$ for the unknown $y$, then $Rx = y$ for the unknown $x$. Each triangular solution requires just $\sim m^2$ flops, so the total work is again $\sim \frac{1}{3}m^3$ flops.

By reasoning analogous to that of Chapter 16, it can be shown that this process is backward stable.

**Theorem 15.6.**
The solution of hermitian positive definite systems $Ax = b$ via Cholesky factorization (algorithm 15.1) is backward stable, generating a computed solution $\tilde{x}$ that satisfies

$$(A + \Delta A)\tilde{x} = b, \qquad \frac{\|\Delta A\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

for some $\Delta A \in \mathbb{C}^{m\times m}$.