Berkeley
UNIVERSITY OF CALIFORNIA

# Numerical Linear Algebra

Notes By: Yuhang Cai

2023 Spring

# CONTENTS

# Part I

# Fundamentals

# CHAPTER 1

# MATRIX-VECTOR MULTIPLICATION

Given a matrix $A \in \mathbb{C}^{m \times n}$, The range of $A$ is:

$$\text{range}(A) = \{Ax | x \in \mathbb{C}^n\}.$$

Hence, range($A$) is the space spanned by the columns of $A$. The column rank of a matrix is the dimension of its column space and the row rank of a matrix is the dimension of the space spanned by its rows. Row rank always equals column rank.

# CHAPTER 2

# NORMS

The essential notions of size and distance in a vector space are captured by norms. These are the yardsticks with which we measure approximations and convergence throughout numerical linear algebra.

## 2.1 Vector Norms

**Definition 2.1** (Norm).
A norm is a function $\|\cdot\| : \mathbb{C}^m \to \mathbb{R}$ that assigns a real-valued length to each vector. In order to conform to a reasonable notion of length, a norm must satisfy the following three conditions. For all vectors $x$ and $y$ and for all scalars $\alpha \in \mathbb{C}$,

- $\|x\| \geqslant 0$, and $\|x\| = 0$ only if $x = 0$,
- $\|x + y\| \leqslant \|x\| + \|y\|$,
- $\|\alpha x\| = |\alpha|\|x\|$.

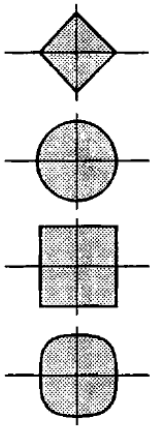
The most important class of vector norms, the $p$-norms, are defined below. The closed unit ball $\{x \in \mathbb{C}^m : \|x\| \leqslant 1\}$ corresponding to each norm is illustrated to the right for the case $m = 2$.

$$\|x\|_1 = \sum_{i=1}^{m} |x_i|,$$

$$\|x\|_2 = \left( \sum_{i=1}^{m} |x_i|^2 \right)^{1/2} = \sqrt{x^* x},$$

$$\|x\|_\infty = \max_{1 \leq i \leq m} |x_i|,$$

$$\|x\|_p = \left( \sum_{i=1}^{m} |x_i|^p \right)^{1/p} \quad (1 \leq p < \infty).$$



## 2.2 Induced Matrix Norms

An $m \times n$ matrix can be viewed as a vector in an $mn$-dimensional space: each of the $mn$ entries of the matrix is an independent coordinate. Any $mn$ dimensional norm can therefore be used for measuring the "size" of such a matrix. However, in dealing with a space of matrices, certain special norms are more useful than the vector norms already discussed. These are the induced matrix norms, defined in terms of the behavior of a matrix as an operator between its normed domain and range spaces.

Given vector norms $\|\cdot\|_{(n)}$ and $\|\cdot\|_{(m)}$ on the domain and the range of $A \in \mathbb{C}^{m\times n}$, respectively, the induced matrix norm $\|A\|_{(m,n)}$ is

$$\|A\|_{(m,n)} = \sup_{\substack{x\in\mathbb{C}^n \\ x\neq 0}} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}} = \sup_{\substack{x\in\mathbb{C}^n \\ \|x\|_{(n)}=1}} \|Ax\|_{(m)}.$$

## 2.3 Cauchy-Schwarz and Hölder Inequalities

Computing matrix $p$-norms with $p \neq 1, \infty$ is more difficult, and to approach this problem, we note that inner products can be bounded using $p$-norms. Let $p$ and $q$ satisfy $1/p + 1/q = 1$, with $1 \leqslant p, q \leqslant \infty$. Then the Hölder inequality states that, for any vectors $x$ and $y$,

$$|x^*y| \leqslant \|x\|_p \|y\|_q \tag{2.1}$$

The Cauchy-Schwarz inequality is the special case $p = q = 2$ :

$$|x^*y| \leqslant \|x\|_2 \|y\|_2 \tag{2.2}$$

Derivations of these results can be found in linear algebra texts. Both bounds are tight in the sense that for certain choices of $x$ and $y$, the inequalities become equalities.

## 2.4 Bounding $\|AB\|$ in an Induced Matrix Norm

Computing matrix $p$-norms with $p \neq 1, \infty$ is more difficult, and to approach this problem, we note that inner products can be bounded using $p$-norms. Let $p$ and $q$ satisfy $1/p + 1/q = 1$, with $1 \leqslant p, q \leqslant \infty$. Then the Hölder inequality states that, for any vectors $x$ and $y$,

$$|x^*y| \leqslant \|x\|_p \|y\|_q$$

The Cauchy-Schwarz inequality is the special case $p = q = 2$ :

$$|x^*y| \leqslant \|x\|_2 \|y\|_2$$

Derivations of these results can be found in linear algebra texts. Both bounds are tight in the sense that for certain choices of $x$ and $y$, the inequalities become equalities.

## 2.5 General Matrix Norms

As noted above, matrix norms do not have to be induced by vector norms. In general, a matrix norm must merely satisfy the three vector norm conditions applied in the $mn$-dimensional vector space of matrices:

- $\|A\| \geqslant 0$, and $\|A\| = 0$ only if $A = 0$,
- $\|A + B\| \leqslant \|A\| + \|B\|$,
- $\|\alpha A\| = |\alpha| \|A\|$.

The most important matrix norm which is not induced by a vector norm is the **Hilbert-Schmidt** or **Frobenius norm**, defined by

$$\|A\|_F = \left( \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}.$$

Observe that this is the same as the 2-norm of the matrix when viewed as an $mn$-dimensional vector. The formula for the Frobenius norm can also be written in terms of individual rows or columns. For example, if $a_j$ is the $j$ th column of $A$, we have

$$\|A\|_F = \left( \sum_{j=1}^{n} \|a_j\|_2^2 \right)^{1/2}$$

This identity, as well as the analogous result based on rows instead of columns, can be expressed compactly by the equation

$$\|A\|_F = \sqrt{\operatorname{tr}(A^*A)} = \sqrt{\operatorname{tr}(AA^*)}.$$

Note that

$$\|AB\|_F^2 \leqslant \|A\|_F^2 \|B\|_F^2.$$

## 2.6 Invariance under Unitary Multiplication

**Theorem 2.2.**
For any $A \in \mathbb{C}^{m\times n}$ and unitary $Q \in \mathbb{C}^{m\times m}$, we have

$$\|QA\|_2 = \|A\|_2, \quad \|QA\|_F = \|A\|_F.$$

# CHAPTER 3

# THE SINGULAR VALUE DECOMPOSITION

The singular value decomposition (SVD) is a matrix factorization whose computation is a step in many algorithms. Equally important is the use of the SVD for conceptual purposes. Many problems of linear algebra can be better understood if we first ask the question: what if we take the SVD?

## 3.1 A Geometric Observation

The SVD is motivated by the following geometric fact:

The image of the unit sphere under any $m \times n$ matrix is a hyperellipse.

The SVD is applicable to both real and complex matrices. However, in describing the geometric interpretation, we assume as usual that the matrix is real.

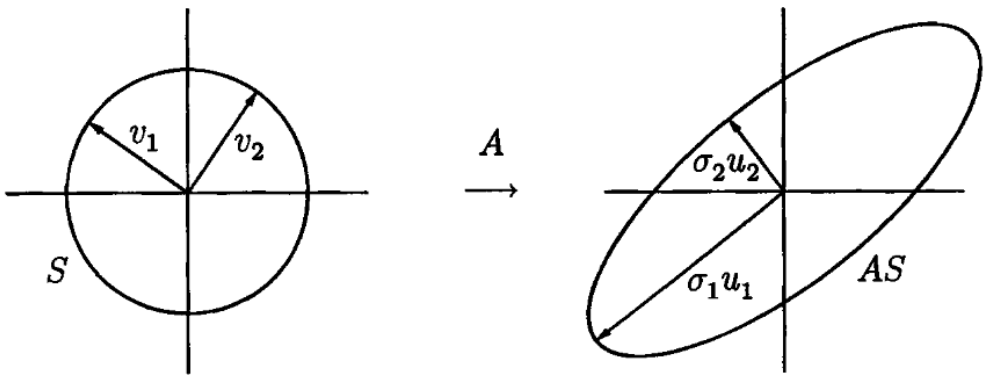



Figure 3.1: SVD of a $2 \times 2$ matrix

Let $S$ be the unit sphere in $\mathbb{R}^n$, and take any $A \in \mathbb{R}^{m \times n}$ with $m \geqslant n$. For simplicity, suppose for the moment that $A$ has full rank $n$. The image $AS$ is a hyperellipse in $\mathbb{R}^m$. We now define some properties of $A$ in terms of the shape of $AS$. The key ideas are indicated in Figure 3.1.

First, we define the $n$ singular values of $A$. These are the lengths of the $n$ principal semiaxes of $AS$, written $\sigma_1, \sigma_2, \ldots, \sigma_n$. It is conventional to assume that the singular values are numbered in descending order, $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_n > 0$

Next, we define the $n$ left singular vectors of $A$. These are the unit vectors $\{u_1, u_2, \ldots, u_n\}$ oriented in the directions of the principal semiaxes of $AS$, numbered to correspond with the singular values. Thus the vector $\sigma_i u_i$ is the $i$ th largest principal semiaxis of $AS$.

Finally, we define the $n$ right singular vectors of $A$. These are the unit vectors $\{v_1, v_2, \ldots, v_n\} \in S$ that are the preimages of the principal semiaxes of $AS$, numbered so that $Av_j = \sigma_j u_j$.

## 3.2 Reduced SVD

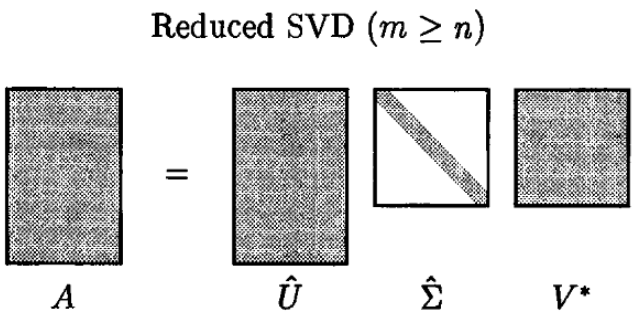
We have just mentioned that the equations relating right singular vectors $\{v_j\}$ and left singular vectors $\{u_j\}$ can be written

$$Av_j = \sigma_j u_j, \quad 1 \leqslant j \leqslant n.$$

More compactly, $AV = \hat{U}\hat{\Sigma}$. In this matrix equation, $\hat{\Sigma}$ is an $n \times n$ diagonal matrix with positive real entries (since $A$ was assumed to have full rank $n$ ), $\hat{U}$ is an $m \times n$ matrix with orthonormal columns, and $V$ is an $n \times n$ matrix with orthonormal columns. Thus $V$ is unitary, and we can multiply on the right by its inverse $V^*$ to obtain

$$A = \hat{U}\hat{\Sigma}V^*$$

This factorization of $A$ is called a reduced singular value decomposition, or reduced $SVD$, of $A$. Schematically, it looks like this:

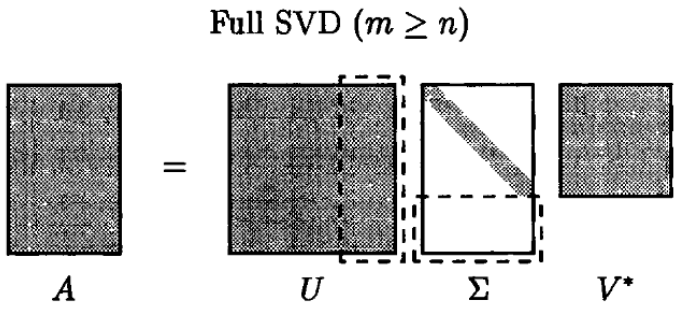



## 3.3 Full SVD

If we extend $U$ to a unitary matrix with extra orthonormal column vectors, we will get the full SVD.

$$A = U\Sigma V^*$$

Here $U$ is $m \times m$ and unitary, $V$ is $n \times n$ and unitary, and $\Sigma$ is $m \times n$ and diagonal with positive real entries. Schematically:




## 3.4 Formal Definition

Let $m$ and $n$ be arbitrary; we do not require $m \geqslant n$. Given $A \in \mathbb{C}^{m\times n}$, not necessarily of full rank, a singular value decomposition (SVD) of $A$ is a factorization

$$A = U\Sigma V^* \tag{3.1}$$

where $U \in \mathbb{C}^{m\times m}$ is unitary, $V \in \mathbb{C}^{n\times n}$ is unitary, $\Sigma \in \mathbb{R}^{m\times n}$ is diagonal. In addition, it is assumed that the diagonal entries $\sigma_j$ of $\Sigma$ are nonnegative and in nonincreasing order; that is, $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_p \geqslant 0$, where $p = \min(m, n)$.

## 3.5 Existence and Uniqueness

**Theorem 3.1** (Existence and uniqueness of SVD).
Every matrix $A \in \mathbb{C}^{m \times n}$ has a singular value decomposition (**??**). Furthermore, the singular values $\{\sigma_j\}$ are uniquely determined, and, if $A$ is square and the $\sigma_j$ are distinct, the left and right singular vectors $\{u_j\}$ and $\{v_j\}$ are uniquely determined up to complex signs (i.e., complex scalar factors of absolute value 1).

*Proof.*

- Existence: Consider $\|A\|_2$ and the corresponding vector.
- Uniqueness: The same.

□

# CHAPTER 4

# MORE ON THE SVD

We continue our discussion of the singular value decomposition, emphasizing its connection with low-rank approximation of matrices in the 2-norm and the Frobenius norm.

## 4.1 SVD vs Eigenvalue Decomposition

There are fundamental differences between the SVD and the eigenvalue decomposition.

- One is that the SVD uses two different bases (the sets of left and right singular vectors), whereas the eigenvalue decomposition uses just one (the eigenvectors).
- Another is that the SVD uses orthonormal bases, whereas the eigenvalue decomposition uses a basis that generally is not orthogonal.
- A third is that not all matrices (even square ones) have an eigenvalue decomposition, but all matrices (even rectangular ones) have a singular value decomposition, as we established in Theorem 3.1.
- In applications, eigenvalues tend to be relevant to problems involving the behavior of iterated forms of $A$, such as matrix powers $A^k$ or exponentials $e^{tA}$, whereas singular vectors tend to be relevant to problems involving the behavior of $A$ itself, or its inverse.

## 4.2 Matrix Properties via the SVD

The power of the SVD becomes apparent as we begin to catalogue its connections with other fundamental topics of linear algebra. For the following theorems, assume that $A$ has dimensions $m \times n$. Let $p$ be the minimum of $m$ and $n$, let $r \leqslant p$ denote the number of nonzero singular values of $A$, and let $\langle x, y, \ldots, z \rangle$ denote the space spanned by the vectors $x, y, \ldots, z$.

**Theorem 4.1.**
The rank of $A$ is $r$, the number of nonzero singular values.

**Theorem 4.2.**
$\text{range}(A) = \langle u_1, \ldots, u_r \rangle$ and $\text{null}(A) = \langle v_{r+1}, \ldots, v_n \rangle$.

**Theorem 4.3.**
$\|A\|_2 = \sigma_1$ and $\|A\|_F = \sqrt{\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2}$.

**Theorem 4.4.**
The nonzero singular values of $A$ are the square roots of the nonzero eigenvalues of $A^*A$ or $AA^*$. (These matrices have the same nonzero eigenvalues.)

**Theorem 4.5.**
If $A = A^*$, then the singular values of $A$ are the absolute values of the eigenvalues of $A$.

## 4.3 Low-Rank Approximations

But what is the SVD? Another approach to an explanation is to consider how a matrix $A$ might be represented as a sum of rank-one matrices.

**Theorem 4.6.**
$A$ is the sum of $r$ rank-one matrices:

$$A = \sum_{j=1}^{r} \sigma_j u_j v_j^* \tag{4.1}$$

Formula (4.1), however, represents a decomposition into rank-one matrices with a deeper property: the $\nu$ th partial sum captures as much of the energy of $A$ as possible. This statement holds with "energy" defined by either the 2-norm or the Frobenius norm. We can make it precise by formulating a problem of best approximation of a matrix $A$ by matrices of lower rank.

**Theorem 4.7.**
For any $\nu$ with $0 \leqslant \nu \leqslant r$, define

$$A_\nu = \sum_{j=1}^{\nu} \sigma_j u_j v_j^* \tag{4.2}$$

if $\nu = p = \min\{m, n\}$, define $\sigma_{\nu+1} = 0$. Then

$$\|A - A_\nu\|_2 = \inf_{\substack{B \in \mathbf{C}^{m\times n} \\ \mathbf{rank}(B) \leqslant \nu}} \|A - B\|_2 = \sigma_{\nu+1}.$$

Similarly,

**Theorem 4.8.**
For any $\nu$ with $0 \leqslant \nu \leqslant r$, the matrix $A_\nu$ of (4.2) also satisfies

$$\|A - A_\nu\|_F = \inf_{\substack{B \in \mathbb{C}^{m\times n} \\ \mathrm{rank}(B) \leqslant \nu}} \|A - B\|_F = \sqrt{\sigma_{\nu+1}^2 + \cdots + \sigma_r^2}.$$

*Proof.*
Note that for $A = A' + A''$, we have

$$\sigma_i(A') + \sigma_j(A'') \geqslant \sigma_{i+j-1}(A).$$

□

# Part II

# QR factorization and Least Squares

# CHAPTER 5

# PROJECTORS

We now enter the second part of the book, whose theme is orthogonality. We begin with the fundamental tool of projection matrices, or projectors, both orthogonal and nonorthogonal.

## 5.1 Projectors

**Definition 5.1** (Projectors).
A projector is a square matrix $P$ that satisfies

$$P^2 = P.$$

This definition includes both orthogonal projectors, to be discussed in a moment, and nonorthogonal ones. To avoid confusion one may use the term oblique projector in the nonorthogonal case.

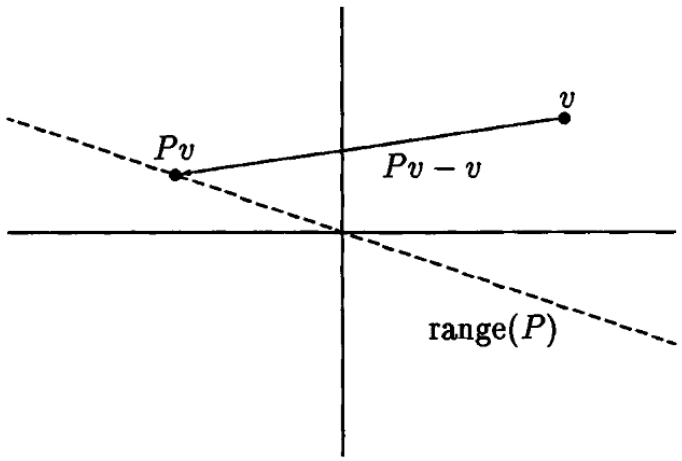



**Figure 6.1.** *An oblique projection.*

## 5.2 Complementary Projectors

**Definition 5.2** (Complementary Projectors).
If $P$ is a projector, $I - P$ is also a projector, for it is also idempotent:

$$(I - P)^2 = I - 2P + P^2 = I - P.$$

We have the following corollary:

**Corollary 5.3.**
We have the following results:

- $\text{range}(I - P) = \text{null}(P)$,
- $\text{null}(I - P) = \text{range}(P)$,
- $\text{range}(P) \cap \text{null}(P) = \{0\}$.

These computations show that a projector separates $\mathbb{C}^m$ into two spaces. Conversely, let $S_1$ and $S_2$ be two subspaces of $\mathbb{C}^m$ such that $S_1 \cap S_2 = \{0\}$ and $S_1 + S_2 = \mathbb{C}^m$, where $S_1 + S_2$ denotes the span of $S_1$ and $S_2$, that is, the set of vectors $s_1 + s_2$ with $s_1 \in S_1$ and $s_2 \in S_2$. (Such a pair are said to be complementary subspaces.) Then there is a projector $P$ such that $\text{range}(P) = S_1$ and $\text{null}(P) = S_2$. We say that $P$ is the projector onto $S_1$ along $S_2$.

## 5.3 Orthogonal Projectors

An orthogonal projector is one that projects onto a subspace $S_1$ along a space $S_2$, where $S_1$ and $S_2$ are orthogonal. (Warning: orthogonal projectors are not orthogonal matrices!)

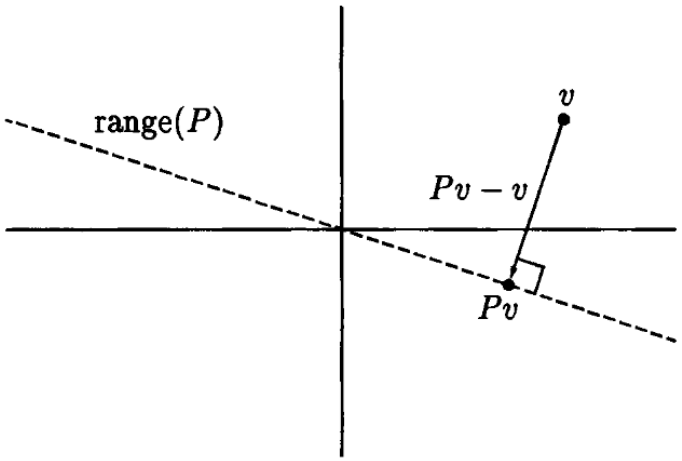



**Figure 6.2.** *An orthogonal projection.*

**Theorem 5.4.**
A projector $P$ is orthogonal if and only if $P = P^*$.

## 5.4 Projection with an Orthonormal Basis

Let $\{q_1, \ldots, q_n\}$ be any set of $n$ orthonormal vectors in $\mathbb{C}^m$, and let $\hat{Q}$ be the corresponding $m \times n$ matrix. Thus the map

$$v \mapsto \sum_{i=1}^{n} (q_i q_i^*)\, v \tag{5.1}$$

is an orthogonal projector onto range $(\hat{Q})$, and in matrix form, it may be written $y = \hat{Q}\hat{Q}^* v$.

An important special case of orthogonal projectors is the rank-one orthogonal projector that isolates the component in a single direction $q$, which can be written

$$P_q = qq^*.$$

These are the pieces from which higher-rank projectors can be made, as in (6.7). Their complements are the rank $m-1$ orthogonal projectors that eliminate the component in the direction of $q$ :

$$P_{\perp q} = I - qq^*.$$

Equations (6.8) and (6.9) assume that $q$ is a unit vector. For arbitrary nonzero vectors $a$, the analogous formulas are

$$P_a = \frac{aa^*}{a^*a},$$
$$P_{\perp a} = I - \frac{aa^*}{a^*a}.$$

## 5.5 Projection with an Arbitrary Basis

An orthogonal projector onto a subspace of $\mathbb{C}^m$ can also be constructed beginning with an arbitrary basis, not necessarily orthogonal. Suppose that the subspace is spanned by the linearly independent vectors $\{a_1, \ldots, a_n\}$, and let $A$ be the $m \times n$ matrix whose $j$ th column is $a_j$.

In passing from $v$ to its orthogonal projection $y \in \text{range}(A)$, the difference $y - v$ must be orthogonal to range $(A)$. This is equivalent to the statement that $y$ must satisfy $a_j^*(y - v) = 0$ for every $j$. Since $y \in \text{range}(A)$, we can set $y = Ax$ and write this condition as $a_j^*(Ax - v) = 0$ for each $j$, or equivalently, $A^*(Ax - v) = 0$ or $A^*Ax = A^*v$. It is easily shown that since $A$ has full rank, $A^*A$ is nonsingular (Exercise 6.3). Therefore

$$x = (A^*A)^{-1} A^*v$$

Finally, the projection of $v, y = Ax$, is $y = A(A^*A)^{-1} A^*v$. Thus the orthogonal projector onto range $(A)$ can be expressed by the formula

$$P = A(A^*A)^{-1} A^*.$$

# CHAPTER 6

# QR FACTORIZATION

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

## 6.1 Reduced QR Factorization

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

$$\langle a_1 \rangle \subseteq \langle a_1, a_2 \rangle \subseteq \langle a_1, a_2, a_3 \rangle \subseteq \cdots$$

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

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

$$\langle q_1, q_2, \ldots, q_j \rangle = \langle a_1, a_2, \ldots, a_j \rangle, \quad j = 1, \ldots, n.$$

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

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

where the diagonal entries $r_{kk}$ are nonzero. Written out, these equations take the form

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

As a matrix formula, we have

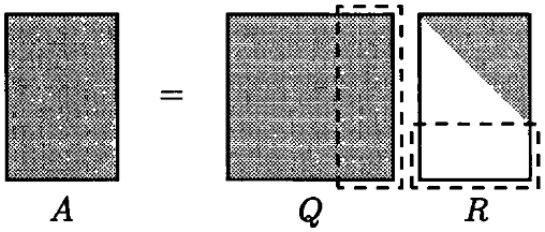
$$A = \hat{Q} \hat{R},$$

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

## 6.2 Full QR Factorization

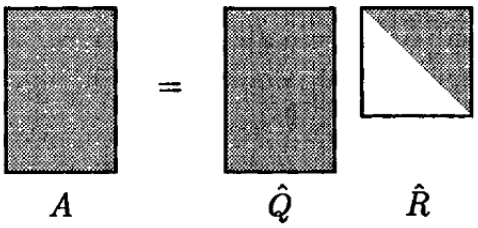
A full $QR$ factorization of $A \in \mathbb{C}^{m\times n}(m \geqslant n)$ goes further, appending an additional $m-n$ orthonormal columns to $\hat{Q}$ so that it becomes an $m \times m$ unitary matrix $Q$. This is analogous to the passage from the reduced to the full SVD. In the process, rows of zeros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

**Full QR Factorization ($m \geq n$)**




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

**Reduced QR Factorization ($m \geq n$)**




## 6.3 Gram-Schmidt Orthogonalization

Given $a_1, a_2, \ldots$, we can construct the vectors $q_1, q_2, \ldots$ and entries $r_{ij}$ by a process of successive orthogonalization. This is an old idea, known as GramSchmidt orthogonalization. The process is that:

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

where

$$
r_{ij} = q_i^* a_j \quad (i \neq j), \qquad |r_{jj}| = \left\| a_j - \sum_{i=1}^{j-1} r_{ij}q_i \right\|_2 .
$$

Note that the sign of $r_{jj}$ is not determined. Arbitrarily, we may choose $r_{jj} > 0$, in which case we shall finish with a factorization $A = \hat{Q}\hat{R}$ in which $\hat{R}$ has positive entries along the diagonal. The algorithm is the Gram-Schmidt iteration. Mathematically, it offers a simple route to understanding

and proving various properties of QR factorizations. Numerically, it turns out to be unstable because of rounding errors on a computer. To emphasize the instability, numerical analysts refer to this as the classical Gram-Schmidt iteration, as opposed to the modified Gram-Schmidt iteration, discussed in the next lecture.

**Algorithm 6.1:** Classical Gram Schmidt (unstable)

```
for j = 1 to n do
    v_j = a_j;
    for i = 1 to j − 1 do
        r_ij = q_i^* a_j;
        v_j = v_j − r_ij q_i ;
    r_jj = ‖v_j‖_2;
    q_j = v_j / r_jj;
```

## 6.4 Existence and Uniqueness

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

**Theorem 6.1.**
Every $A \in \mathbb{C}^{m\times n} (m \geqslant n)$ has a full $QR$ factorization, hence also a reduced $QR$ factorization.

**Theorem 6.2.**
Each $A \in \mathbb{C}^{m\times n} (m \geqslant n)$ of full rank has a unique reduced $QR$ factorization $A = \hat{Q}\hat{R}$ with $r_{jj} > 0$.

## 6.5 Solution of $Ax = b$ by QR Factorization

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

$$Rx = Q^*b.$$

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

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

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

# CHAPTER 7

# GRAM-SCHMIDT ORTHOGONALIZATION

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

## 7.1 Gram-Schmidt Projections

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

Let $A \in \mathbb{C}^{m\times n}, m \geqslant n$, be a matrix of full rank with columns $\{a_j\}$. Consider now the sequence of formulas

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

In these formulas, each $P_j$ denotes an orthogonal projector. Specifically, $P_j$ is the $m \times m$ matrix of rank $m-(j-1)$ that projects $\mathbb{C}^m$ orthogonally onto the space orthogonal to $\langle q_1, \ldots, q_{j-1}\rangle$. (In the case $j = 1$, this prescription reduces to the identity: $P_1 = I$.) The projector $P_j$ can be represented explicitly. Let $\hat{Q}_{j-1}$ denote the $m\times$ $(j-1)$ matrix containing the first $j-1$ columns of $\hat{Q}$. Then $P_j$ is given by

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

## 7.2 Modified Gram-Schmidt Algorithm

In practice, the Gram-Schmidt formulas are not applied as we have indicated in algorithm 6.1, for this sequence of calculations turns out to be numerically unstable. Fortunately, there is a simple modification that improves matters. We have not discussed numerical stability yet; this will come in the next lecture and then systematically beginning in Chapter 13. By the definition of $P_j$, it's not difficult to see that:

$$P_j = P_{\perp qj-1} \cdots P_{\perp q2} P_{\perp q1}. \tag{7.1}$$

Thus, an equivalent statement is that:

$$v_j = P_{\perp q_{j-1}} \cdots P_{\perp q2} P_{\perp q_1} a_j.$$

This leads to the modified Gram-Schmidt algorithm. Mathematically, these two methods are equivalent. However, the sequences of arithmetic operations implied by these formulas are different. The modified

algorithm calculates $v_j$ by evaluating the following formulas in order:

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

In finite precision computer arithmetic, we shall see that the modified method introduces smaller errors than GS method.

**Algorithm 7.1:** Modified Gram-Schmidt

```
for i = 1 to n do
    v_i = a_i;
for i = 1 to n do
    r_ii = ‖v_i‖;
    q_i = v_i / r_ii ;
    for j = i + 1 to n do
        r_ij = q_i^* v_j;
        v_j = v_j − r_ij q_i;
```

## 7.3 Operation Count

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

**Theorem 7.1.**
Algorithms 6.1 and 7.1 require $\sim 2mn^2$ flops to compute a QR factorization of an $m \times n$ matrix.

Theorem 7.1 can be established as follows. To be definite, consider the modified Gram-Schmidt algorithm, Algorithm 7.1. When $m$ and $n$ are large, the work is dominated by the operations in the innermost loop:

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

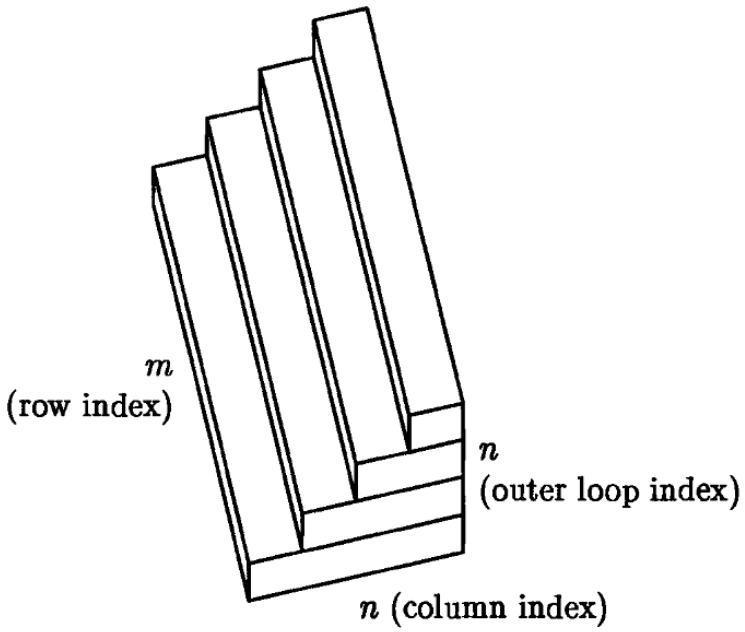
The first line computes an inner product $q_i^* v_j$, requiring $m$ multiplications and $m-1$ additions, and the second computes $v_j - r_{ij} q_i$, requiring $m$ multiplications and $m$ subtractions. The total work involved in a single inner iteration is consequently $\sim 4m$ flops, or 4 flops per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

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

## 7.4 Counting Operations Geometrically

Operation counts can always be determined algebraically, and this is the standard procedure in the numerical analysis literature. However, it is also enlightening to take a different, geometrical route to

the same conclusion. The argument goes like this. At the first step of the outer loop, Algorithm 7.1 operates on the whole matrix, subtracting a multiple of column 1 from the other columns. At the second step, it operates on a submatrix, subtracting a multiple of column 2 from columns $3, \ldots, n$. Continuing on in this way, at each step the column dimension shrinks by 1 until at the final step, only column $n$ is modified. This process can be represented by the following diagram:




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

## 7.5 Gram-Schmidt as Triangular Orthogonalization

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

$$\begin{bmatrix} \mathbf{v_1} & \mathbf{v_2} & \cdots & \mathbf{v_n} \end{bmatrix} \begin{bmatrix} \frac{1}{r_{11}} & \frac{-r_{12}}{r_{11}} & \frac{-r_{13}}{r_{11}} & \cdots \\ & 1 & & \\ & & 1 & \\ & & & \ddots \end{bmatrix} = \begin{bmatrix} q_1 & v_2^{(2)} & \cdots & v_n^{(2)} \end{bmatrix}.$$

In general, step $i$ of Algorithm 7.1 subtracts $r_{ij}/r_{ii}$ times column $i$ of the current $A$ from columns $j > i$ and replaces column $i$ by $1/r_{ii}$ times itself. This corresponds to multiplication by an upper-

triangular matrix $R_i$ :

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

At the end of the iteration we have

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

This formulation demonstrates that the Gram-Schmidt algorithm is a method of triangular orthogonalization. It applies triangular operations on the right of a matrix to reduce it to a matrix with orthonormal columns. Of course, in practice, we do not form the matrices $R_i$ and multiply them together explicitly. The purpose of mentioning them is to give insight into the structure of the Gram-Schmidt algorithm. In Chapter 19 we shall see that it bears a close resemblance to the structure of Gaussian elimination.

# CHAPTER 8

# 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.

## 8.1 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.

## 8.2 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 $r_{jj}$.

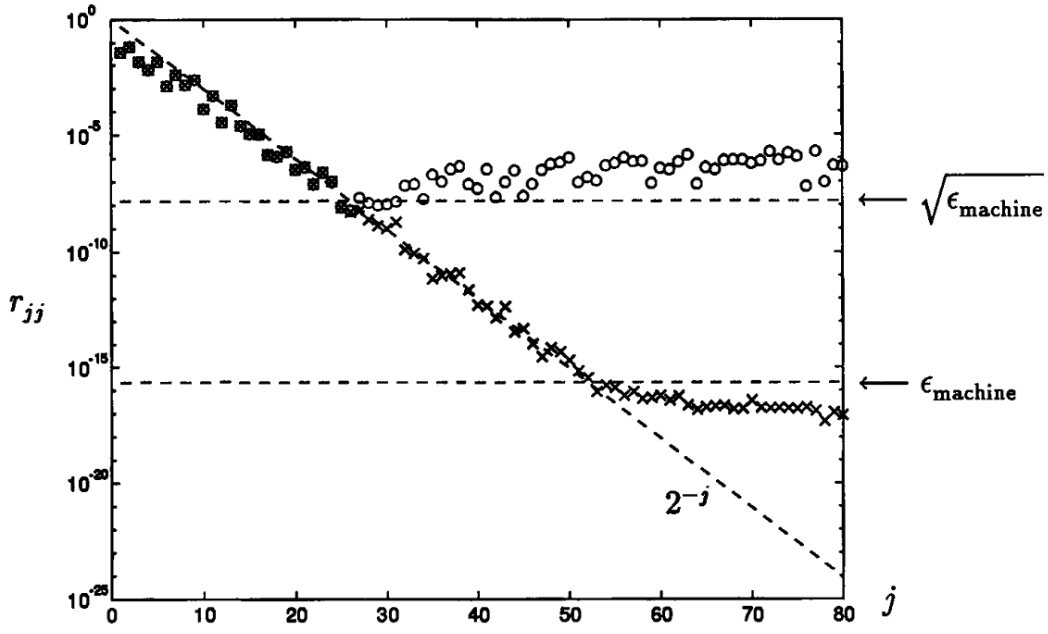



Figure 8.1: The classical GS is represented by circles while the modified GS is represented by crosses.

Notice that:

- $r_{jj}$ is close to $2^{-j}$. This is because

$$A = 2^{-1}u_1v_1^* + 2^{-2}u_2v_2^* + 2^{-3}u_3v_3^* + \cdots + 2^{-80}u_{80}v_{80}^*,$$
$$a_j = 2^{-1}\bar{v}_{j1}u_1 + 2^{-2}\bar{v}_{j2}u_2 + 2^{-3}\bar{v}_{j3}u_3 + \cdots + 2^{-80}\bar{v}_{j,80}u_{80}.$$

  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.

## 8.3 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.

```
1    A = [.70000 .70711                                   Define A.
2    .70001 .70711] ;
3    [Q, R] = qr(A);                          Compute factor Q by Householder.
4    norm(Q'*Q-eye(2))                                 Test orthogonality of Q.
5    [Q, R] = mgs(A);                         Compute factor Q by modified GS.
6    norm(Q'*Q - eye(2))                               Test orthogonality of Q.
```

The results are:

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

# CHAPTER 9

# 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.

## 9.1 Householder and Gram-Schmidt

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

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

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

$$\underbrace{Q_n \cdots Q_2 Q_1}_{Q^*} A = R.$$

The two methods can be summerized as the follows:

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

## 9.2 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.

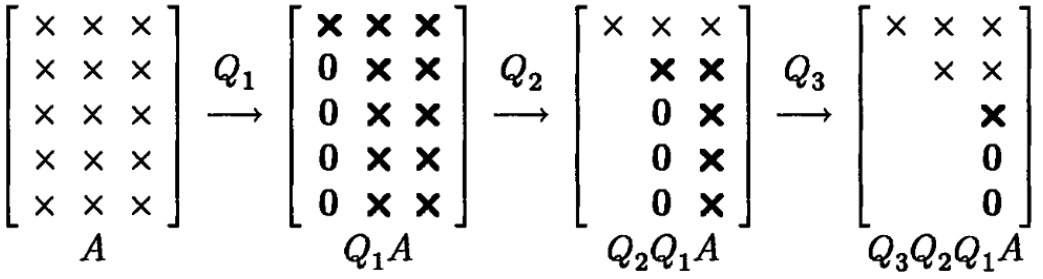



Figure 9.1: One example in $\mathbb{R}^{5\times 3}$

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.

## 9.3 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 $k, \ldots, m$ 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 $v = \|x\|e_1 - x$.

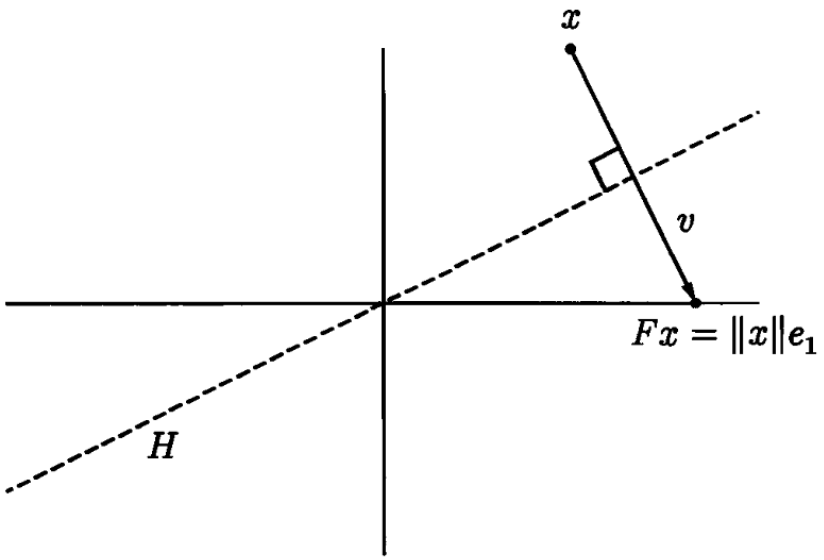



Figure 9.2: A Householder reflection

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.

## 9.4 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.

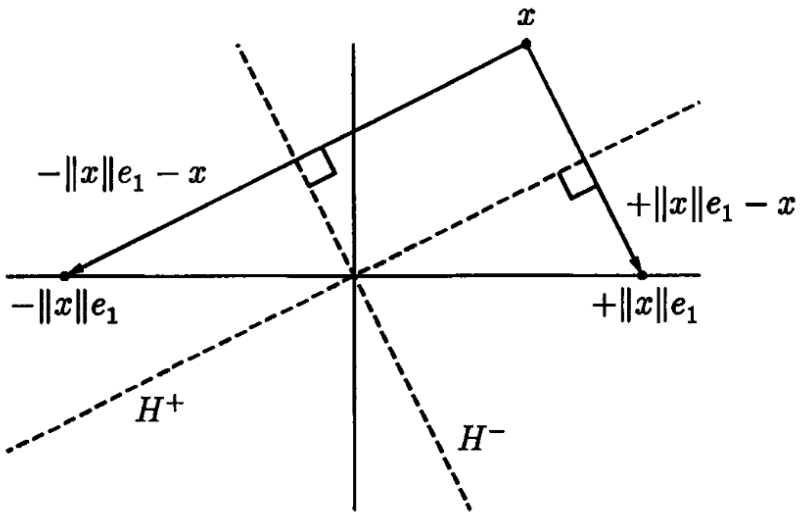



Figure 9.3: Two possible reflections

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 $\operatorname{sign}(x_1) = 1$.

## 9.5 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 9.1:** Householder QR factorization

```
for k = 1 to n do
    x = A_{k:m,k};
    v_k = sign(x_1) ‖x‖_2 e_1 + x;
    v_k = v_k / ‖v_k‖_2 ;
    A_{k:m,k:n} = A_{k:m,k:n} − 2v_k (v_k^* A_{k:m,k:n});
```

## 9.6 Applying or Forming $Q$

The algorithm 9.1 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 9.2:** Implicit Calculation of a Product $Q^*b$

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

**Algorithm 9.3:** Implicit Calculation of a Product $Qx$

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

The work involved in either of these algorithms is of order $O(mn)$, not $O\left(mn^2\right)$ as in algorithm 9.1 (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 9.3. 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 $Qe_1, Qe_2, \ldots, Qe_n$.

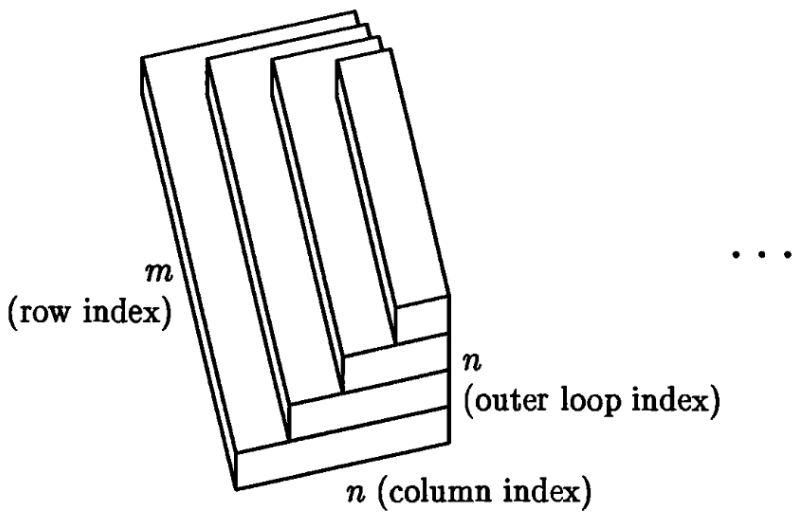
## 9.7 Operation Count

The work involved in algorithm 9.1 is dominated by the innermost loop,

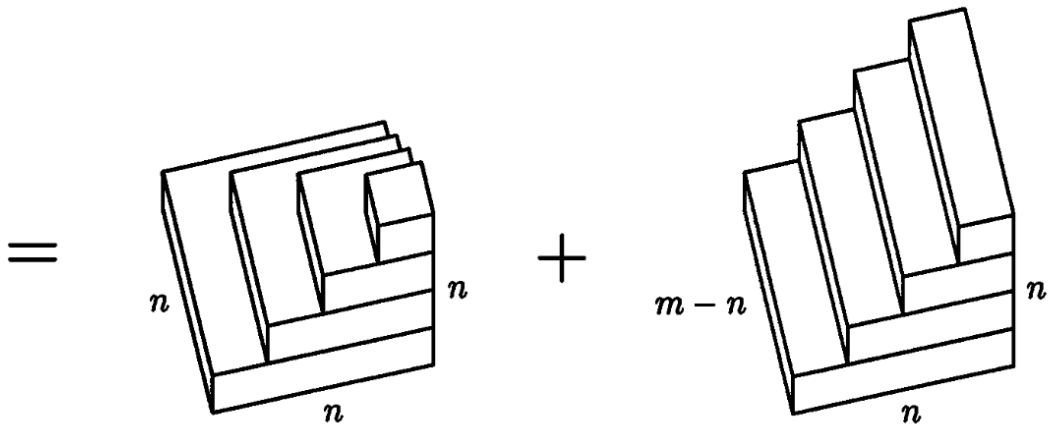
$$A_{k:m,j} - 2v_k \left(v_k^* A_{k:m,k}\right)$$

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 9.1.**
Work for Householder orthogonalization: $\sim 2mn^2 - \frac{2}{3}n^3$ flops.

# CHAPTER 10

# 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 $b - Ax$.

## 10.1 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:

$$\min_{x \in \mathbb{C}^n} \quad \|b - Ax\|_2 \tag{10.1}$$

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 $b$.

## 10.2 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.

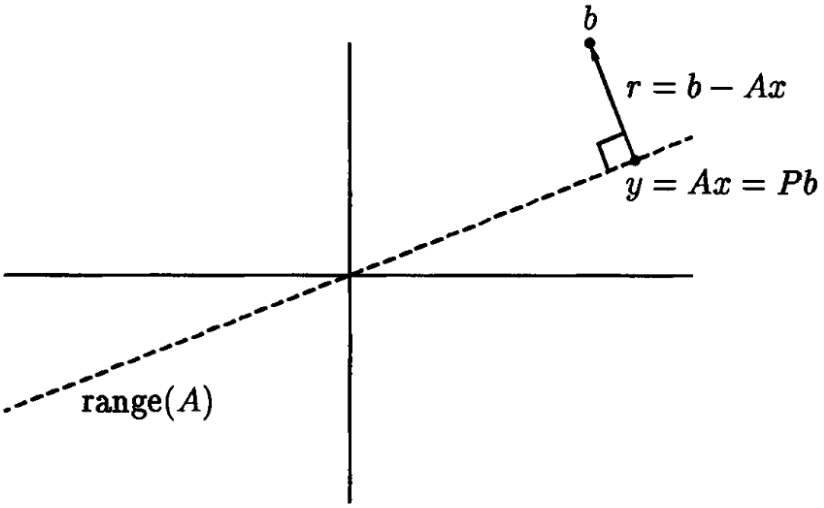



Figure 10.1: Formulation of LS in terms of orthogonal projection

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 10.1.**
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 10.1 if and only if $r \perp$ range$(A)$, that is,

$$A^*r = 0$$

or equivalently,

$$A^*Ax = A^*b \tag{10.2}$$

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 10.2, 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.

## 10.3 Pseudoinverse

**Definition 10.2 (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:

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

## 10.4 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 10.1:** 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 10.3.**
The work for algorithm 10.1: $\sim mn^2 + \frac{1}{3}n^3$ flops.

## 10.5 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 10.2:** 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 10.2 is dominated by the cost of QR. If we use Householder reflections, we have

**Corollary 10.4.**
Work for algorithm 10.2: $\sim 2mn^2 - \frac{2}{3}n^3$ flops.

## 10.6 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 10.3:** 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 10.5.**
Work for algorithm 10.3: $\sim 2mn^2 + 11n^3$ flops.

## 10.7 Comparison of Algorithms

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

- algorithm 10.1 is the fastest but solving the normal equations might not be stable;
- Thus for many years, numerical analysts have recommended algorithm 10.2 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 10.2 itself has less-than-ideal stability properties, and in such cases there are good reasons to turn to algorithm 10.3 based on the SVD.

# Part III

# Conditioning and Stability

# CHAPTER 11

# 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.

## 11.1 Condition of a Problem

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

**Definition 11.1** (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 $f(x)$.

## 11.2 Absolute Condition Number

**Definition 11.2** (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 $Y$.

## 11.3 Relative Condition Number

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

**Definition 11.3** (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 11.4.**

- 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 11.5** (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",

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

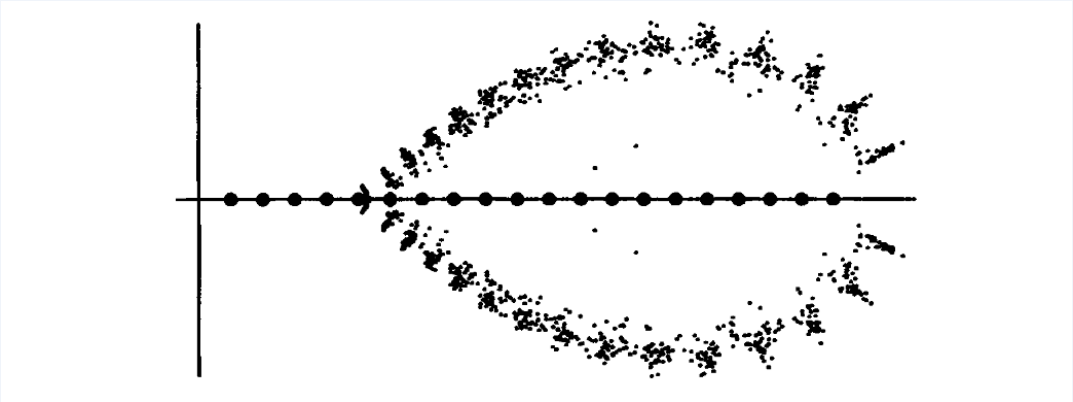


Figure 11.1: Wilkinson's classic example of ill-conditioning. The large dots are the roots of the unperturbed polynomial. The samll dots are the super imposed roots in the complex plane of 100 randomly perturbed polynomials with coefficients defined by $\overline{a_k} = a_k(1 + 10^{-10}r_k)$, where $r_k \sim \mathcal{N}(0, 1)$.

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:

$$\kappa \approx \frac{1.67 \times 10^8 \cdot 15^1 4}{5!14!} \approx 5.1 \times 10^{13}.$$

## 11.4 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 11.6.**
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

$$\kappa = \|A\| \frac{\|x\|}{\|b\|} \leqslant \|A\| \left\|A^{-1}\right\| \tag{11.1}$$

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

$$\kappa = \left\|A^{-1}\right\| \frac{\|b\|}{\|x\|} \leqslant \|A\| \left\|A^{-1}\right\| \tag{11.2}$$

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 $\sigma_1$.

## 11.5 Condition Number of a Matrix

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

**Definition 11.7** (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

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

## 11.6 Condition of a System of Equations

In Theorem 11.6 , 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,

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

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

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

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 11.8.**
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

$$\kappa = \|A\|\,\|A^{-1}\| = \kappa(A). \tag{11.3}$$

**Note 11.9.**
Theorems 11.6 and 11.8 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 12

# 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.

## 12.1 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

$$1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, 1 + 3 \times 2^{-52}, \ldots, 2. \tag{12.1}$$

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

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

and in general, the interval $[2^j, 2^{j+1}]$ is represented by Equation 12.1 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!

## 12.2 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: $\mathbf{F} = \beta\mathbf{F}$.

## 12.3 Machine Epsilon

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

$$\epsilon_{\text{machine}} = \frac{1}{2}\beta^{1-t}. \tag{12.2}$$

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 12.1** (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 12.1 can be rewritten as:

**Proposition 12.2.**
For all $x \in \mathbb{R}$, there exists $\epsilon$ with $|\epsilon| \leqslant \epsilon_{\text{machine}}$ such that

$$\text{fl}(x) = x(1 + \epsilon). \tag{12.3}$$

## 12.4 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 12.3** (Fundamental Axiom of FPA).
For all $x, y \in \mathbf{F}$, there exists $\epsilon$ with $|\epsilon| \leqslant \epsilon_{\text{machine}}$ such that

$$x \circledast y = (x * y)(1 + \epsilon). \tag{12.4}$$

## 12.5 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 12.2 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 13

# 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.

## 13.1 Algorithms

Recall Definition 11.1, we can define algorithms.

**Definition 13.1** (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., $f(x) = g(\text{fl}(x))$. And $f$ maps $X$ to $\text{fl}(Y)$.

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.

## 13.2 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**,

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \tag{13.1}$$

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 $x \in X$,

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = O\left(\epsilon_{\text{machine}}\right) \tag{13.2}$$

## 13.3 Stability

If the problem $f$ is ill-conditioned, however, the goal of accuracy as defined by Equation 13.2 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 13.2** (Stability).
We say that an algorithm $\tilde{f}$ for a problem $f$ is stable if for each $x \in X$,

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O\left(\epsilon_{\text{machine}}\right), \tag{13.3}$$

for some $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O\left(\epsilon_{\text{machine}}\right). \tag{13.4}$$

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.

## 13.4 Backward Stability

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

**Definition 13.3** (Backward stable).
We say that an algorithm $\tilde{f}$ for a problem $f$ is backward stable if for each $x \in X$,

$$\tilde{f}(x) = f(\tilde{x}) \text{ for some } \tilde{x} \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = O\left(\epsilon_{\text{machine}}\right). \tag{13.5}$$

In words,

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

Examples are given in the next chapter.

## 13.5 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 13.4.**
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 14

# 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.

## 14.1 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 $\oslash$ provided with the computer. As it happens, the axioms (12.4) and (12.3) 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 12.4, 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 $|\epsilon_4|, |\epsilon_4| \leqslant 2\epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}})$. 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 14.1.**

- 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.

## 14.2 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 11.5, 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 11.5. 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!

## 14.3 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 14.2** (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 (12.3) and (12.4). Then the relative errors satisfy

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = O(\kappa(x)\epsilon_{\text{machine}}). \tag{14.1}$$

*Proof.*
By the Definition 13.3 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 11.3, 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. □

## 14.4 Backward Error Analysis

The process we have just carried out in proving Theorem 14.2 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 15

# 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 $Ax = b$.

## 15.1 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, $\epsilon_{\text{machine}} \approx 1.11 \times 10^{-16}$.

```
1 R = triu(randn(50));          Set R to a 50 × 50 upper-triangular matrix with normal random entries.
2 [Q,X] = qr(randn(50));     Set Q to a random orthogonal matrix by orthogonalizing a random matrix.
3 A=Q*R;                                          Set A to the product QR, up to rounding errors.
4 [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:

```
1 norm(Q2-Q)                                                          How accurate is Q2?
2    ans = 0.00889
3 norm(R2-R)/norm(R)                                                  How accurate is R2?
4    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 $R_2$:

```
1 norm(A-Q2*R2)/norm(A)                                               How accurate is Q2R2?
2    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 Q3 to a random perturbation of Q
                                % that is closer to Q than Q2 is.
R3 = R+1e-4*randn(50);          % Set R3 to a random perturbation of R
                                % that is closer to R than R2 is.
norm(A-Q3*R3)/norm(A)           % How accurate is Q3R3?
    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 14.2). 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.

## 15.2 Theorem

In fact, Householder triangularization is backward stable for all matrices $A$ and all computers satisfying (12.3) and (12.4). 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 15.1** (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 (12.3) and (12.4), and let the computed factors $\tilde{Q}$ and $\tilde{R}$ be defined as indicated above. Then, we have

$$\tilde{Q}\tilde{R} = A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}) \tag{15.1}$$

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

## 15.3 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 10.2.

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 10.2 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 15.1.

The second step is computation of $\tilde{Q}^*b$ by algorithm 9.2. 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:

$$(\tilde{Q} + \delta Q)\tilde{y} = b, \quad \|\delta Q\| = O(\epsilon_{\text{machine}}). \tag{15.2}$$

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:

$$(\tilde{R} + \delta R)\tilde{x} = \tilde{y}, \quad \frac{\|\delta R\|}{\|\tilde{R}\|} = O(\epsilon_{\text{machine}}). \tag{15.3}$$

As always, the equality on the left is exact. We will derive this in full detail in the next chapter. Now, combine (15.1) (15.2) and (15.3), we get the following theorem.

**Theorem 15.2** (Backstap of linear system).
algorithm 10.2 is backward stable, satisfying

$$(A + \Delta A)\tilde{x} = b, \quad \frac{\|\Delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}) \tag{15.4}$$

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

*Proof.*
Combine (15.3) and (15.2), 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 (15.1),

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

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 (15.1). (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 (15.2). Similarly,

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

by (15.3). 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 11.8, Theorem 14.2 and Theorem 15.2 gives the following result about accuracy of solutions of $Ax = b$.

**Theorem 15.3** (Error of LS).
The solution $\tilde{x}$ computed by algorithm 10.2 satisfies

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\kappa(A)\epsilon_{\text{machine}}). \tag{15.5}$$

# CHAPTER 16

# 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 "$O(\epsilon_{\text{machine}})$".

## 16.1 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,

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ & r_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & r_{mm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}, \tag{16.1}$$

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 16.1:** Back Substitution

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

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 16.1.**
The work for back substitution: $\sim m^2$ flops.

## 16.2 Backward Stability Theorem

Now we will try to show (15.3), 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 16.2 (Backstap of backsub).**
Let algorithm 16.1 be applied to a problem (16.1) consisting of floating point numbers on a computer satisfying (12.4). This algorithm is backward stable in the sense that the computed solution $\tilde{x} \in \mathbb{C}^m$ satisfies

$$(R + \delta R)\tilde{x} = b \tag{16.2}$$

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

$$\frac{\|\delta R\|}{\|R\|} = O(\epsilon_{\text{machine}}). \tag{16.3}$$

Specifically, for each $i, j$,

$$\frac{|\delta r_{ij}|}{|r_{ij}|} \leqslant m\epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}). \tag{16.4}$$

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

## 16.3 $m = 1$

When $d = 1$, the problem is:

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

The axiom (12.4) for $\oslash$ 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,

$$\tilde{x}_1 = \frac{b_1}{r_{11}(1 + \epsilon_1')}, \quad |\epsilon_1'| \leqslant \epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}). \tag{16.5}$$

Hence, we have

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

with $\Delta r_{11} = \epsilon_1' r_{11}$. In other words,

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

## 16.4 $m = 2$

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:

$$\tilde{x}_2 = b_2 \oslash r_{22} = \frac{b_2}{r_{22}(1 + \epsilon_1)}, \quad |\epsilon_1| \leqslant \epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}}). \tag{16.6}$$

The second step is defined by the formula

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

In fact, if we apply (12.4), 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,

$$\tilde{x}_1 = \frac{b_1 - \tilde{x}_2 r_{12}\,(1+\epsilon_2)}{r_{11}\,(1+2\epsilon_5)}, \tag{16.7}$$

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.

## 16.5 $m = 3$

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:

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

where

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

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 $\oslash$ 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

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

## 16.6 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

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

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

$$\oslash: \quad \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}.$$

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

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

Adding them produces the result we want. This completes the proof of Theorem 16.2.

# CHAPTER 17

# 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.

## 17.1 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:

$$\begin{aligned} &\text{Given } A \in \mathbb{C}^{m\times n} \text{ of full rank, } m \geqslant n, b \in \mathbb{C}^m, \\ &\text{find } x \in \mathbb{C}^n \text{ such that } \|b - Ax\| \text{ is minimized.} \end{aligned} \tag{17.1}$$

The solution is given by,

$$x = A^\dagger b, \quad y = Pb. \tag{17.2}$$

We consider the conditioning of (17.1) 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.

## 17.2 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 17.1** (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 (17.1) 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 17.2.**
In the special case $m = n$, (17.1) 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 (11.2) and (11.3) derived earlier.

## 17.3 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

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

## 17.4 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.

## 17.5 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 $\delta b = e_n$.

## 17.6 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

$$\delta\alpha \leqslant \frac{\|\delta A\|}{\sigma_n} = \frac{\|\delta A\|}{\|A\|}\kappa(A), \tag{17.3}$$

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

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

We begin with the left-hand entry in the second row of the table in Theorem 17.1. 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 $\delta\alpha$.

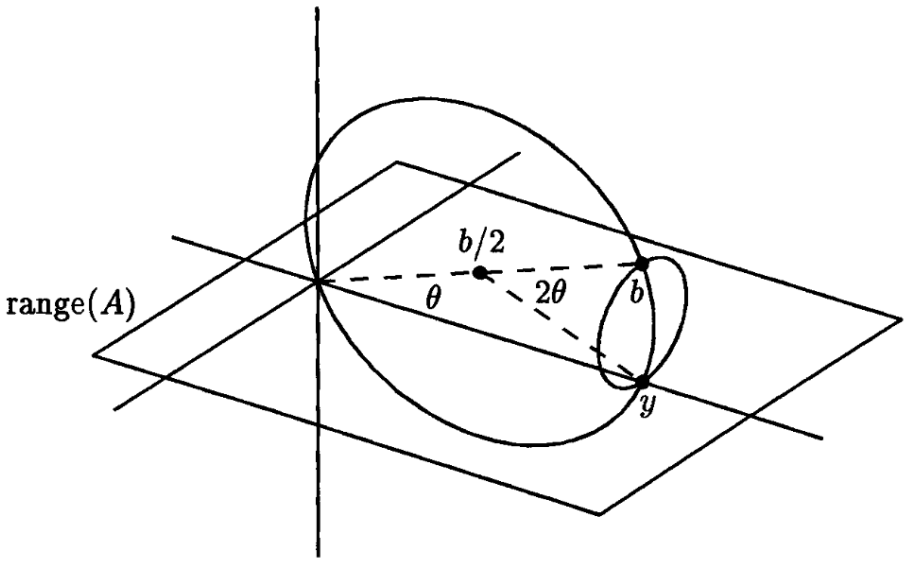



Figure 17.1: Two circles on the sphere along which $y$ moves as range($A$) varies. The large circle, of radius $\|b\|/2$, corresponds to tilting range($A$) in the plane $0 - b - y$, and the small circle of radius $(\|b\|/2)\sin\theta$, corresponds to tilting it in an orthogonal direction. However range($A$) 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

$$\|\delta y\| \leqslant \|b\| \sin(\delta\alpha) \leqslant \|b\| \delta\alpha. \tag{17.4}$$

By the definition of $\theta$ and (17.3), we have

$$\|\delta y\| \leqslant \frac{\|\delta A\|}{\|A\|} \kappa(A) \cdot \frac{\|y\|}{\cos\theta} \Rightarrow \frac{\|\delta y\|}{\|y\|} \Big/ \frac{\|\delta A\|}{\|A\|} \leqslant \frac{\kappa(A)}{\cos\theta}. \tag{17.5}$$

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

We are now ready to analyze the most interesting relationship of Theorem 17.1: 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 (17.1) without changing $b_1$. The condition for such perturbations is given by (11.3), which here takes the form

$$\frac{\|\delta x\|}{\|x\|} \Big/ \frac{\|\delta A_1\|}{\|A\|} \leqslant \kappa(A_1) = \kappa(A). \tag{17.6}$$

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 (11.2), which takes the form

$$\frac{\|\delta x\|}{\|x\|} \Big/ \frac{\|\delta b_1\|}{\|b_1\|} \leqslant \frac{\kappa(A_1)}{\eta(A_1; x)} = \frac{\kappa(A)}{\eta}. \tag{17.7}$$

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 (17.4), 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 (17.3) 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 (17.6) establishes the lower-right result of Theorem 17.1.

# CHAPTER 18

# 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.

## 18.1 Example

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

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

Now we check the quantities:

```
1  x = A\b; y = A*x;                                           Solve LS problem.
2  kappa = cond(A)
3     kappa = kappa = 2.2718e+10                                              κ(A)
4  theta = asin(norm(b-y)/norm(b))
5     theta = 3.7461e-06                                                      θ
6  eta = norm(A)*norm(x)/norm(y)
7     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 | $1.1 \times 10^5$ |
| $A$ | $2.3 \times 10^{10}$ | $3.2 \times 10^{10}$ |

## 18.2 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 10.2 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 9.2. 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 18.1** (Backstap of Householder for LS).
Let the full-rank least squares problem (10.1) be solved by Householder triangularization (Algorithm 10.2) on a computer satisfying (12.3) and (12.4). This algorithm is backward stable in the sense that the computed solution $\tilde{x}$ has the property

$$\|(A+\delta A)\tilde{x} - b\| = \min, \quad \frac{\|\delta A\|}{\|A\|} = O\left(\epsilon_{\text{machine}}\right) \tag{18.1}$$

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 9.2. It also holds for Householder triangularization with arbitrary column pivoting.

## 18.3 GS Orthogonalization

Another way to solve a least squares problem is by modified Gram-Schmidt orthogonalization (Algorithm 7.1). 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 18.2** (Backstap of GS for LS).
The solution of the full-rank least squares problem (10.1) by Gram-Schmidt orthogonalization is also backward stable, satisfying (18.1), provided that $\hat{Q}^*b$ is formed implicitly as indicated in the code segment above.

## 18.4 Normal Equations

A fundamentally different approach to least squares problems is the solution of the normal equations (Algorithm 10.1), 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 14.2 of and Thm 17.1 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 18.3** (Stability of normal equations).
The solution of the full-rank least squares problem (10.1) via the normal equations (Algorithm 10.1) 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.

## 18.5 SVD

SVD is stable.

```
1 [U,S,V] = svd(A,0);
2 x = V*(S\(U'*b));
3 x(15)
4    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 18.4** (Backstap of SVD).
The solution of the full-rank least squares problem (10.1) by the SVD (Algorithm 10.3) is backward stable, satisfying the estimate (18.1).

## 18.6 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 10.1 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 IV

# Systems of Equations

# CHAPTER 19

# 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.

## 19.1 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 $L_k$:

$$\underbrace{L_{m-1}\cdots L_2 L_1}_{L^{-1}} A = U. \tag{19.1}$$

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

$$A = LU, \tag{19.2}$$

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 19.1.**
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.

## 19.2 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

$$L_1 A = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}.$$

Then

$$L_2 L_1 A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}.$$

Finally,

$$L_3 L_2 L_1 A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = U.$$

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:

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

## 19.3 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:

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

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

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

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 19.1:** Gaussian Elimination without Pivoting

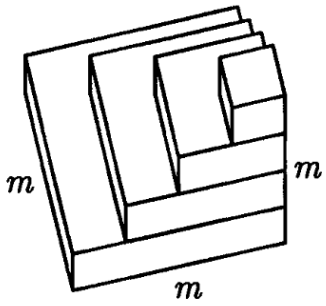
```
1 U = A, L = I;
2 for k = 1 to m − 1 do
3     for j = k + 1 to m do
4         l_jk = u_jk / u_kk;
5         u_{j,k:m} = u_{j,k:m} − l_jk u_{k,k,:m}
```

**Note 19.2.**
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 $A$.

## 19.4 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 $k + 1, \ldots, m$. 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 19.3.**
Work fo Gaussian elimination: $\sim \frac{2}{3}m^3$ flops.

## 19.5 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 19.4.**
**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.

## 19.6 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

$$A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}. \tag{19.3}$$

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 19.5.**
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 20

# 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.

## 20.1 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 $x_{kk}$:

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

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

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

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

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

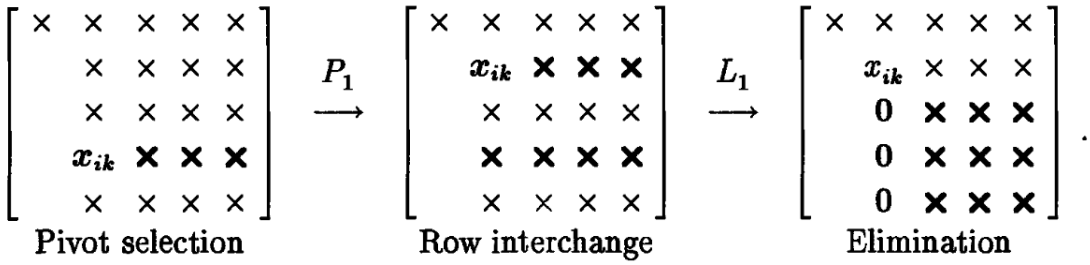
All in all, we are free to choose any entry of $X_{k:m,k:m}$ as the pivot, as long as it's nonzero. For numerical stability, it's desirable to pivot even when $x_{kk}$ is nonzero if there is a larger element available. In practice, we just pick as pivot the largest number among a set of entries. Now 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 20.1.**
> 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.

## 20.2 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 $U$:

$$L_{m-1}P_{m-1}\cdots L_2P_2L_1P_1A = U. \tag{20.1}$$

## 20.3 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:

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

## 20.4 $PA = LU$ 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:

$$L_3 P_3 L_2 P_2 L_1 P_1 A = U,$$

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

$$PA = LU. \tag{20.2}$$

In general, any square matrix $A$, singular or nonsingular, has a factorization (20.2), 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 20.1:** Gaussian Elimination with Partial Pivoting

```
U = A, L = I, P = I;
for k = 1 to m − 1 do
    Select i ⩾ k to maximize |u_ik|;
    u_{k,k:m} ↔ u_{i,k:m} (interchange two rows);
    l_{k,1:k−1} ↔ l_{i,1:k−1} ;
    p_{k,:} ↔ p_{i,:} ;
    for j = k + 1 to m do
        l_jk = u_jk/u_kk;
        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 19.1, 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.

## 20.5 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 21

# 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.

## 21.1 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 (19.3), 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 21.1.**
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 19.1) on a computer satisfying the axioms (12.3) and (12.4). 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

$$\tilde{L}\tilde{U} = A + \delta A, \quad \frac{\|\delta A\|}{\|L\|\|U\|} = O\left(\epsilon_{\text{machine}}\right) \tag{21.1}$$

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.

## 21.2 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, (21.1) 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

$$\rho = \frac{\max_{i,j} |u_{ij}|}{\max_{i,j} |a_{ij}|}. \tag{21.2}$$

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

**Corollary 21.2.**
Let the factorization $PA = LU$ of a matrix $A \in \mathbb{C}^{m\times m}$ be computed by Gaussian elimination with partial pivoting (algorithm 20.1) on a computer satisfying the axioms (12.3) and (12.4). 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 21.3.**
Is Gaussian elimination backward stable? According to Corollary 21.2 and our definition (13.5) 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.

## 21.3 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

$$A = \begin{bmatrix} 1 & & & & 1 \\ -1 & 1 & & & 1 \\ -1 & -1 & 1 & & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}. \tag{21.3}$$

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

$$U = \begin{bmatrix} 1 & & & & 1 \\ & 1 & & & 2 \\ & & 1 & & 4 \\ & & & 1 & 8 \\ & & & & 16 \end{bmatrix}. \tag{21.4}$$

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 21.4.**
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.

## 21.4 Stability in Practice

However, despite examples like (21.3), Gaussian elimination with partial pivoting is utterly stable in practice. Large factors $U$ like (21.4) 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.

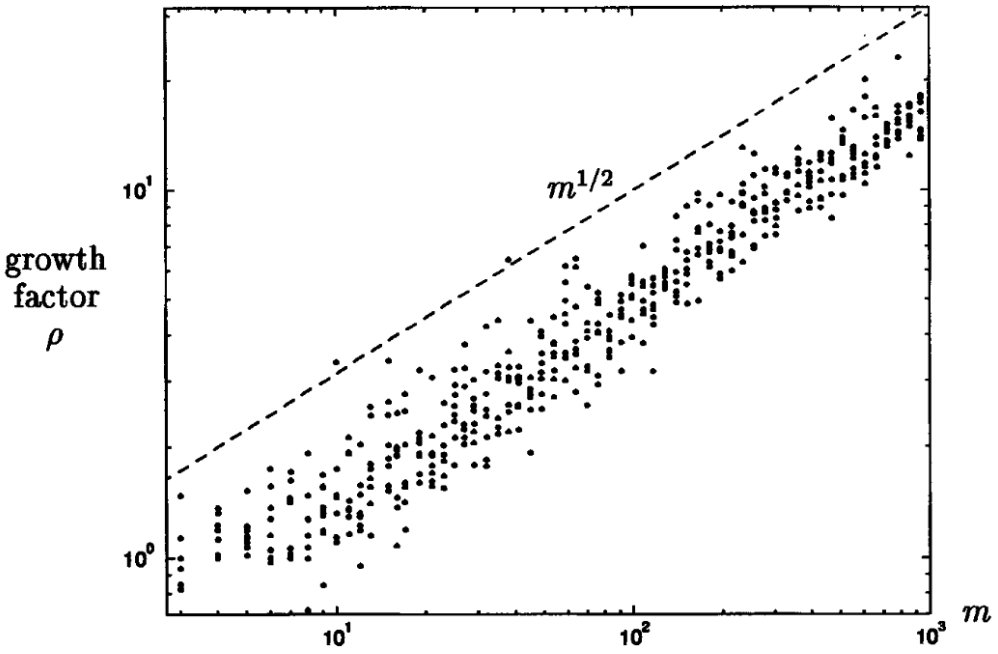



Figure 21.1: Growth factors for Gaussian elimination with partial pivoting applied to 496 random matrices of various dimensions. The typical size of $\rho$ is of order $m^{\frac{1}{2}}$, much less than the maximal possible value $2^{m-1}$.

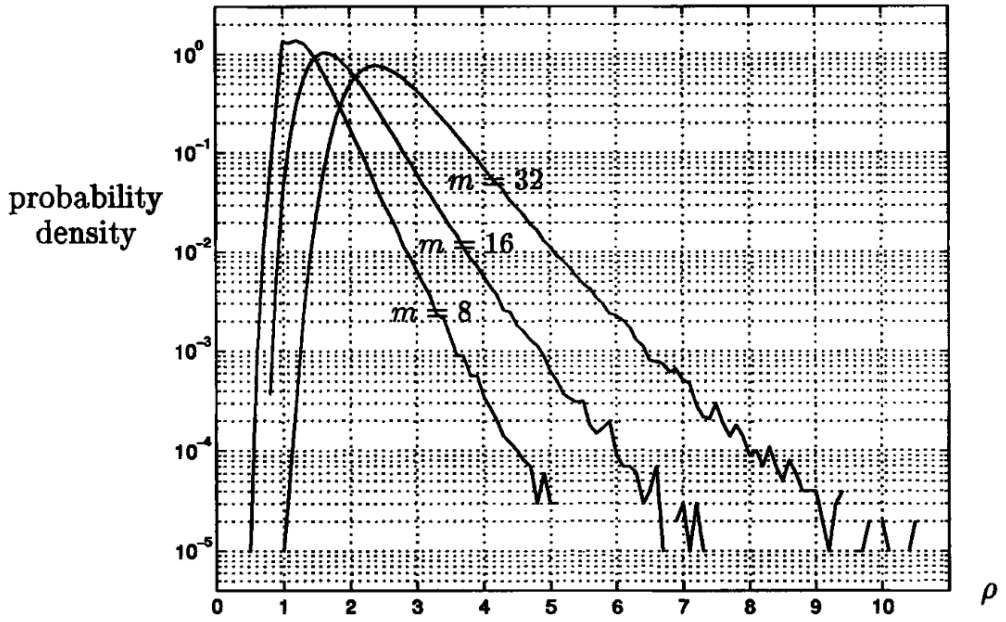



Figure 21.2: Probability distributions for growth factors of random matrices of dimensions $m = 8, 16, 32$, based on sample sizes of one million for each dimension. The density appears to decrease exponentially with $\rho$.

Figures 21.1 and 21.2 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 21.1, 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 21.2, 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.

## 21.5 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.

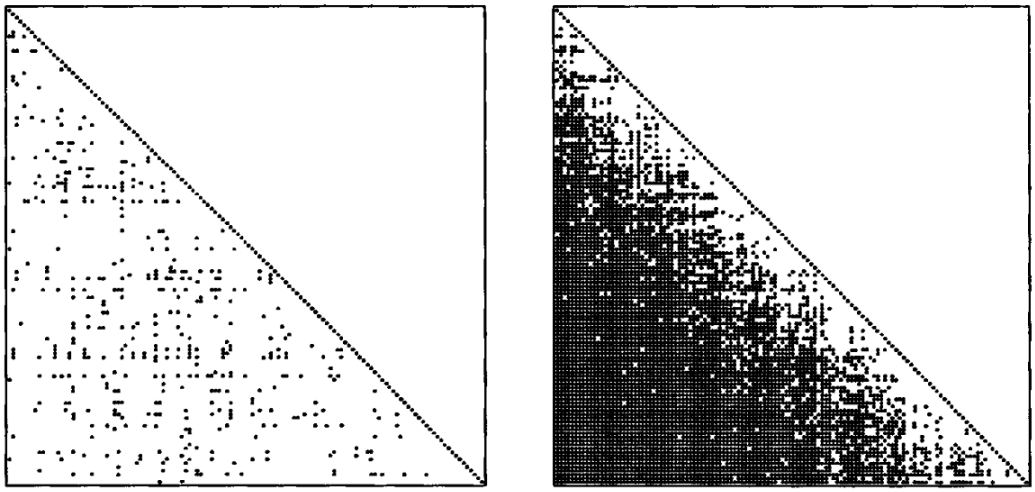



Figure 21.3: Let $A$ be a random $128 \times 128$ matrix with factorization $PA = LU$. On the left, $L^{-1}$ is shown: the dots represent entries with magnitude $\geqslant 1$. On the right, a similar picture for $\tilde{L}^{-1}$, where $\tilde{L}$ is the same as $L$ except that the signs of its subdiagonal 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 21.4 gives evidence of this. We use the Q portrait, defined by the Matlab commands:

```
1  [Q, R] = qr(A);
2  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.

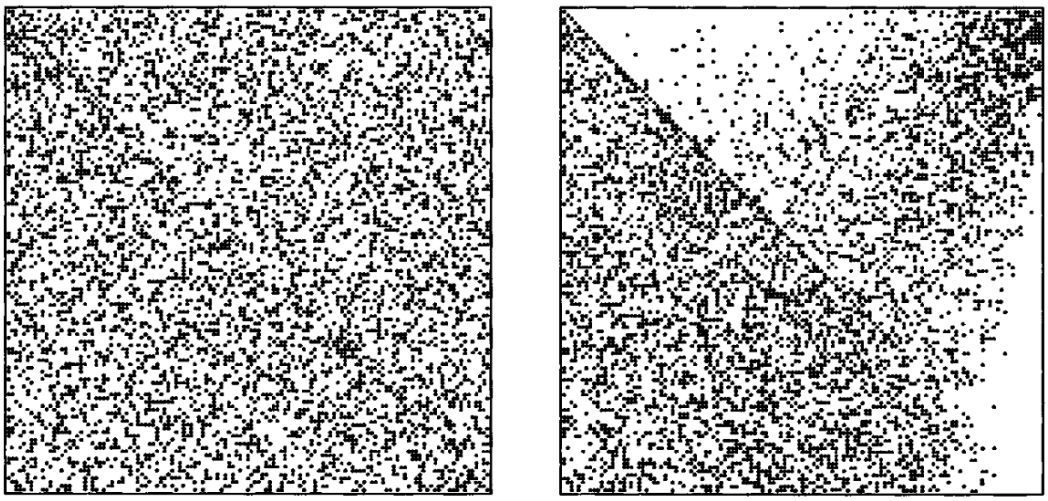



Figure 21.4: Q portraits of the same two matrices. On the left, the random matrix $A$ after permutation to the form $PA$, or equivalently, the factor $L$. On the right, the matrix $\tilde{L}$ with randomized signs. The column spaces of $\tilde{L}$ are skewed in a manner exponentially unlikely to arise in typical classes of random matrices.

**Note 21.5.**
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 22

# 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.

## 22.1 Symmetric Gaussian Elimination

First we recall the definition of Hermitian positive definite matrices.

**Definition 22.1** (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

$$A = \begin{bmatrix} 1 & w^* \\ w & K \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ w & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & K - ww^* \end{bmatrix} \begin{bmatrix} 1 & w^* \\ 0 & I \end{bmatrix}. \tag{22.1}$$

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.

## 22.2 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 (22.1) 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

$$A = \underbrace{R_1^* R_2^* \cdots R_m^*}_{R^*} \underbrace{R_m \cdots R_2 R_1}_{R}. \tag{22.2}$$

This equation has the form

$$A = R^* R, \quad r_{jj} > 0, \tag{22.3}$$

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 22.2** (Cholesky factorization).
Every Hermitian positive definite matrix $A \in \mathbb{C}^{m \times m}$ has a unique Cholesky factorization (22.3).

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.

## 22.3 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 22.1:** 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} / √R_{kk}
```

## 22.4 Operation Count

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

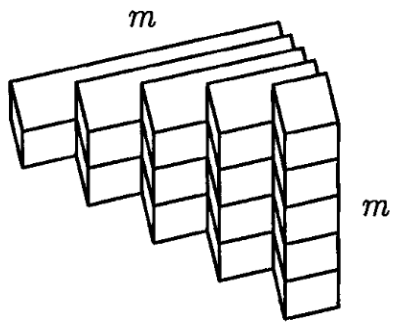
$$R_{j,j:m} = R_{j,j:m} - R_{k,j:m} R_{kj} / R_{kk}$$

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 22.3.**
Work for Cholesky factorization: $\sim \frac{1}{3} m^3$ flops.

## 22.5 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 22.4** (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 22.1 on a computer satisfying (12.3) and (12.4). 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

$$\tilde{R}^* \tilde{R} = A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = O\left(\epsilon_{\text{machine}}\right) \tag{22.4}$$

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

**Note 22.5.**
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 $\|\tilde{R} - R\|/\|R\| = O\left(\kappa(A)\epsilon_{\text{machine}}\right)$. (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 (22.4). Thus the errors introduced in $\tilde{R}$ by rounding are large but "diabolically correlated," just as we saw in Chapter 16 for QR factorization.

## 22.6 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 22.1 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 22.6.**
The solution of hermitian positive definite systems $Ax = b$ via Cholesky factorization (algorithm 22.1) is backward stable, generating a computed solution $\tilde{x}$ that satisfies

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

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

# Part V

# Eigenvalues

# CHAPTER 23

# EIGENVALUE PROBLEMS

Eigenvalue problems are particularly interesting in scientific computing, because the best algorithms for finding eigenvalues are powerful, yet particularly far from obvious. Here, we review the mathematics of eigenvalues and eigenvectors.

## 23.1 Eigenvalues and Eigenvectors

**Definition 23.1** (Eigenvalues and eigenvectors).
Let $A \in \mathbb{C}^{m\times m}$ be a square matrix. A nonzero vector $x \in \mathbb{C}^m$ is an eigenvector of $A$ and $\lambda \in \mathbb{C}$ is its corresponding eigenvalue, if

$$Ax = \lambda x.$$

The set of all eigenvalues of a matrix $A$ is the spectrum of $A$ denoted by $\Lambda(A)$.

Broadly speaking, eigenvalues and eigenvectors are useful for two reasons.

- Algorithmically, eigenvalue analysis can simplify solutions of certain problems by reducing a coupled system to a collection of scalar problems.
- Physically, eigenvalue analysis can give insight into the behavior of evolving systems governed by linear equations.

The most familiar examples in this latter class are the study of resonance and of stability.

## 23.2 Eigenvalue Decomposition

**Definition 23.2** (Eigenvalue Decomposition).
An eigenvalue decomposition of a square matrix $A$ is factorization

$$A = X\Lambda X^{-1}.$$

Here $X$ is nonsingular and $\Lambda$ is diagonal.

This formula is equivalent to $AX = X\Lambda$. This makes it clear that if $X_j$ is the jth column of $X$ and $\lambda_j$ is the jth diagonal entry of $\Lambda$, then $Ax_j = \lambda_j x_j$.

## 23.3 Geometric Multiplicity

As stated above, the set of eigenvector corresponding to a single eigenvalue, together with the zero vector, forms a subspace of $\mathbb{C}^m$ known as an eigenspace. If $\lambda$ is an eigenvalue of $A$, let us denote the

corresponding eigenspace by $E_\lambda$. An eigenspace $E_\lambda$ is an example of in invariant subspace of $A$; that is, $AE_\lambda \subset E_\lambda$.

The dimension of $E_\lambda$ can be interpreted as the maximum number of linearly independent eigenvector that can be found, all with the same eigenvalue $\lambda$. This number is known as the **geometric multiplicity** of $\lambda$. The geometric multiplicity can also be described as the dimension of the nullspace $A - \lambda I$, since that nullspace is again $E_\lambda$.

## 23.4 Characteristic Polynomial

**Definition 23.3 (Characteristic Polynomial).**
The characteristic polynomial of $A \in \mathbb{C}^{m\times m}$, denoted by $p_A$ is the degree $m$ polynomial defined by

$$p_A(z) = \det(zI - A).$$

Note that $p$ is monic, due to the placement of the minus sign.

**Theorem 23.4.**
$\lambda$ is an eigenvalue of $A$ if and only if $p_A(\lambda) = 0$.

Thm 23.4 has an important consequence. Even if a matrix is real, some of its eigenvalues may be complex. Physically, this is related to the phenomenon that real dynamical systems can have motions that oscillate as well as grow or decay.

## 23.5 Algebraic Multiplicity

By the fundamental theorem of algebra, we can write $p_A$ in the form

$$p_A(z) = (z - \lambda_1)(z - \lambda_2)\cdots(z - \lambda_m) \tag{23.1}$$

for some numbers $\lambda_j \in \mathbb{C}$. By Theorem 23.4, each $\lambda_j$ is an eigenvalue of $A$, and all eigenvalues of $A$ appear somewhere in this list. In general, an eigenvalue might appear more that once. We define the **algebraic multiplicity** of an eigenvalue $\lambda$ of $A$ to be its multiplicity as a root $p_A$. An eigenvalue is **simple** if its algebraic multiplicity is 1.

**Theorem 23.5.**
If $A \in \mathbb{C}^{m\times m}$, then $A$ has $m$ eigenvalues, counted with algebraic multiplicity. In particular, if the roots of $p_A$ are simple, then $A$ has $m$ distinct eigenvalues.

## 23.6 Similarity Transformation

**Definition 23.6 (Similarity transformation).**
If $X \in \mathbb{C}^{m\times m}$ is nonsingular, then the map $A \mapsto X^{-1}AX$ is called a similarity transformation of $A$. We say two matrices $A$ and $B$ are similar if there is a similarity transformation relating one the other.

**Theorem 23.7.**
If $A$ and $B$ are similar, they have the same characteristic polynomial, eigenvalues and algebraic and geometric multiplicities.

We can now relate geometric multiplicity to algebraic multiplicity.

**Theorem 23.8.**
The algebraic multiplicity of an eigenvalue $\lambda$ is at least as great as its geoemtric multiplicity.

*Proof sketch.*
Form a unitary matrix $V$ with the first several columns as the orthonormal basis for the specific eigenvalues. □

## 23.7 Defective Eigenvalues and Matrices

**Example 23.9.**
Consider the matrices

$$A = \begin{pmatrix} 2 & & \\ & 2 & \\ & & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 1 & \\ & 2 & 1 \\ & & 2 \end{pmatrix}.$$

Both $A$ and $B$ have characteristic polynomial $(z-2)^3$, so there is a single eigenvalue $\lambda = 2$ of algebraic multiplicity 3. In the case of $A$, we can find 3 eigenvectors for 2, while there is only one for $B$. So the geometric multiplicity of 2 for $A$ is 3, while for $B$ is 1.

An eigenvalue whose algebraic multiplicity exceeds its geometric multiplicity is a **defective eigenvalue**. A matrix that one or more defective eigenvalues is a **defective matrix**. Any diagonal matrix is non defective.

## 23.8 Diagonalizability

**Theorem 23.10.**
An $m \times m$ matrix $A$ is nondefective if and only if it has an eigenvalue decomposition $A = X\Lambda X^{-1}$.

In view of this result, another term for nondefective is **diagonalizable**.

## 23.9 Determinant and Trace

**Definition 23.11 (trace).**
The trace of $A \in \mathbb{C}^{m\times m}$ is the sum of its diagonal elements:

$$\operatorname{tr}(A) = \sum_{j=1}^{m} a_{jj}.$$

**Theorem 23.12.**
Given the square matrix $A \in \mathbb{C}^{m\times m}$, we have

$$\det(A) = \prod_{j=1}^{m} \lambda_j, \quad \operatorname{tr}(A) = \sum_{j=1}^{m} \lambda_j. \tag{23.2}$$

*Proof sketch.*
As for the trace equation, use the characteristic polynomial to prove it. □

## 23.10 Unitary Diagonalization

**Definition 23.13** (Unitarily diagonalizable).
Given $A \in \mathbb{C}^{m\times m}$, $A$ is unitarily diagonalizable if $\exists Q$ s.t.

$$A = Q\Lambda Q^*.$$

Note that this factorization is both an eigenvalue decomposition and a singular value decomposition, aside from the matter of the signs (possibly complex) of entries of $\Lambda$.

**Theorem 23.14.**
A hermitian matrix is unitarily diagonalizable, and its eigenvalues are real.

**Theorem 23.15.**
A matrix is unitarily diagonalizable if and only if it's normal i.e., $A^*A = AA^*$.

*Proof sketch.*
Use Schur decomposition to show $A = QTQ^*$. $T$ should be also normal and a normal upper triangular matrix is diagonal. □

## 23.11 Schur Factorization

**Definition 23.16** (Schur factorization).
A Schur factorization of a matrix $A$ is

$$A = QTQ^*,$$

where $Q$ is unitary and $T$ is upper-triangular.

**Theorem 23.17.**
Every square matrix $A$ has a Schur factorization.

*Proof sketch.*
Use induction on the first column of $A$ and extend it to a full unitary matrix. □

## 23.12 Eigenvalue-Revealing factorizations

In the preceding pages we have described three examples of eigenvalue-revealing factorizations, factorizations of a matrix that reduce it to a form in which the eigenvalues are explicitly displayed. we can summarize these as follows.

- A diagonalization $A = X\Lambda X^{-1}$ exists if and only if $A$ is nondefective.
- A unitary diagonalization $A = Q\Lambda Q^*$ exists if and only if $A$ is normal.
- A unitary triangularization $A = QTQ^*$ always exists.

To compute eigenvalues, we shall construct one of these factorizations. In general, this will be Schur factorization, since this applies without restriction to all matrices. Moreover, since unitary transformations are involved, the algorithms that result tend to be numerically stable. If $A$ is normal, then the Schur form comes out diagonal, and in particular, if $A$ is hermitian, then we can take advantage of this symmetry throughout the computation and reduce $A$ to diagonal form with half as much work or less than is required for general $A$.

# CHAPTER 24

# OVERVIEW OF EIGENVALUE ALGORITHMS

This and the next five chapters describe some of the classical "direct" algorithms for computing eigenvalues and eigenvectors, as well as a few modern variants. Most of these algorithms proceed in two phases: first, a preliminary reduction from full to structured form; them , an iterative process for the final convergence. This lecture outlines the two-phase approach and explains why it is advantageous.

## 24.1 Shortcomings of Obvious Algorithms

Perhaps the first method one might think of would be to compute the coefficients of the characteristic polynomial and use a rootfinder to extract its roots. UNfortunately, as mentioned in Chapter 14, this strategy is a bad one, because polynomial rootfinding is an ill-conditioned problem in general, even when the underlying eigenvalue problem is well-conditioned. (In fact, polynomial rootfinding is by no means a mainstream topic in scientific computing – precisely because it is so rarely the best way to solve applied problems.)

Another idea would be to take advantage of the fact that the sequence

$$\frac{x}{\|x\|}, \frac{Ax}{\|Ax\|}, \frac{A^2x}{\|A^2x\|}, \frac{A^3x}{\|A^3x\|}, \dots$$

converges, under certain assumptions, to an eigenvector corresponding to the largest eigenvalue of $A$ in absolute value. THis method for finding an eigenvector is called **power iteration**. Unfortunately, although power iteration is famous, it is by no means an effective tool for general use. Except for special matrices, it is very slow.

Instead of ideas like these, the best general purpose eigenvalue algorithms are based on a different principle: the computation of an eigenvalue-revealing factorization of $A$, where the eigenvalues appear as entries of one of the factors. We saw three eigenvalue-revealing factorizations in the last lecture: diagonalization, unitary diagonalization, and unitary triangularization (Schur factorization). In practice, eigenvalues are usually computed by constructing one of these factorizations. Conceptually, what must be done to achieve this is to apply a sequence of transformations to $A$ to introduce zeros in the necessary places, just as in the algorithms we have considered in the preceding chapters. Thus we see that finding eigenvaleus ends up rather similar in flavor to solving systems of equations or least square problems. The algorithms of numerical linear algebra are mainly built upon one technique used over and over again: putting zeros into matrices.

## 24.2 A Fundamental Difficulty

Though the flavors are related, however, a new spice appears in the dish when it comes to computing eigenvalues. What is new is that it would appear that algebraic considerations must preclude the success of any algorithm of this kind.

To see the difficulty, note that just as eigenvalue problems can be reduced to polynomial rootfinding problems, conversely, any polynomial rootfinding problem can be stated as an eigenvalue problem. Suppose we have the monic polynomial

$$p(z) = z^m + a_{m-1}z^{m-1} + \cdots + a_1 z + a_0.$$

It's not hard to verify that $p(z)$ is equal the determinant of the $m \times m$ matrix

$$\begin{bmatrix} -z & & & & & -a_0 \\ 1 & -z & & & & -a_1 \\ & 1 & -z & & & -a_2 \\ & & 1 & \ddots & & \vdots \\ & & & \ddots & -z & -a_{m-2} \\ & & & & 1 & (-z - a_{m-1}) \end{bmatrix}.$$

This means that the roots of $p$ are equal to the eigenvalues of the matrix

$$A = \begin{bmatrix} 0 & & & & & -a_0 \\ 1 & 0 & & & & -a_1 \\ & 1 & 0 & & & -a_2 \\ & & 1 & \ddots & & \vdots \\ & & & \ddots & 0 & -a_{m-2} \\ & & & & 1 & -a_{m-1} \end{bmatrix}.$$

Here $A$ is called a **companion matrix** corresponding to $p$. Now the difficulty is apparent. It's well known that no formula exists for expressing the roots of an arbitrary polynomial, given its coefficients. Abel proved in 1824 that no analogue of the quadratic formula can exist for polynomials of degree 5 or more.

This does not mean that we cannot write a good eigenvalue solver. It does mean, however, that such a solver cannot be based on the same kind of techniques that we have used so far for solving linear systems. Methods like Householder reflections and Gaussian elimination would solve linear systems of equations exactly in a finite number of steps if they could be implemented in exact arithmetic. By contrast,

Any eigenvalue solver must be iterative.

The goal of an eigenvalue solver is to produce sequences of numbers that converge rapidly towards eigenvalues. In this respect eigenvalue computations are more representative of scientific computing than solutions of linear systems of equations.

## 24.3 Schur Factorization and Diagonalization

Most of the general purpose eigenvalue algorithms in use today proceed by computing the Schur factorization. We compute a Schur factorization $A = QTQ^*$ by transforming $A$ by a sequence of elementary unitary similarity transformations $X \mapsto Q_j^* X Q_j$, so that the product

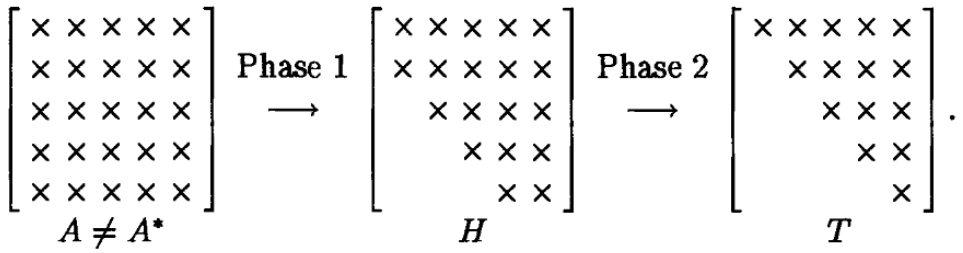
$$Q_j^* \cdots Q_2^* Q_1^* A Q_1 Q_2 \cdots Q_j \tag{24.1}$$

converges to an upper-triangular matrix $T$ as $j \to \infty$.

If $A$ is real but not symmetric, then in general it may have complex eigenvalues in conjugate pairs, in which case its Schur form will be complex. Thus an algorithm that computes the Schur factorization will have to capable of generating complex outputs from real inputs. This can certainly be done. Alternatively, it's possible to carry out the entire computation in real arithmetic if one computes what is know as a areal Schur factorization. Here, $T$ is permitted to have $2 \times 2$ blocks along the diagonal, one for each complex conjugate pair of eigenvalues. This option is important in practice, and is included in all the software libraries.

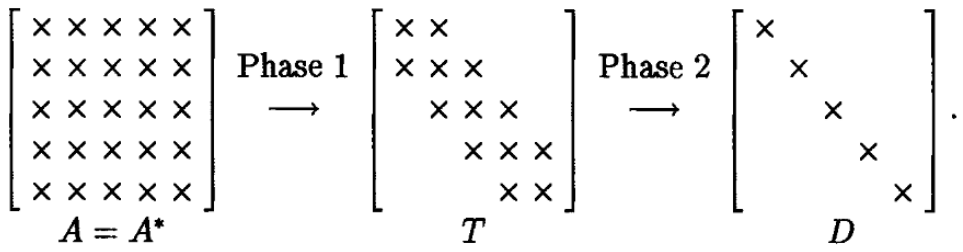
## 24.4 Two Phases of Eigenvalue Computations

Whether or not $A$ is hermitian, the sequence (24.1) is usually split into two phases. In the first phase, a direct method is applied to produce an upper-Hessenberg matrix $H$, that is, a matrix with zeros below the first subdiagonal. In the second phase, an iteration is applied to generate a formally infinite sequence of Hessenberg matrices that converge to a triangular form. Schematically, the process looks like this:




The first phase, a direct reduction, requires $O(m^3)$ flops. The second, iterative phase never terminates in principle, and if left to run forever would require an infinite number of flops. However, in practice, convergence to machine precision is achieved in $O(m)$ iterations. Each iteration requires $O(m^2)$ flops, and thus the total work requirement of $O(m^3)$ flops. These figures explain the importance of Phase 1. Without that preliminary step, each iteration of Phase 2 would involve a full matrix, requiring $O(m^3)$ work, and this would bring the total to $O(m^4)$– or higher, since convergence might also sometimes require more than $O(m)$ iterations.

If $A$ is hermitian, the two-phase approach becomes even faster. The intermediate matrix is now a hermitian Hessenberg matrix, that is, **tridiagonal**. The final result is a hermitian triangular matrix, that is, diagonal, as mentioned above. Schematically:




In this hermitian case, we shall see that if only eigenvalues are required, then each step of Phase 2 can be carried out with only $O(m)$ flops. Bringing the total work estimate for Phase 2 to $O(m^2)$ flops. Thus, for hermitian eigenvalue problems, we are in the paradoxical situation that the "infinite" part of the algorithm is in practice not merely as fast as the "finite" part, but an order of magnitude faster.

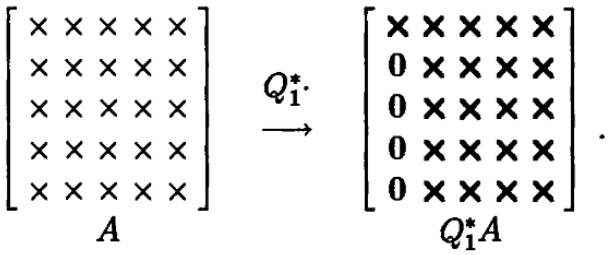
# CHAPTER 25

# REDUCTION TO HESSENBERG OR TRIDIAGONAL FORM

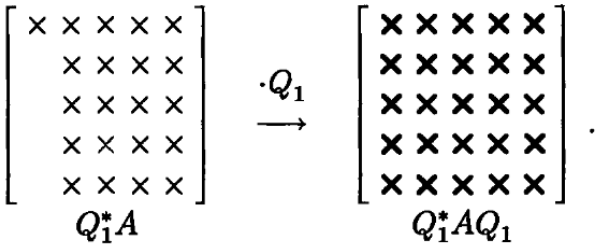
We now describe the first of the two computational phases outlined in the previous chapter: reduction of a full matrix to Hessenberg form by a sequence of unitary similarity transformations. If the origional matrix is hermitian, the result is tridiagonal.

## 25.1 A Bad Idea

To compute the Schur factorization $A = QTQ^*$, a natural first idea might be to attempt direct triangularization by Householder reflectors. We consider the following case. As usual, entries that are changed at each step are written in boldface:




Unfortunately, to complete the similarity transformation, we must also multiply by $Q_1$ on the right of $A$:




This has the effect of replacing each column of the matrix by a linear combination of all the columns. The result is that the zeros that were previously introduced are destroyed; we are no better off than when we started.

Of course, this is doomed to fail, since we know no finite process can reveal the eigenvalues of $A$ exactly. Curiously, this too-simple strategy, which appears futile as we have discussed it, does have the effect, typically, of reducing the size of the entries below the diagonal, even if it does not make them zero.

## 25.2 A Good Idea

The right strategy for introducing zeros in Phase 1 is to be less ambitious and operate on fewer entries of the matrix. We shall only conquer territory we are sure we can defend.

At the first step, we select a Householder reflector $Q_1^*$ that leaves the first row unchanged. When it is multiplied on the left of $A$, it forms linear combinations of only rows $2, \ldots, m$ to introduce zeros into row $3, \ldots, m$ of the first column. Then, when $Q_1$ is multiplied on the right of $Q_1^* A$, it leaves the first column unchanged.

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}} \xrightarrow{Q_1^* \cdot} \underset{Q_1^* A}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}} \xrightarrow{\cdot Q_1} \underset{Q_1^* A Q_1}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix}}.$$

This idea is repeated to introduce zeros into subsequent columns. Then we can also find a second $Q_2$ to change the second column:

$$\underset{Q_1^* A Q_1}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & \times & \times & \times & \times \end{bmatrix}} \xrightarrow{Q_2^* \cdot} \underset{Q_2^* Q_1^* A Q_1}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & 0 & \times & \times & \times \\ & 0 & \times & \times & \times \end{bmatrix}} \xrightarrow{\cdot Q_2} \underset{Q_2^* Q_1^* A Q_1 Q_2}{\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \times \end{bmatrix}}.$$

After repeating this process $m-2$ times, we have a product in Hessenberg form, as desired:

$$\underbrace{Q_{m-2}^* \cdots Q_2^* Q_1^*}_{Q^*} A \underbrace{Q_1 Q_2 \cdots Q_{m-2}}_{Q} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix} = H.$$

The algorithm is formulated below.

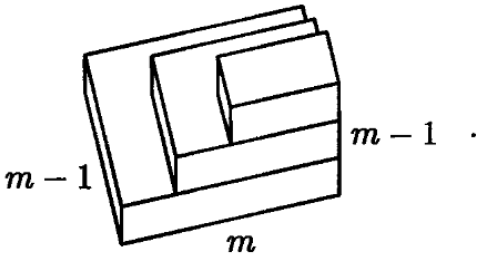
**Algorithm 25.1:** Householder Reduction to Hessenberg Form

```
for k = 1 to m − 2 do
    x = A_{k+1:m,k};
    v_k = sign(x_1)‖x‖_2 e_1 + x;
    v_k = v_k/‖v_k‖_2;
    A_{k+1:m,k:m} = A_{k+1:m,k:m} − 2v_k(v_k^* A_{k+1:m,k:m});
    A_{1:m,k+1:m} = A_{1:m,k+1:m} − 2(A_{1:m,k+1:m} v_k)v_k^*;
```

## 25.3 Operation Count

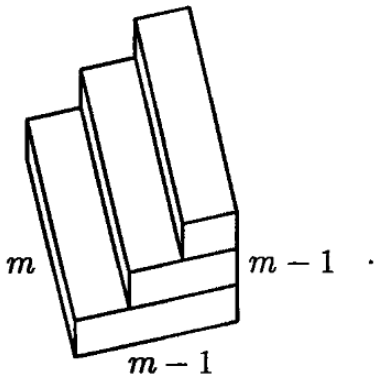
The number of operations required by Algo 25.1 can be counted with the same geometric reasoning we have used before. The rule of thumb is that unitary operations require four flops for reach element operated upon.

The work is dominated by the two updates of submatrices of $A$. The first loop applies a Householder reflector on the left of the matrix. The $k$ th such reflector operates on the last $m-k$ rows. Note that arithmetic has to be perfomed only on the last $m-k+1$ entries of each row. The picture is as follows:




As $m \to \infty$, the volume converges to $\frac{1}{3}m^3$. At four flops per element, the amount of work in this loop is $\sim \frac{4}{3}m^3$ flops.

The second inner loop applies a Householder reflector on the right of the matrix. At the $k$th step, the reflector operates by forming linear combination of the last $m-k$ columns. This loop involves more work than the first one because there are no zeros that can be ignored. Arithmetic must be performed on all of the $m$ entries of each of the columns operated upon. A total of $m(m-k)$ entries for a single value of $k$. The picture looks like this:




The volume converges as $m \to \infty$ to $\frac{1}{2}m^3$, so, at four flops per element, this second loop requires $\sim 2m^3$ flops.

**Theorem 25.1.**
The work for Hessenberg reduction: $\sim \frac{10}{3}m^3$ flops.

## 25.4 The Hermitian Case

If $A$ is hermitian, the algorithm will reduced $A$ to tridiagonal form. Since zeros are now introduced in rows as well as columns, additional arithmetic can be avoided by ignoring these additional zeros. Hence, the total amount of arithmetic is reduced to $\frac{8}{3}m^3$.

However, if we take advantage of both sparsity and symmetry, the toal work can be further reduced by a factor of two.

**Theorem 25.2.**
Work for tridiagonal reduction: $\sim \frac{4}{3}m^3$ flops.

## 25.5 Stability

Like the Householder algorithm for QR factorization, the algorithm just described is backward stable. Let $\tilde{H}$ be the actual Hessenberg matrix computed in floating arithmetic, and let $\tilde{Q}$, as before, be the exactly unitary matrix corresponding to the reflection vectors $\tilde{v}_k$ computed in floating point arithmetic. We have

**Theorem 25.3** (Backstab of Hessenberg reduction).
Let the Hessenberg reduction $A = QHQ^*$ of a matrix $A \in \mathbb{C}^{m\times m}$ be computed by Algo 25.1 on a computer satifying the axioms (12.3) and (12.4), and let the computed factors $\tilde{Q}$ and $\tilde{H}$ be defiend as indicated above. Then we have

$$\tilde{Q}\tilde{H}\tilde{Q}^* = A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}) \tag{25.1}$$

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

# CHAPTER 26

# RAYLEIGH QUOTIENT, INVERSE ITERATION

In this chapter, we present some classical eigenvalue algorithms. Individually, these tools are useful in certain circumstances–especially inverse iteration, which is the standard method for determining an eigenvector when the corresponding eigenvalue is known. Combined, they are ingredients of the celebrated QR algorithm.

## 26.1 Restriction ot Real Symmetric Matrices

Throughout numerical linear algebra, most algorithmic ideas are applicable either to general matrices or, with certain simplifications, to hermitian matrices. For the topics discussed in this and the next three chapters, this continues to be at least partly true, but some of the differences between the general and the hermitian cases are rather sizable. Therefore, in these four chapter, we simplify matters by considering only matrices that are real and symmetric. We also assume throughout that $\|\cdot\| = \|\cdot\|_2$.

Thus, for these four chapters: $A = A^\top \in \mathbb{R}^{m\times m}, x \in \mathbb{R}^m, x^* = x^\top, \|x\| = \sqrt{x^\top x}$. In particular, this means that $A$ has real eigenvalues and a complete set of orthogonal eigenvectors. We use the following notation:

- Real eigenvalues: $\lambda_1, \ldots, \lambda_m$,
- Orthonormal eigenvectors: $q_1, \ldots, q_m$.

Most of the ideas to be described in the next few lectures pertain to Phase 2 of the two phases described in Chapter 24. This means $A$ will be tridiagonal. This tridiagonal structure is occasionally of mathematical importance, for example in choosing shifts for the QR algorithm, and it's always of algorithmic importance, reducing many steps from $O(m^3)$ to $O(m)$ flops.

## 26.2 Rayleigh Quotient

**Definition 26.1** (Rayleigh quotient).
The Rayleigh quotient of a vector $x \in \mathbb{R}^m$ is the scalar

$$r(x) = \frac{x^\top A x}{x^\top x}.$$

Notice that if $x$ is an eigenvector, then $r(x) = \lambda$ is the corresponding eigenvalue. Besides, given $x$,if we want to find a scalar $\alpha$ that "acts most like an eigenvalue" in the sense of minimizing $\|Ax - \alpha x\|_2$, the answer should be $\alpha = r(x)$. Thus $r(x)$ is a natural eigenvalue estimate to consider if $x$ is close to, but not necessarily equal to, an eigenvector.

Furthermore, we can compute the derivative of $r$, we have

$$\frac{\partial r(x)}{\partial x_j} = \frac{\frac{\partial}{\partial x_j}(x^\top Ax)}{x^\top x} - \frac{(x^\top Ax)\frac{\partial}{\partial x_j}(x^\top x)}{(x^\top x)^2} = \frac{2(Ax)_j}{x^\top x} - \frac{(x^\top Ax)2x_j}{(x^\top x)^2} = \frac{2}{x^\top x}(Ax - r(x)x)_j.$$

Combine this, we get

$$\nabla r(x) = \frac{2}{x^\top x}(Ax - r(x)x).$$

Geometrically speaking, the eigenvectors of $A$ are the stationary points of the function $r(x)$. Besides, since $r(x)$ is independent of the scale of $x$, these stationary points lie along lines through the origin in $\mathbb{R}^m$. We can restrict our attention to the unit sphere $\|x\| = 1$.

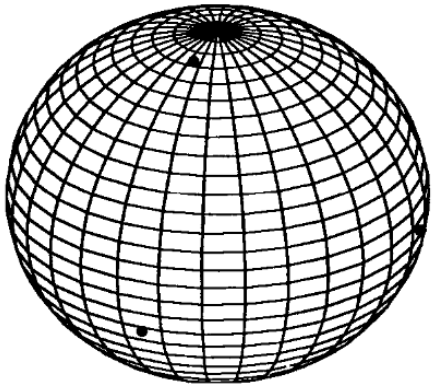


Figure 26.1: The Rayleigh quotient $r(x)$ is a continuous function on the unit sphere $\|x\| = 1$ in $\mathbb{R}^m$, and the stationary points of $r(x)$ are the normalized eigenvectors of $A$. In this example with $m = 3$, there are three orthogonal stationary points.

Let $q_J$ be one of the eigenvectors of $A$. From the fact that $\nabla r(q_J) = 0$, together with the smoothness of the function $r(x)$, we derive an important consequence:

$$r(x) - r(q_J) = O(\|x - q_J\|^2), \quad \text{as } x \to q_J. \tag{26.1}$$

Thus the Rayleigh quotient is a **quadratically accurate estimate of an eigenvalue**. Herein lies its power.

## 26.3 Power Iteration

Now we switch tacks. Suppose $v^{(0)}$ is a vector with $\|v^{(0)}\| = 1$. The following process, **power iteration**, was cited as a not especially good idea at the beginning of Chapter 24. It may be expected to produce a sequence $v^{s(i)}$ that converges to an eigenvector corresponding to the largest eigenvalue of $A$¿

**Algorithm 26.1:** Power Iteration

```
v^(0) = some vector with ||v^(0)|| = 1;
for k = 1, 2, ... do
    w = A v^(k-1);
    v^(k) = w / ||w||;
    λ^(k) = (v^(k))^T A v^(k);
```

In this and the algorithms to follow, we give no attention to termination conditions, describing the loop only by the suggestive expression. Of course, in practice, termination conditions are very important, and this is one of this points where top-quality software such as can be found in LAPACK or Matlab is likely to be superior to a program an individual might write.

We can analyze power iteration easily. Write $v^{(0)}$ as a linear combination of these orthonormal eigenvectors $q_i$:

$$v^{(0)} = a_1 q_1 + a_2 q_2 + \cdots + a_m q_m.$$

Hence, $v^{(k)}$ is:

$$\begin{aligned} v^{(k)} &= c_k A^k v^{(0)} \\ &= c_k(a_1 \lambda_1^k q_1 + a_2 \lambda_2^k q_2 + \cdots + a_m \lambda_m^k q_m) \\ &= c_k \lambda_1^k (a_1 q_1 + a_2 (\lambda_2/\lambda_1)^k q_2 + \cdots + a_m (\lambda_m/\lambda_1)^k q_m), \end{aligned}$$

for some constants $c_k$.

**Theorem 26.2.**
Suppose $|\lambda_1| > |\lambda_2| \geqslant \cdots \geqslant |\lambda_m| \geqslant 0$ and $q_1^\top v^{(0)} \neq 0$. Then the iterates of Algo 26.1 satisfy

$$\|v^{(k)} - (\pm q_1)\| = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right), \quad |\lambda^{(k)} - \lambda_1| = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}\right) \tag{26.2}$$

as $k \to \infty$. The $\pm$ sign, means at each step $k$, one or the other choice of sign is to be taken, and then the indicated bound holds.

On its own, power iteration is of limited use, for several reasons.

- First, it can find only the eigenvector corresponding to the largest eigenvalue.
- Second, the convergence is linear, reducing the error only by a constant factor $\approx |\lambda_2/\lambda_1|$ at each iteration.
- Finally, the quality of this factor depends on having a largest eigenvalue that is significantly larger than the others.

Fortunately, there is a way to amplify the differences between eigenvalues.

## 26.4 Inverse Iteration

For any $\mu \in \mathbb{R}$ s.t. $\mu \notin \operatorname{Spec} A$, we must have $E_{(\lambda - mu)^{-1}}((A - \mu I)^{-1}) = E_\lambda(A)$ and

$$\operatorname{Spec}(A - \mu I)^{-1} = \{(\lambda_j - \mu)^{-1} | \lambda_j \in \operatorname{Spec} A\}.$$

Suppose $\mu$ is close to an eigenvalue $\lambda_J$ of $A$. Then $(\lambda_J - \mu)^{-1}$ may be much larger than $(\lambda_j - mu)^{-1}$ for all $j \neq J$. Thus, it will converge rapidly to $q_J$. This idea is called **inverse iteration**.

**Algorithm 26.2:** Inverse Iteration

```
1 v^(0) = some vector with ||v^(0)|| = 1;
2 for k = 1, 2, ... do
3 |   Solve (A − μI)w = v^(k−1) for w;
4 |   v^(k) = w/||w||;
5 |   λ^(k) = (v^(k))^⊤ A v^(k);
```

What if $\mu$ is an eigenvalue of $A$, so that $A - \mu I$ is singular? What if it's nearly an eigenvalue, so that $A - \mu I$ is so ill-conditioned that an accurate solution of $(A - \mu I)w = v^{(k-1)}$ cannot be expected? These cannot cause real problems.

Like power iteration, inverse iteration exhibits only linear convergence. Unlike power iteration, however, we can choose the eigenvector that will be found by supplying an estimate $\mu$ of the corresponding eigenvalue. Furthermore, the rate of linear convergence can be controlled, for it depends on the quality of $\mu$. We have the following theorem:

**Theorem 26.3.**
Suppose $\lambda_J$ is the closest eigenvalue to $\mu$ and $\lambda_K$ is the second closest, that is, $|\mu-\lambda_J| < |\mu-\lambda_K| \leqslant |\mu-\lambda_j|$ for each $j \neq J$. Furthermore, suppose $q_J^\top v^{(0)} \neq 0$. Then the iterates of Algo 26.2 satisfy

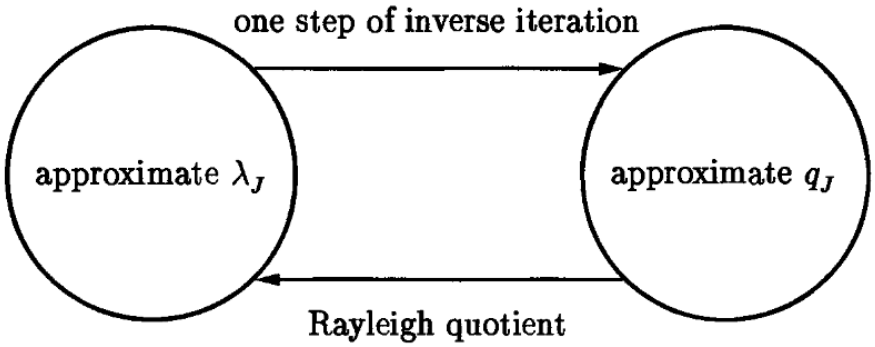
$$\|v^{(k)} - (\pm q_J)\| = O\left(\left|\frac{\mu-\lambda_J}{\mu-\lambda_K}\right|^k\right), \quad |\lambda^{(k)} - \lambda_J| = O\left(\left|\frac{\mu-\lambda_J}{\mu-\lambda_K}\right|^{2k}\right)$$

as $k \to \infty$, where the $\pm$ sign has the same meaning as in Thm 26.2.

Inverse iteration is one of the most valuable tools of NLA, for it's the standard method of calculating one or more eigenvectors of a matrix if the eigenvalues are already known.

## 26.5 Rayleigh Quotient Iteration

In this chapter, we presented two methods: one for estimating eigenvector and one for obtaining an eigenvector estimate from an eigenvalue estimate. We can combine them.




This algorithm is called **Rayleigh quotient iteration**.

**Algorithm 26.3:** Rayleigh Quotient Iteration

$v^{(0)}$ = some vector with $\|v^{(0)}\| = 1$;
$\lambda^{(0)} = (v^{(0)})^\top A v^{(0)}$;
**for** $k = 1, 2, \ldots$ **do**
  Solve $(A - \lambda^{(k-1)} I)w = v^{(k-1)}$ for $w$;
  $v^{(k)} = \frac{w}{\|w\|}$;
  $\lambda^{(k)} = (v^{(k)})^\top A v^{(k)}$;

The convergence of this algorithm is spectacular: each iteration triples the number of digits of accuracy.

**Theorem 26.4.**
Raleigh quotient iteration converges to an eigenvalue/eigenvector pair for all except a set of measure zero of starting vectors $v^{(0)}$. When it converges, the convergence is ultimately cubic in the sense that if $\lambda_J$ is an eigenvalue of $A$ and $v^{(0)}$ is sufficiently close to the eigenvector $q_J$, then

$$\|v^{(k+1)} - (\pm q_J)\| = O(\|v^{(k)} - (\pm q_J)\|^3) \tag{26.3}$$

and

$$|\lambda^{(k+1)} - \lambda_J| = O(|\lambda^{(k)} - \lambda_J|^3) \tag{26.4}$$

as $k \to \infty$. The $\pm$ signs are not necessarily the same on the two sides of (26.3).

*Proof sketch.*
Firstly, we have (26.1),

$$|\lambda^{(k)} - \lambda_J| = C\|v^{(k)} - q_J\|^2.$$

Then following the proof of Thm 26.3,

$$\|v^{(k+1)} - q_J\| = O(|\lambda^{(k)} - \lambda_J|\|v^{(k)} - q_J\|) = C\|v^{(k)} - q_J\|^3.$$

Similar rate can be achieved for $|\lambda^{(k)} - \lambda_J|$. □

**Example 26.5.**
Cubic convergence is so fast. Consider the symmetric matrix

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 4 \end{bmatrix},$$

and let $v^{(0)} = (1, 1, 1)^\top/\sqrt{3}$ be the initial eigenvector estimate. When Rayleigh quotient iteration is applied to $A$, the following values $\lambda^{(k)}$ are computed by the first three iterations:

$$\lambda^{(0)} = 5, \quad \lambda^{(1)} = 5.2131\ldots, \quad \lambda^{(2)} = 5.214319743184\ldots$$

The actual value of the eigenvalue corresponding to the eigenvector closet to $v^{(0)}$ is $\lambda = 5.414319743377$. After only three iterations, Rayleigh quotient iteration has produced a result accurate to ten digits.

## 26.6 Operation Counts

First, suppose $A \in \mathbb{R}^{m\times m}$ is a full matrix. Then each step of power iteration involves a matrix-vector multiplication, requiring $O(m^2)$ flops. Each step of inverse iteration involves the solution of a linear system, which might seem to require $O(m^3)$ flops, but this figure reduces to $O(m^2)$ if the matrix is processed in advance by LU or QR factorization or another method. In the case of Rayleigh quotient iteration, the matrix to be inverted changes at each step, and beating $O(m^3)$ flops per step is not so straightforward.

These figures improve greatly if $A$ is tridiagonal. Now, all three iterations require just $O(m)$ flops per step. For the analogous iterations involving nonsymmetric matrices, incidentally, we must deal with Hessenberg instead of tridiagonal structure, and this figure increases to $O(m^2)$.

# CHAPTER 27

# QR ALGORITHM WITHOUT SHIFTS

The QR algorithm, dating to the early 1960s, is one of the jewels of numerical analysis. Here we show that in its simplest form, this algorithm can be viewed as a stable procedure for computing QR factorizations of the matrix powers $A, A^2, A^3, \ldots$.

## 27.1 The QR Algorithm

The most basic version of the QR algorithm seems impossibly simple.

**Algorithm 27.1:** "Pure" QR Algorithm

```
A^(0) = A;
for k = 1, 2, ... do
    Q^(k) R^(k) = A^(k-1) ;        // QR factorization of A^(k-1)
    A^(k) = R^(k) Q^(k) ;          // Recombine factors in reverse order
```

Under suitable assumptions, this single algorithm converges to a Schur form for the matrix $A$. Note that here we assume $A$ is real and symmetric, with real eigenvalues $\lambda_j$ and orthonormal eigenvectors $q_j$.

We should notice that $A^{(k)} = (Q^{(k)})^\top A^{(k-1)} Q^{(k)}$. This is the "bad idea" mentioned in Chapter 25. Although this transformation is a bad idea when trying to reduce $A$ to triangular form in a single step, it turns out to be quite powerful as the basis of an iteration.

Like the Rayleigh quotient iteration, the QR algorithm for real symmetric matrices converges cubically. To achieve this performance, however, the algorithm as presented above must be modified by the introduction of shifts at each step. The use of shifts is one of three modifications of Algo 27.1 that are required to bring it closer to a practical algorithm:

1. Before starting the iteration, $A$ is reduced to tridiagonal form, as discussed in Chapter 25.
2. Instead of $A^{(k)}$, a shifted matrix $A^{(k)} - \mu^{(k)} I$ is factored at each step, where $\mu^{(k)}$ is some eigenvalue estimate.
3. Whenever possible, and in particular whenever an eigenvalue is found, the problem is "deflated" by breaking $A^{(k)}$ into submatrices.

A QR algorithm incorporating these modifications has the following outline.

**Algorithm 27.2:** "Practical" QR Algorithm

$(Q^{(0)})^* A^{(0)} Q^{(0)} = A$ ; // $A^{(0)}$ is a tridiagonalization of $A$

**for** $k = 1, 2, \ldots$ **do**

  Pick a shift $\mu^{(k)}$ ; // e.g., choose $\mu^{(k)} = A_{mm}^{(k-1)}$

  $Q^{(k)} R^{(k)} = A^{(k-1)} - \mu^{(k)} I$ ; // QR factorization of $A^{(k-1)} - \mu^{(k)} I$

  $A^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I$;

  If any off-diagonal element $A_{j,j+1}^{(k)}$ is sufficient close to zero, set $A_{j,j+1} = A_{j+1,j} = 0$ to obtain

$$\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} = A^{(k)}$$

  and now apply the QR algorithm to $A_1$ and $A_2$.

This algorithm, the QR algorithm with well-chosen shifts, has been the standard method for computing all the eigenvalues of a matrix since the early 1960s. Only in the 1990s has a competitor emerged, the divide-and -conquer algorithm described in Chapter 29.

Tridiagonalization was discussed in Chapter 25, shifts are discussed in the next chapter, and deflation is not discussed further in this note. For now, let us confine our attention to the "pure" QR algorithm and explain how it finds eigenvalues.

## 27.2 Unnormalized Simultaneous Iteration

Our approach will be to relate the QR algorithm to another method called **simultaneous iteration**, whose behavior is more obvious.

The idea of simultaneous iteration is to apply the power iteration to several vectors at once. (An equivalent term is **block power iteration**.) Suppose we start with a set of $n$ linearly independent vectors $v_1^{(0)}, \ldots, v_n^{(0)}$. It seems plausible that as $A^k v_1^{(0)}$ converges as $k \to \infty$ (under suitable assumptions) to the eigenvector corresponding the largest eigenvalue of $A$ in absolute value, the space $\langle A^k v_1^{(0)}, \ldots, A^k v_n^{(0)} \rangle$ should converge (again under suitable assumptions) to the space $\langle q_1, \ldots, q_n \rangle$ spanned by the eigenvectors $q_1, \ldots, q_n$ of $A$ corresponding to the $n$ largest eigenvalues in absolute value.

In matrix notation, we might proceed like this. Define $V^{(0)}$ to be the $m \times n$ initial matrix

$$V^{(0)} = \left[ \begin{array}{c|c|c} & & \\ v_1^{(0)} & \cdots & v_n^{(0)} \\ & & \end{array} \right], \tag{27.1}$$

and define $V^{(k)}$ to be the result after $k$ applications of $A$:

$$V^{(k)} = A^k V^{(0)} = \left[ \begin{array}{c|c|c} & & \\ v_1^{(k)} & \cdots & v_n^{(k)} \\ & & \end{array} \right]. \tag{27.2}$$

Since our interest is in the column space of $V^{(k)}$, let us extract a well-behaved basis for this space by computing a reduced QR factorization of $V^{(k)}$:

$$\hat{Q}^{(k)} \hat{R}^{(k)} = V^{(k)}. \tag{27.3}$$

Here $\hat{Q}^{(k)} \in \mathbb{C}^{m \times n}, \hat{R}^{(k)} \in \mathbb{C}^{n \times n}$. It seems plausible that as $k \to \infty$, under suitable assumptions, the successive columns of $\hat{Q}^{(k)}$ should converge to the eigenvectors $\pm q_1, \pm q_2, \ldots, \pm q_n$.

If we expand $v_j^{(0)}$ and $v_j^{(k)}$ in the eigenvectors of $A$, we have

$$\begin{aligned} v_j^{(0)} &= a_{1j} q_1 + \cdots + a_{mj} q_m, \\ v_j^{(k)} &= \lambda_1^k a_{1j} q_1 + \cdots + \lambda_m^k a_{mj} q_m. \end{aligned}$$

As in the last section, simple convergence results will hold provided that two conditions are satisfied.

1. The leading $n+1$ eigenvalues are distinct in absolute value: $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > |\lambda_{n+1}| \geqslant |\lambda_{n+2}| \geqslant \cdots \geqslant |\lambda_m|$.
2. The collection of expansion coefficients $a_{ij}$ is in an appropriate sense nonsingular. Define $\hat{Q}$ to be the $m \times n$ matrix whose columns are the eigenvectors $q_1, q_2, \ldots, q_n$. We assume the following: All the leading principal minors of $\hat{Q}^\top V^{(0)}$ are nonsingular.

**Theorem 27.1.**
Suppose that the iteration (27.1)-(27.3) is carried out and that two assumptions are satisfied. Then as $k \to \infty$, the columns of the matrices $\hat{Q}^{(k)}$ converge linearly to the eigenvectors of $A$:

$$\|q_j^{(k)} - \pm q_j\| = O(C^k) \tag{27.4}$$

for each $j$ with $1 \leqslant j \leqslant n$, where $C = \max_{1 \leqslant k \leqslant n} \frac{|\lambda_{k+1}|}{|\lambda_k|} < 1$.

*Proof sketch.*
We extend $\hat{Q}$ to the full $m \times m$ orthogonal matrix $Q$ of $A$, and we have the eigenvalue decomposition $A = Q\Lambda Q^\top$. We also define $\hat{\Lambda}$ to be the leading $n \times n$ section of $\Lambda$. Then we have

$$V^{(k)} = A^k V^{(0)} = Q\Lambda^k Q^\top V^{(0)} = \hat{Q}\hat{\Lambda}^k \hat{Q}^\top V^{(0)} + O(|\lambda_{n+1}|^k) = (\hat{Q}\hat{\Lambda}^k + O(|\lambda_{n+1}|^k))\hat{Q}^\top V^{(0)}.$$

Then we can tell the column space of

$$\hat{Q}\hat{\Lambda}^k + O(|\lambda_{n+1}|^k)$$

will converge to that of $\hat{Q}$ linearly. Besides, due to all principals minors of $\hat{Q}^\top V^{(0)}$ are nonsingular, the first several columns will converge linearly to the space spanned by the corresponding columns of $\hat{Q}$. □

## 27.3 Simultaneous Iteration

As $k \to \infty$, the vectors $v_1^{(k)}, \ldots, v_n^{(k)}$ in the algorithm (27.1)-(27.3) all converge to multiples of the same dominant eigenvector $q_1$ of $A$. Thus although the space they span, $\langle v_1^{(k)}, \ldots, v_j^{(k)} \rangle$, converges to something useful, these vectors constitute a high ill-conditioned basis of that space.

The remedy is simple: we do orthonormalize at each step rather than once and for all by QR decomposition.

**Algorithm 27.3:** Simultaneous Iteration

Pick $\hat{Q}^{(0)} \in \mathbb{R}^{m \times n}$ with orthonormal columns. ;
**for** $k = 1, 2, \ldots$ **do**
  $Z = A\hat{Q}^{(k-1)}$;
  $\hat{Q}^{(k)}\hat{R}^{(k)} = Z$ ; `// reduced QR factorization of` $Z$

Since the columns spaces are the same for $Z$ and $\hat{Q}^{(k)}$, this new formulation of simultaneous iteration converges under the same condition as the old one.

**Theorem 27.2.**
Algo 27.3 generate the same matrices $\hat{Q}^{(k)}$ as the iteration (27.1) and (27.3) considered in Thm 27.1 and converges as described in that theorem.

## 27.4 Simultaneous Iteration ⟺ QR Algorithm

Now we can show QR algorithm is equivalent to the simultaneous iteration applied to a full set of $n = m$ initial vectors, namely, the identity matrix, $\hat{Q}^{(0)} = I$. Since the matrices $\hat{Q}^{(k)}$ are now square,

we are dealing with full QR factorizations and can drop the hats on $\hat{Q}^{(k)}$ and $\hat{R}^{(k)}$. In fact, we shall replace $\hat{R}^{(k)}$ by $R^{(k)}$ but $\hat{Q}^{(k)}$ by $\underline{Q}^{(k)}$ in order to distinguish the $Q$ matrices of simultaneous iteration from those of the QR algorithm.

Now we compare the two methods. Here are the three formulas that define simultaneous iteration with $\underline{Q}^{(0)} = I$, followed by a fourth formula that we shall take as a definition of an $\bar{m} \times m$ matrix $A^{(k)}$ :

Simultaneous Iteration:

$$Q^{(0)} = I, \tag{27.5}$$
$$Z = A\underline{Q}^{(k-1)}, \tag{27.6}$$
$$Z = \underline{Q}^{(k)} R^{(k)}, \tag{27.7}$$
$$A^{(k)} = (\underline{Q}^{(k)})^\top A \underline{Q}^{(k)}. \tag{27.8}$$

And here are the three formulas that define the pure QR algorithm, followed by a fourth formula that we shall take as a definition of an $m \times m$ matrix $\underline{Q}^{(k)}$:

Unshifted QR Algorithm:

$$A^{(0)} = A \tag{27.9}$$
$$A^{(k-1)} = Q^{(k)} R^{(k)} \tag{27.10}$$
$$A^{(k)} = R^{(k)} Q^{(k)} \tag{27.11}$$
$$\underline{Q}^{(k)} = Q^{(1)} Q^{(2)} \cdots Q^{(k)}. \tag{27.12}$$

Additionally, for both algorithms, let us define one further $m \times m$ matrix $\underline{R}^{(k)}$,

$$\underline{R}^{(k)} = R^{(k)} R^{(k-1)} \cdots R^{(1)}. \tag{27.13}$$

**Theorem 27.3.**
The processes (27.5)-(27.8) and (27.9)-(27.12) generate identical sequences of matrices $\underline{R}^{(k)}, \underline{Q}^{(k)}$ and $A^{(k)}$, namely, those defined by the QR factorization of the $k$th power of $A$,

$$A^k = \underline{Q}^{(k)} \underline{R}^{(k)}, \tag{27.14}$$

together with the projection

$$A^{(k)} = (\underline{Q}^{(k)})^\top A \underline{Q}^{(k)}. \tag{27.15}$$

## 27.5 Convergence of the QR Algorithm

All the pieces are in place. We can now say a great deal about the convergence of the unshifted QR algorithm.

For qualitative understanding: (27.14) and (27.15) are the key.

- The QR algorithm can find the eigenvectors because it construct orthonormal bases for successive powers $A^k$.
- It finds the eigenvalues since the diagonal elements of $A^{(k)}$ are Rayleigh quotients of $A$ corresponding to the columns of $\underline{Q}^{(k)}$.

We also have the following consequences of Thm 27.2.

**Theorem 27.4.**
Let the pure $QR$ algorithm (Algo 27.1) be applied to a real symmetric matrix $A$ whose eigenvalues satisfy $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_m|$ and whose corresponding eigenvector matrix $Q$ has all nonsingular leading principal minors. Then as $k \to \infty$, $A^{(k)}$ converges linearly with constant $\max_k |\lambda_{k+1}| / |\lambda_k|$ to $\operatorname{diag}(\lambda_1, \ldots, \lambda_m)$, and $\underline{Q}^{(k)}$ (with the signs of its columns adjusted as necessary) converges at

the same rate to $Q$.

# CHAPTER 28

# QR ALGORITHM WITH SHIFTS

What makes the QR iteration fly is the introduction of shifts $A \to A - \mu I$ at each step. Here we explain how this idea leads to cubic convergence, thanks to an implicit connection with Rayleigh quotient iteration.

## 28.1 Connection with Inverse Iteration

As we have seen, the "pure" algorithm (Algo 27.1) is equivalent to simultaneous iteration applied to the identity matrix, and in particular the fist column of the result evolves according to the power iteration applied to $e_1$. There is a dual to this observation. Algo 27.1 is also equivalent to **simultaneous inverse iteration** applied to a "flipped" identity matrix $P$, and in particular, the $m$th column of the result evolves according to inverse iteration applied to $e_m$.

Recall that we have $Q^{(k)}$ and

$$\underline{Q}^{(k)} = \prod_{j=1}^{k} Q^{(j)} = \left[\begin{array}{c|c|c|c} & & & \\ q_1^{(k)} & q_2^{(k)} & \cdots & q_m^{(k)} \\ & & & \end{array}\right].$$

Besides, we have $\underline{Q}^{(k)}$ is the orthogonal factor in a QR factorization,

$$A^k = \underline{Q}^{(k)} \underline{R}^{(k)}.$$

Now we can invert this formula. We have

$$A^{-k} = \left(\underline{R}^{(k)}\right)^{-1} \underline{Q}^{(k)\top} = \underline{Q}^{(k)} \left(\underline{R}^{(k)}\right)^{-\top}.$$

Let

$$P = \begin{bmatrix} & & 1 \\ & 1 & \\ \cdots & & \\ 1 & & \end{bmatrix}.$$

Since $P^2 = I$, we have

$$A^{-k}P = [\underline{Q}^{(k)}P][P(\underline{R}^{(k)})^{-\top}P].$$

Note that $\underline{Q}^{(k)}P$ is orthogonal and $P(\underline{R}^{(k)})^{-\top}P$ is upper-triangular. Hence, we get a QR factorization of $A^{-k}P$. In other words, we are effectively carrying out simultaneous iteration on $A^{-1}$ applied to the initial matrix $P$, which is to say, simultaneous inverse iteration on $A$. In particular–the last column of $\underline{Q}^{(k)}$–is the result of applying $k$ steps of inverse iteration to the vector $e_m$.

## 28.2 Connection with Shifted Inverse Iteration

Thus the QR algorithm is both simultaneous iteration and simultaneous inverse iteration. Then, we can add shifts to accelerate the QR algorithm. Algo 27.2 showed how shifts are introduced into a step of the QR algorithm. Doing this corresponds exactly to shifts in the corresponding simultaneous iteration and inverse iteration processes, and their beneficial effect is therefore exactly the same.

Let $\mu^{(k)}$ denote the eigenvalue estimate chosen at the $k$th step of the QR algorithm. From Algo 27.2, we have

$$\begin{aligned} A^{(k-1)} - \mu^{(k)} I &= Q^{(k)} R^{(k)}, \\ A^{(k)} &= R^{(k)} Q^{(k)} + \mu^{(k)} I. \end{aligned}$$

This implies,

$$A^{(k)} = (Q^{(k)})^\top A^{(k-1)} Q^{(k)},$$

and by induction,

$$A^{(k)} = (\underline{Q}^{(k)})^\top A \underline{Q}^{(k)},$$

which is the same to (27.15). However (27.3) no longer holds. Instead, we have

$$(A - \mu^{(k)} I)(A - \mu^{(k-1)} I) \cdots (A - \mu^{(1)} I) = \underline{Q}^{(k)} \underline{R}^{(k)}.$$

## 28.3 Connection with Rayleigh Quotient Iteration

We have discovered a powerful tool hidden in the shifted QR algorithm: shifted inverse iteration. To complete the idea, we now need a way of choosing shifts to achieve fast convergence in the last column of $\underline{Q}^{(k)}$.

We can use the Rayleigh quotient. To estimate the eigenvalue corresponding to the eigenvector approximated by the last column of $\underline{Q}^k$, it's natural to apply the Rayleigh quotient. This gives us

$$\mu^{(k)} = \frac{(q_m^{(k)})^\top A q_m^{(k)}}{(q_m^{(k)})^\top q_m^{(k)}} = (q_m^{(k)})^\top A q_m^{(k)}.$$

If this number is chosen as the shift at every step, the eigenvalue and eigenvector estimates $\mu^{(k)}$ and $q_m^{(k)}$ are identical to those that are computed by the Rayleigh quotient iteration starting with $e_m$. Therefore, the QR algorithm has cubic convergence in the sense that $q_m^{(k)}$ converges cubically to an eigenvector.

Notice that, in the QR algorithm, the Raleigh quotient appears as the $(m, m)$ entry of $A^{(k)}$.

$$A_{mm}^{(k)} = e_m^\top A^{(k)} e_m = e_m^\top \underline{Q^{(k)\top}} A \underline{Q}^{(k)} e_m = q_m^{(k)\top} A q_m^{(k)}.$$

Therefore, we can simply set $\mu^{(k)} = A_{mm}^{(k)}$. This is known as the **Rayleigh quotient shift**.

## 28.4 Wilkinson Shift

Although the Rayleigh quotient shift gives cubic convergence in the generic case, convergence is not guaranteed for all initial conditions. Consider

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

The unshifted QR algorithm does not converge at all:

$$\begin{aligned} A = Q^{(1)} R^{(1)} &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \\ A^{(1)} = R^{(1)} Q^{(1)} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = A. \end{aligned}$$

The Rayleigh quotient shift $\mu = A_{mm}$, however, has no effect either, since $A_{mm} = 0$. Thus it's clear that in the worst case, the QR algorithm with the Rayleigh quotient shift may fail.

The problem arises because of the symmetry of the eigenvalues. One eigenvalue is $+1$ and the other is $-1$, so when we attempt to improve the eigenvalue estimate $o$, the tendency to favor each eigenvalue is equal, and the estimate is not improved. What is needed is an eigenvalue estimate that can break the symmetry. One such choice is defined as follows. Let $B$ denote the lower-rightmost $2 \times 2$ submatrix of $A^{(k)}$:

$$B = \begin{bmatrix} a_{m-1} & b_{m-1} \\ b_{m-1} & a_m \end{bmatrix}.$$

The **Wilkinson shift** is defined as that eigenvalue of $B$ that is closer to $a_m$, where in the case of a tie, one of the two eigenvalues of $B$ is chosen arbitrarily. A numerically stable for the Wilkinson shift is

$$\mu = a_m - \operatorname{sign}(\delta) b_{m-1}^2 / \left( |\delta| + \sqrt{\delta^2 + b_{m-1}^2} \right), \tag{28.1}$$

where $\delta = (a_{m-1} - a_m)/2$. If $\delta = 0$, $\operatorname{sign}(\delta)$ can be arbitrarily set equal to 1 or $-1$.

Like the Rayleigh quotient shift, the Wilkinson shift achieves cubic convergence in the generic case. Moreover, it can be shown that it achieves at least quadratic convergence in the worst case. In particular, the QR algorithm with the Wilkinson shift always converges (in exact arithmetic).

For this example, the Wilkinson shift is either $+1$ or $-1$. Thus the symmetry is broken, and convergence takes place in one step.

## 28.5 Stability and Accuracy

As one might expect from its use of orthogonal matrices, the QR algorithm is backward stable. As in previous chapter, the simplest way to formulate this result is to let $\tilde{\Lambda}$ denote the diagonalization of $A$ as computed in floating point arithmetic, and $\tilde{Q}$ the exact orthogonal matrix associated with the product of all numerically computed Householder reflections.

**Theorem 28.1** (Backstab of QR for eig).
Let a real, symmetric, tridiagonal matrix $A \in \mathbb{R}^{m \times m}$ be diagonalized by the QR algorithm (Algo 27.2) on a computer satisfying (12.3) and (12.4), and let $\tilde{\Lambda}$ and $\tilde{Q}$ be defined as indicated above. Then we have

$$\tilde{Q}\tilde{\Lambda}\tilde{Q}^* = A + \delta A, \quad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}) \tag{28.2}$$

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

Besides, we have some further results that the computed eigenvalues $\tilde{\lambda}_j$ satisfy

$$\frac{|\tilde{\lambda}_j - \lambda_j|}{\|A\|} = O(\epsilon_{\text{machine}}). \tag{28.3}$$

This is not a bad result at all for an algorithm that requires just $\sim \frac{4}{3}m^3$ flops, two thirds the cost of computing the product of a part of $m \times m$ matrices!

# CHAPTER 29

# OTHER EIGENVALUE ALGORITHMS

There is more to the computation of eigenvalues than the QR algorithm. In this chapter, we briefly mention three famous alternatives for real symmetric eigenvalue problems: the Jacobi algorithm, for full matrices, and the bisection and divide-and-conquer algorithms, for tridiagonal matrices.

## 29.1 Jacobi

One of the oldest ideas for computing eigenvalues of matrices is the Jacobi algorithm, introduced by Jacobi in 1845. This method has attracted attention throughout the computer era, especially since the advent of parallel computing, though it has never quite managed to displace the competition.

The idea is to diagonalize a small submatrix of $A$, then another, and so on, hoping eventually to converge to a diagonalization of the full matrix.

We start with a $2 \times 2$ real symmetric matrix and its diagonalization:

$$J^\top \begin{bmatrix} a & d \\ d & b \end{bmatrix} J = \begin{bmatrix} \neq 0 & 0 \\ 0 & \neq 0 \end{bmatrix} \tag{29.1}$$

where $J$ is orthogonal. Now there are several ways to choose $J$. One is the Householder reflection:

$$F = \begin{bmatrix} -c & s \\ s & c \end{bmatrix}, \tag{29.2}$$

where $s = \sin\theta$ and $c = \cos\theta$ for $\theta$. Note that $\det F = -1$. Alternatively, one can use a rotation but not a reflection,

$$J = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \tag{29.3}$$

with $\det J = 1$. This is the standard approach for the Jacobi algorithm. It can be shown that the diagonalization is accomplished if $\theta$ satisfies

$$\tan(2\theta) = \frac{2d}{b-a}, \tag{29.4}$$

and the matrix $J$ based this choice is called a **Jacobi rotation**. Note that it has the same form as a Givens rotation, the only difference is that $\theta$ is chosen to make $J^\top A J$ diagonal rather than $J^T A$ triangular.

Now let $A \in \mathbb{R}^{m\times m}$ be symmetric. The Jacobi algorithm of the iterative application of transformations (29.1) based on matrices defined by (29.3) and (29.4). The matrix $J$ is now enlarged to an $m \times m$ matrix that is the identity in all but four entries, where it has the form (29.3). Applying $J^\top$ on the left modifies two rows of $A$, and applying $J$ on the right modifies two columns. At each step a symmetric pair of zeros is introduced into the matrix, but previous zeros are destroyed. Just as with the QR algorithm, however, the usual effect is that the magnitudes of these nonzeros shrink steadily.

The approach naturally fitted to hand computation is to pick the largest off-diagonal entry at each step. We can show that $\sum_{i \neq j} a_{ij}^2$ decreases by at least the factor $1 - \frac{2}{m^2 - m}$ at each step. After $O(m^2)$ steps, each requiring $O(m)$ operations, the sum of squares must drop by a constant factor, and convergence to accuracy $\epsilon_{\text{machine}}$ is assured after $O(m^3 \log(\epsilon_{\text{machine}}))$ operations. In fact, it's known that the convergence is better than this, ultimately quadratic rather than linear, so the actual operation count is $O\left(m^3 \log(|\log(\epsilon_{\text{machine}})|)\right)$.

On a computer, the off-diagonal entries are generally eliminated in a cyclic manner that avoids the $O(m^2)$ search for the largest. For example, if the $m(m-1)/2$ superdiagonal entries are eliminated in the simplest row-wise order, beginning with $a_{12}, a_{13}, \ldots$, then rapid asymptotic convergence is again guaranteed. After one *sweep* of $2 \times 2$ operations involving all of them $\frac{(m-1)}{2}$ pairs of off-diagonal entries, the accuracy has generally improved by better than a constant factor, and again, the convergence is ultimately quadratic.

**Note 29.1.**
The Jacobi method is attractive because it deals only with pairs of rows and columns at a time, making it easily parallelizable. The matrix is not tridiagonalized in advance; the Jacobi rotations would destroy that structure. Convergence for matrices of dimension $m \leqslant 1000$ is typically achieved in fewer than ten sweeps, and the final componentwise accuracy is generally even better than can be achieved by the QR algorithm. Unfortunately, even on parallel machines, the Jacobi algorithm is not usually as fast as tridiagonalization followed by the QR or divide-and-conquer algorithm, though it usually comes within a factor of 10.

## 29.2 Bisection

After a symmetric matrix has been tridiagonalized, this is the standard next step of one does not want all of the eigenvalues but just a subset of them. For example, bisection can find the largest 10% of the eigenvalues, or the smallest thirty eigenvalues, or all the eigenvalues in the in the interval $[1, 2]$. Once the desired eigenvalues are found, the corresponding eigenvectors can bee obtained by one step of inverse iteration (Algo 26.2).

Since the eigenvalues of a real symmetric matrix are real, we can find them by searching the real line for roots of the polynomial $p(x) = \det(A - xI)$. This sounds like a bad idea, for did we not mention in Chapter 14 and 24 that polynomial rootfinding is highly unstable procedure for finding eigenvalues? The difference is that those remarks pertained to the idea of find roots from the polynomial *coefficients*. Now, the idea is to find the roots by evaluating $p(x)$ at various points $x$, without ever looking at its coefficients, and applying the usual bisection process for nonlinear functions. This could be done, for example, by Gaussian elimination with pivoting, and the resulting algorithm would be highly stable.

This much sounds useful enough, but not very exciting. What gives the bisection method its power and its appeal are some additional properties of eigenvalues and determinants that are not immediately obvious.

Given a symmetric matrix $A \in \mathbb{R}^{m \times m}$, let $A^{(1)}, \ldots, A^{(m)}$ denote its principal square submatrices of dimensions $1, \ldots, m$. It can be shown that the eigenvalues of these matrices *interlace*. Before defining this property, let us first sharpen it by assuming that $A$ is tridiagonal and *irreducible* in the sense that of its off-diagonal entries are nonzero:

$$A = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_1 & b_2 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{m-1} \\ & & & b_{m-1} & a_m \end{bmatrix}, \quad b_j \neq 0. \tag{29.5}$$

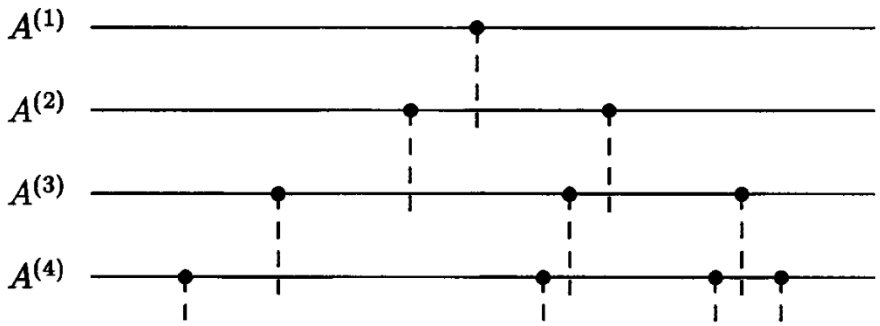



Figure 29.1: Illusion of the strict eigenvalue interlace property from the principal submatrice $\{A^{(j)}\}$ of an irreducible tridagonal real symmetric matrix $A$. The eigenvalues of $A^{(k)}$ interlace those of $A^{(k+1)}$. The bisection algorithm takes advantage of this property.

If there are zeros on the off-diagonal, then the eigenvalue problem can be deflated. Note that the eigenvalues of $A^{(k)}$ are distinct. Let them be denoted by $\lambda_1^{(k)} < \lambda_2^{(k)} < \cdots < \lambda_k^{(k)}$. The crucial property that makes bisection powerful is that these eigenvalues *strictly interlace*, satisfying the inequalities

$$\lambda_j^{(k+1)} < \lambda_j^{(k)} < \lambda_{j+1}^{(k+1)}$$

for $k = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots, k-1$. This behavior is sketched in Figure 29.1.

It is the interlacing property that makes it possible to count the exact number of eigenvalues of a matrix in a specified interval. For example, consider the $4 \times 4$ tridiagonal matrix

$$A = \begin{bmatrix} 1 & 1 & & \\ 1 & 0 & 1 & \\ & 1 & 2 & 1 \\ & & 1 & -1 \end{bmatrix}.$$

From the numbers

$$\det(A^{(1)}) = 1, \quad \det(A^{(2)}) = -1, \quad \det(A^{(3)}) = -3, \quad \det(A^{(4)}) = 4,$$

We know that $A^{(1)}$ has no negative eigenvalues, $A^{(2)}$ has one negative eigenvalue, $A^{(3)}$ has one negative eigenvalue, and $A^{(4)}$ has two negative eigenvalues. In general, for any symmetric tridiagonal $A \in \mathbb{R}^{m \times m}$, the number of negative eigenvalues is equal to the number of sign changes in the sequence

$$1, \det(A^{(1)}), \det(A^{(2)}), \ldots, \det(A^{(m)}),$$

which is known as a **Sturm sequence**. By shifting $A$ by a multiple of the identity, we can determine the number of eigenvalues in any interval $[a, b)$: it is the number of eigenvalues in $(-\infty, b)$ minus the number in $(-\infty, a)$.

One more observation completes the description of the bisection algorithm. Expanding $\det(A^{(k)})$ by minors, we have

$$\det(A^{(k)}) = a_k \det(A^{(k-1)}) - b_{k-1}^2 \det(A^{(k-2)}).$$

Introducing the shift by $xI$ and writing $p^{(k)}(x) = \det(A^{(k)} - xI)$, we get

$$p^{(k)}(x) = (a_k - x)p^{(k-1)}(x) - b_{k-1}^2 p^{(k-2)}(x). \tag{29.6}$$

We can define $p^{(-1)}(x) = 0, p^{(0)}(x) = 1$.

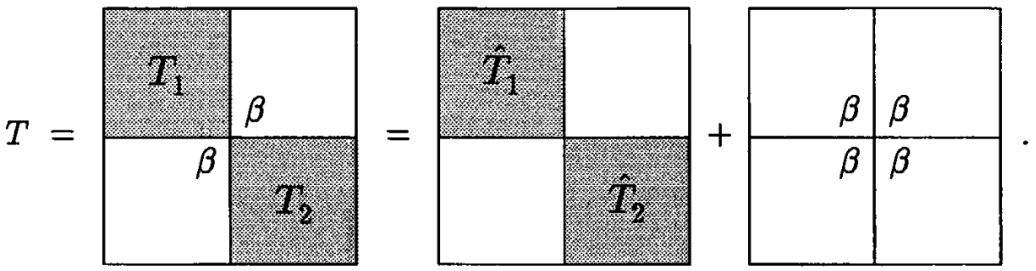
By applying (29.6) for a succession of values of $x$ and counting sign changes along the way, the bisection algorithm locates eigenvalues in arbitrarily small intervals. The cost is $O(m)$ flops for each evaluation of the sequence, hence $O(m \log(\epsilon_{\text{machine}}))$ flops in total to find an eigenvalue to relative accuracy $\epsilon_{\text{machine}}$. If a small number of eigenvalues are needed, this is a distinct improvement over the $O(m^2)$ operation count for the QR algorithm. On a multiprocessor computer, multiple eigenvalues can be found independently on separate processors.

## 29.3 Divide-and-Conquer

The divide-and-conquer algorithm, based on a recursive subdivision of a symmetric tridiagonal eigenvalue problem into problems of smaller dimension, represents the most important advance in matrix eigenvalue algorithms since the 1960s. First introduced by Cuppen in 1981, this methods is more than twice as fast as the QR algorithm if eigenvectors as well as eigenvalues are required.

We shall give just the essential idea, omitting all details. But the reader is warned that in this area, the details are particularly important, for the algorithm is not fully stable unless they are gotten right–a matter that was not well understood for a decade after Cuppen's original paper.

Let $T \in \mathbb{R}^m$ with $m \geqslant 2$ be symmetric, tridiagonal, and irreducible. Then $\forall 1 \leqslant n < m, T$ can be split into submatrices as follows:




where $T_1 \in \mathbb{R}^{n\times n}, T_2 \in \mathbb{R}^{(m-n)\times(m-n)}$ and $\beta = t_{n+1,n} = t_{n,n+1} \neq 0$. Note that $(T_1)_{nn} = (\hat{T}_1)_{nn} + \beta$ and $(T_2)_{11} = (\hat{T}_2)_{11} + \beta$. Besides, the rightmost matrix has rank one. In a word, a tridiagonal matrix can be written as the sum of a $2 \times 2$ block-diagonal matrix with tridiagonal blocks and a rank-one correction.

The divide-and-conquer algorithm proceeds as follows. Split the $T$ with $n \approx \frac{m}{2}$. Suppose the eigenvalues of $\hat{T}_1$ and $\hat{T}_2$ are known. Since the correction matrix is of rank one, a nonlinear but rapid calculation can be used to get from the eigenvalues of $\hat{T}_1$ and $\hat{T}_2$ to those of $T$ itself. Now recurse on this idea, finding the eigenvalues of $\hat{T}_1$ and $\hat{T}_2$ by further subdivisions with rank-one corrections, and so on. In this manner an $m \times m$ eigenvalue problem is reduced to a set of $1 \times 1$ eigenvalue problems together with a collection of rank-one corrections. (In practice, for maximal efficiency, it's customary to switch to the QR algorithm when the submatrices are of sufficiently small dimension rather than to carry the recursion all the way.).

In this process there is one key mathematical point. If the eigenvalues of $\hat{T}_1$ and $\hat{T}_2$ are known, how can those of $T$ be found? Assume

$$\hat{T}_1 = Q_1 D_1 Q_1^\top, \quad \hat{T}_2 = Q_2 D_2 Q_2^\top$$

have been computed. Then it follows that

$$T = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \left( \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} + \beta z z^\top \right) \begin{bmatrix} Q_1^\top & \\ & Q_2^\top \end{bmatrix} \tag{29.7}$$

with $z^\top = (q_1^\top, q_2^\top)$, where $q_1^\top$ is the last row of $Q_1$ and $Q_2^\top$ is the first row of $Q_2$. Hence, we have have to figure the eigenvalues of a diagonal matrix plus a rank-one correction.

We show how this is done. We consider a simplified example. Suppose we wish to find the eigenvalues of $D + ww^\top$, where $D \in \mathbb{R}^{m\times m}$ is a diagonal matrix with distinct diagonal entries $\{d_j\}$ and $w \in \mathbb{R}^m$ is a vector. Here, we assume $w_j \neq 0$ for all $j$, for otherwise, the problem is reducible. Then the eigenvalues of $D + ww^\top$ are the roots of the rational function

$$f(\lambda) = 1 + \sum_{j=1}^{m} \frac{w_j^2}{d_j - \lambda},$$

as illustrated in Figure 29.2.

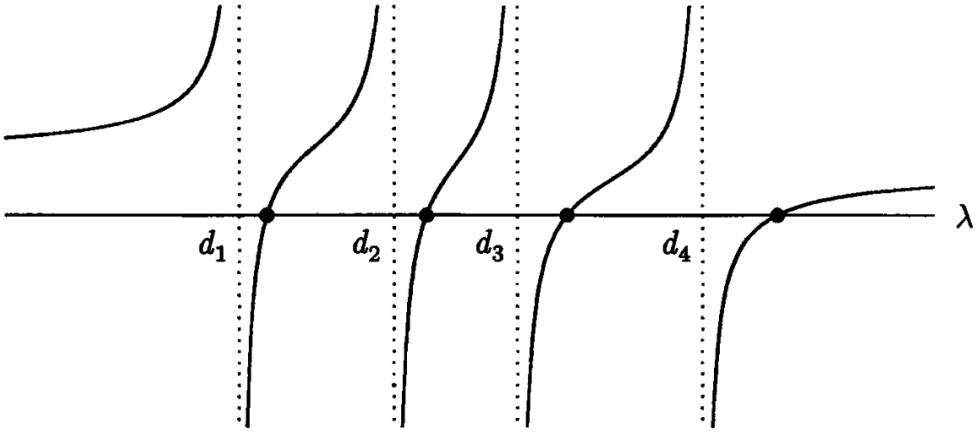



Figure 29.2: Plot of the function $f(\lambda)$ for a problem of dimension 4. The poles of $f(\lambda)$ are the eigenvalues $\{d_j\}$ of $D$, and the roots of $f(\lambda)$ are the eigenvalues of $D + ww^\top$. The rapid determination of these roots is the basis of each recursive step of the divide-and-conquer algorithm.

This assertion can be justified by that

$$(D+ww^\top)q = \lambda q \Rightarrow (D-\lambda I)q+w(w^\top q) = 0 \Rightarrow q+(D-\lambda I)^{-1}w(w^\top q) = 0 \Rightarrow w^\top q+w^\top(D-\lambda I)^{-1}w(w^\top q) = 0.$$

Hence, we have $f(\lambda)(w^\top q) = 0$, in which $w^\top q$ must be nonzero. Hence, if $q$ is an eigenvector of $D+ww^\top$ with eigenvalue $\lambda$, then $f(\lambda) = 0$. The equation $f(\lambda)$ is known as the **secular equation**.

At each step of the divide-and-conquer algorithm, the roots are found by Newton's method. Only $O(1)$ iterations are required for each root (or $O(\log(|\log(\epsilon_{\text{machine}})|))$ iterations if $\epsilon_{\text{machine}}$ is viewed as a variable), making the operation count $O(m)$ flops per root for an $m \times m$ matrix, or $O(m^2)$ all tother. If we image a recursion in which a matrix of dimension $m$ is split exactly in half at each step, the total operation count for finding eigenvalues of a tridiagonal matrix by the divide-and-conquer algorithm becomes

$$O\left(m^2 + \left(\frac{m}{2}\right)^2 + 4\left(\frac{m}{4}\right)^2 + \cdots + m\left(\frac{m}{m}\right)^2\right), \tag{29.8}$$

a series which converges to $O(m^2)$. Thus the operation count would appear to be of the same order $O(m^2)$ as for the QR algorithm.

So far, it is not clear why the divide-and-conquer algorithm is advantageous. Since the reduction of a full matrix to tridiagonal form requires $\frac{4m^3}{3}$ flops, it would seeme that any improvement in the $O(m^2)$ operation count for diagonalization of that tridiagonal matrix is hardly important. However, the economics change if one is computing eigenvectors as well as eigenvalues. Now, Phase 1 requires $\frac{8m^3}{3}$ but Phase 2 also requires $O(m^3)$ flops–for the QR algorithm, $\approx 6m^3$. The divide-and-conquer algorithm reduces this figure, ultimately because its nonlinear iterations involve just the scalar function, not the orthogonal matrices $Q_j$, whereas the QR algorithm must manipulate matrices $Q_j$ at every iterative step.

An operation count reveals the following. The $O(m^3)$ part of the divide-and-conquer computation is the multiplication by $Q_j$ and $Q_j^\top$ in (29.7). The total operation count, summed over all steps of recursion is $\frac{4m^3}{3}$ flops a great improvement over $\approx 6m^3$ flops. Adding in the $\frac{8m^3}{3}$flops for Phase 1 gives an improvement from $\approx 9m^3$ to $4m^3$.

Actually, the DC algorithm uusually does better, since for most matrices $A$, many of the vectors $z$ and matrices $Q_j$ that arise in (29.7) turn out to be numerically sparse in the sense that many of their entries have relative magnitudes less than machine precision. This sparsity allows a process of numerical deflation, whereby successive tridiagonal eigenvalue problems are reduced to uncoupled problems of smaller dimensions. In typical cases, this reduces the Phase 2 operation count to an order less than $m^3$ flops, reducing the operation count for Phase 1 and 2 combined to $\frac{8m^3}{3}$. For eigenvalueus alone, the cost (29.8) becomes an overestimate and the Phase 2 operation count is reduced to an order lower than $m^2$ flops. The root of this fascinating phenomenon of deflation, which we shall not discuss

further, is the fact that most of eigenvectors of most tridiagonal matrices are "exponentially localized"–a fact that has related by physicists to the phenomenon that glass is transparent.

We have spoken as if there is a single divide-and-conquer algorithm , but in fact, there are many variants. MOst complicated rank-one updates are often used for stability reasons, and rank-two updates are also sometimes used. Various methods are employed for finding the roots of $f(\lambda)$, and for large $m$, the fastest way to carry out the multiplications by $Q_j$ is via multipole expansions rather than the obvious algorithm. A high-quality implementation of a divide-and-conquer algorithm can be found in the LAPACK library.

# CHAPTER 30

# COMPUTING THE SVD

The computation of the SVD of an arbitrary matrix can be reduced to the computation of the eigenvalue decomposition of a hermitian square matrix, but the most obvious way of doing this is not stable. Instead, the standard methods for computing the SVD are based implicitly on another kind of reduction to hermitian form. For speed, the matrix is first unitarily diagonalized.

## 30.1 SVD of $A$ and Eigenvalues of $A^*A$

The SVD of $A \in \mathbb{R}^{m\times n}$ ,$A = U\Sigma V^*$ is related to the eigenvalue decomposition of the matrix $A^*A$,

$$A^*A = V\Sigma^*\Sigma V^*.$$

Thus, we can calculate the SVD of A as follows:

1. Form $A^*A$;
2. Compute the eigenvalue decomposition $A^*A = V\Lambda V^*$;
3. Let $\Sigma$ be the $m \times n$ nonnegative diagonal square root of $\Lambda$;
4. Solve the system $U\Sigma = AV$ for unitary $U$ (e.g. via QR).

This algorithm is frequently used, often by people who have rediscovered the SVD for themselves. The matrix $A^*A$ is known as the *covariance matrix* of $A$, and it has familiar interpretations in statistics and other fields. The algorithm is unstable, however, because it reduces the SVD problem to an eigenvalue problem that may be much sensitive to perturbations.

The difficulty can be explained as follows. We have seen that when a hermitian matrix $A^*A$ is perturbed by $\delta B$, the absolute changes in each eigenvalue are bounded by the 2-norm of the perturbation. In detail, we have

$$|\lambda_k(A^*A + \delta B) - \lambda_k(A^*A)| \leqslant \|\delta B\|_2.$$

As is implied by (30.1) below, a similar bound holds for the singular values of $A$ itself, $|\sigma_k(A + \delta A) - \sigma_k(A)| \leqslant \|\delta A\|_2$. Thus a backward stable algorithm for computing singular values would obtain $\tilde{\sigma}_k$ satisfying

$$\tilde{\sigma}_k = \sigma_k(A + \delta A), \quad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}),$$

which would imply that

$$|\tilde{\sigma}_k - \sigma_k| = O(\epsilon_{\text{machine}}\|A\|).$$

Now observe what happens if we proceed by computing $\lambda_k(A^*A)$. If $\lambda_k(A^*A)$ is computed stably, we must expect errors of the order

$$|\tilde{\lambda}_k - \lambda_k| = O(\epsilon_{\text{machine}}\|A^*A\|) = O(\epsilon_{\text{machine}}\|A\|^2).$$

Square-rooting to get $\sigma_k$, we find

$$|\tilde{\sigma}_k - \sigma_k| = O(|\tilde{\lambda}_k - \lambda_k|/\sqrt{\lambda_k}) = O(\epsilon_{\text{machine}}\|A\|^2/\sigma_k).$$

This is worse than the previous result by a factor of $(\|A\|/\sigma_k)$. This is no problem for the dominant singular values of $A$, with $\sigma_k \approx \|A\|$, but it's a big problem for any singular values with $\sigma_k \ll \|A\|$. For the smallest singular value $\sigma_n$, we must expect a loss of accuracy of order $\kappa(A)$.

## 30.2 A Different Reduction

Assume $A$ is square, we consider the $2m \times 2m$ hermitian matrix

$$H = \begin{bmatrix} 0 & A^* \\ A & 0 \end{bmatrix}. \tag{30.1}$$

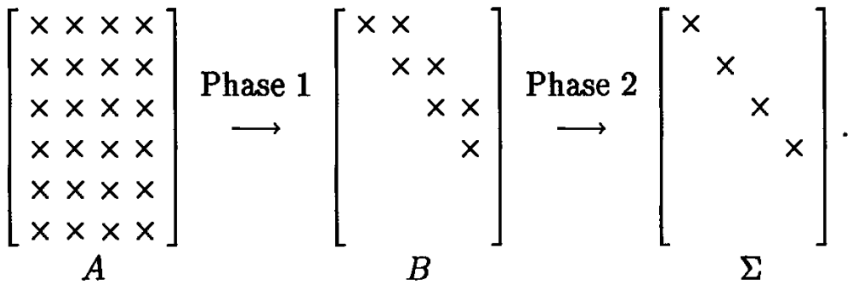
Since $A = U\Sigma V^*$ implies $AV = U\Sigma$ and $A^*U = V\Sigma^* = V\Sigma$, we have

$$\begin{bmatrix} 0 & A^* \\ A & 0 \end{bmatrix} \begin{bmatrix} V & V \\ U & -U \end{bmatrix} = \begin{bmatrix} V & V \\ U & -U \end{bmatrix} \begin{bmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{bmatrix}, \tag{30.2}$$

which is an eigenvalue decomposition of $H$. Thus we can obtain the SVD of $A$ by the eigenvalue decomposition of $H$. In contrast to the use of $AA^*$ or $A^*A$, this approach is stable. A key step to make this process fast is an initial unitary reduction to bidiagonal form.

## 30.3 Two Phases

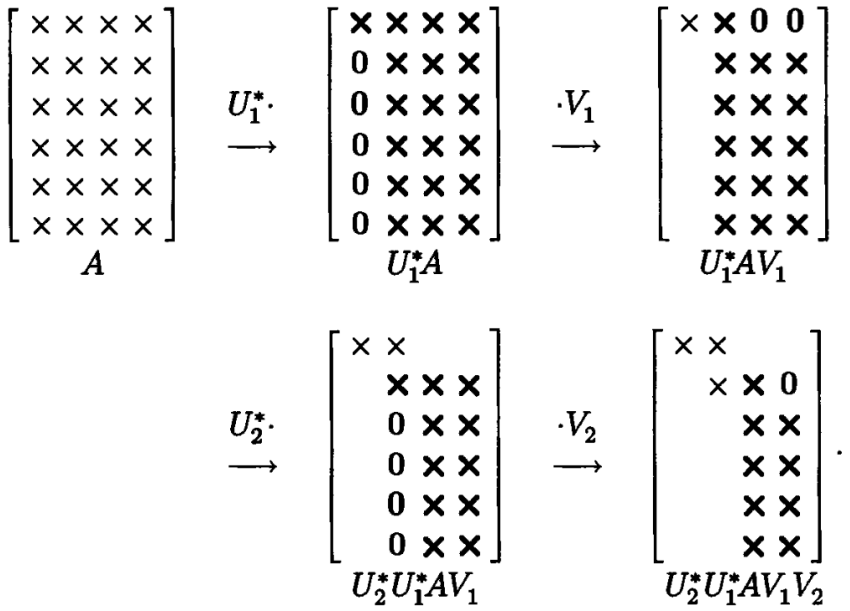
Since the work of Golub, Kahan, and others in the 1960s, an analogous two-phase approach has been standard for the SVD. The matrix is brought into bidiagonal form, and then the bidiagonal matrix is diagonalized:




Phase 1 involves $O(mn^2)$ flops. Phase 2 in principal requires an infinite number of operations, but the standard algorithms converge superlinearly, and thus only $O(n \log(|\log(\epsilon_{\text{machine}})|))$ iterations are needed for convergence to order $\epsilon_{\text{machine}}$. Because the matrix operated on is bidiagonal, each of these iterations requires only $O(n)$ flops. Phase 2 therefore requires $O(n^2)$ flops all tother. Thus, although Phase 1 is finite and Phase 2 is in principle infinite, in practice the latter is much the less expensive, just as we found for the symmetric eigenvalue problem.

## 30.4 Golub-Kahan Bidiagonalization

Note that the difference to eigenvalue decomposition is that we don't similar transformation and we can completely triangularize and also introduce zeros above the first superdiagonal. The simplest method for accomplishing this is the **Glub-Kahan bidiagonalization**. House holder reflectors are applied alternately on the left and the right. For example

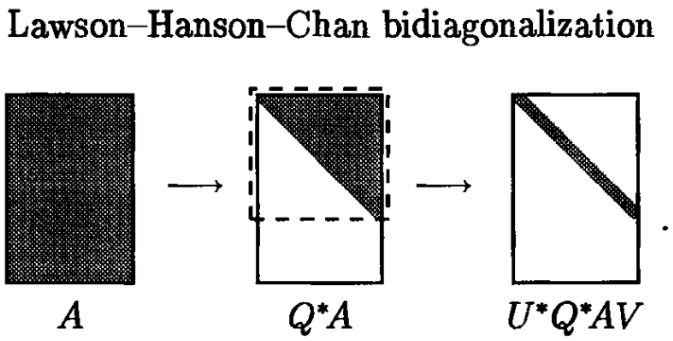



At the end of this process, $n$ reflectors have been applied on the left and $n-2$ on the right. The total operation count is therefore twice that of a QR factorization.

**Proposition 30.1.**
The work for Golub-Kahan bidiagonalization: $\approx 4mn^2 - \frac{4}{3}n^3$ flops.

## 30.5 Faster Methods for Phase 1

For $m \gg n$, this operation count is unnecessarily large. A single QR factorization would introduce zeros everywhere below the diagonal, and for $m \gg n$, there are the great majority of the zeros that are needed. Hence, we can consider the an alternative method for bidiagonalization with with $m \gg n$, first proposed by Lawson and Hanson and later developed by Chan. The idea, LHC bidiagonalization is illustrated as follows:

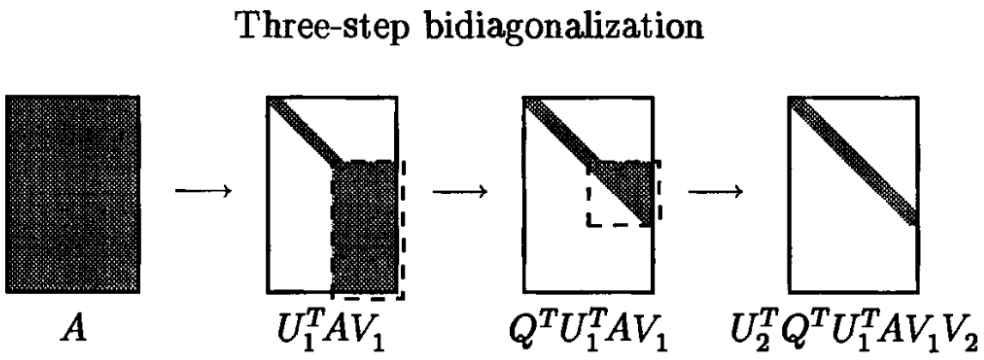



We do QR $A = QR$ first and then compute the Golub-Kahan bidiagonalization $B = U^*RV$ of $R$. The QR needs $2mn^2 - \frac{2}{3}n^3$ flops (Corollary 9.1), and the Gould-Kahan procedure which requires $\frac{8}{3}n^3$ flops. The total operation count is:

**Corollary 30.2.**
The work for LHC bidiagonalization: $\approx 2mn^2 + 2n^3$ flops.

This is cheaper than Golub-Kahan bidiagonalization for $m > \frac{5}{3}n$.

The LHC procure is advantageous only when $m > \frac{5}{3}n$, but the idea can be generalized so as to realize a saving for any $m > n$. The trick is to apply the QR not the beginning of the computation, but at a suitable point in the middle.




When should the QR factorization be performed? If we aim solely to minimize the operation count, the answer is simple: when the aspect ratio reaches $(m-k)/(n-k) = 2$. This choice leads to the formula:

**Corollary 30.3.**
Work for three-step bidiagonalization: $\sim 4mn^2 - \frac{4}{3}n^3 - \frac{2}{3}(m-n)^3$ flops.

This is a modest improvement over the other two methods for $n < m < 2n$. It must be admitted that the improvement achieved by the three-step method is small enough that in practice, other matters besides the count may determine which method is best on a real machine.

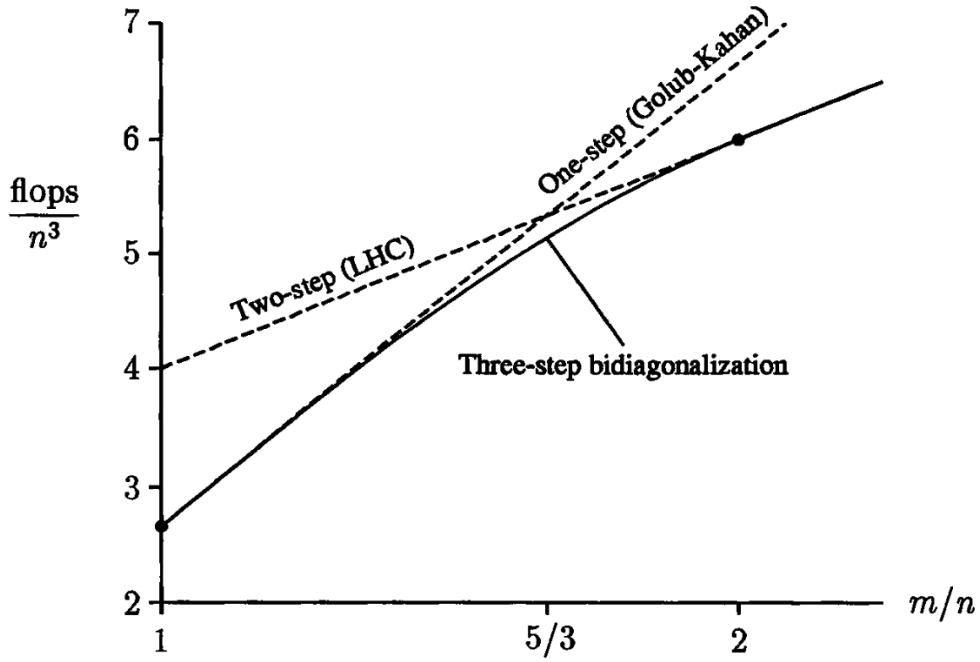



Figure 30.1: Operation counts for three bidiagonalization algorithms applied to $m \times n$ matrices. Three-step bidiagonalization provides a pleasing smooth interpolant between the other two methods, thought the improvement is hardly large.

## 30.6 Phase 2

In Phase 2 of the computation of the SVD, the SVD of the bidiagonal matrix B is determined. From the 1960s to the 1990s, the standard algorithm for this was a variant of the QR algorithm. More recently, divide-and-conquer algorithms have also become competitive, and in the future, they are likely to become the standard. We shall not give details.

# Part VI

# Iterative Methods

# CHAPTER 31

# OVERVIEW OF ITERATIVE METHODS

With this lecture the flavor of the book changes. We move from direct methods, a classical topic that is rather thoroughly understood, to the relatively untamed territory of iterative methods. These are the methods that seem likely to dominate the large-scale computations of the future.

## 31.1 Why iterate?

- Noniterative or "direct" algorithms require $O(m^3)$ work.
- Iterative methods can solve matrix problems in $O(m^2)$ operations.

## 31.2 Structure, Sparsity, and Black Boxes

- Large matrix problems are far from random and they might have sparsity or structure.
- The iterative algorithm requires nothing more thant the ability to determine $Ax$ for any $x$, which can make use of the sparsity.
- Gaussian or Householder triangularization will destroy the sparsity.

## 31.3 Projection into Krylov Subspaces

**Definition 31.1** (Krylov subspaces).
Given $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$, the associated Krylov sequence is

$$b, Ab, A^2b, A^3b, \dots$$

The corresponding Krylov subspaces are the spaces $\{\langle b, Ab, A^2b, \dots, A^mb\rangle\}_{m=1}^{\infty}$.

The algorithms we shall discuss can be arranged in the following table:

| | $Ax = b$ | $Ax = \lambda x$ |
|---|---|---|
| $A = A^*$ | CG | Lanczos |
| $A \neq A^*$ | GMRES<br>CGN<br>BCG et al. | Arnoldi |

## 31.4 Works and Preconditioning

- Gaussian elimination, QR factorization, and most other algorithms of dense linear algebra fit the following pattern: there are $O(m)$ steps, each requiring $O(m^2)$ work, for a total work estimate of $O(m^3)$.
- The ideal iterative method in linear algebra reduces the number of steps from $m$ to $O(1)$ and the work per step from $O(m^2)$ to $O(m)$, reducing the total work from $O(m^3)$ to $O(m)$. Such extraordinary speedups do occur in practical problems, but a more typical improvement is perhaps from $O(m^3)$ to $O(m^2)$.

## 31.5 Direct Methods that Beat $O(m^3)$

Finally, we must mention that there exist direct algorithms-finite, in principle exact-that solve $Ax = b$ and related problems in less than $O(m^3)$ operations. The first algorithm of this kind was discovered in 1969 by Volker Strassen, who reduced Gauss's exponent of 3 to $\log_2(7) \approx 2.81$, and subsequent improvements have reduced the best known exponent to its current value of $\approx 2.376$ due to Coppersmith and Winograd. The history of these best known exponents is recorded in Figure 31.1.

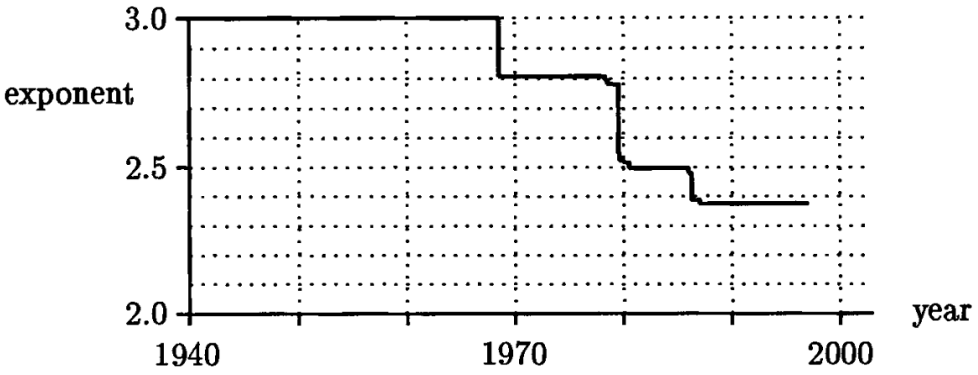



Figure 31.1: Best known exponents for direct solution of $Ax = b$ for $m \times m$ matrices, as a function of time. Until 1968, the best known algorithms were of complexity $O(m^3)$. The current best known algorithm solves $Ax = b$ in $O(m^{2.376})$ flops, but the constants are so large that this algorithm is impractical.

# CHAPTER 32

# CLASSICAL METHODS

In this Chapter, we introduce three classical iterative methods for the linear system $Ax = b$.

## 32.1 Jacobi Method

Given

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \in \mathbb{R}^{n \times n},$$

we define,

$$D = \text{diag}(A), \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

Then, the iteration is

$$x^{(k+1)} = D^{-1}(b - (L + U)x^{(k)}).$$

And the element-based formula for each row is:

$$x_i^{(k)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}).$$

As for the convergence, we know that

$$(D + L + U)x_* = b \Rightarrow x_* = D^{-1}(b - (L + U)x_*).$$

Combine this with the iteration, we have

$$(x^{(k+1)} - x_*) = D^{-1}(L + U)(x^{(k)} - x_*).$$

Hence, we can get some convergence condition.

**Corollary 32.1** (Convergence of Jacobi Method).
Jacobi method will converge if:

- $\|D^{-1}(L + U)\|_2 \leqslant 1$ or,
- $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$.

## 32.2 Gauss-Seidel Method

With the same settings as the Jacobi method, the iteration is

$$x^{(k+1)} = (L + D)^{-1}(b - Ux^{(k)}).$$

Similarly, we have

$$(x^{(k+1)} - x_*) = (L + D)^{-1}U(x_* - x^{(k)}).$$

**Corollary 32.2.**
Gauss-Seidel method will converge if either:

- $A$ is symmetric and strictly positive definite or,
- $|a_{ii}| > \sum_{j\neq i} |a_{ij}|$.

*Proof sketch.*
For the first part, we only need to care $G_{GS} = -(L + D)^{-1}U$. This is similar to

$$G = D^{\frac{1}{2}} G_{GS} D^{-\frac{1}{2}} = -(I + \hat{L})^{-1}\hat{L}^\top,$$

where $\hat{L} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$. If

$$-(I + \hat{L})^{-1}\hat{L}^\top v = \lambda v,$$

then $-v^\top \hat{L}^\top v = \lambda(1 + v^\top \hat{L} v)$. If $v^\top L v = a + bi$, then

$$|\lambda|^2 = \left|\frac{-a + bi}{1 + a + bi}\right|^2 = \frac{a^2 + b^2}{1 + 2a + a^2 + b^2}.$$

Beside, $D^{-\frac{1}{2}} A D^{-\frac{1}{2}} = I + \hat{L} + \hat{L}^\top$ is positive define. Hence, $1 + 2a > 0$ and hence $|\lambda| < 1$.

For the second part, we just use the induction method. □

## 32.3 SOR Method

SOR is the abbreviation for successive over-relaxation. The update is

$$(D + wL)x = wb - [wU + (w - 1)D]x.$$

The error part is

$$(D + wL)(x^{(k+1)} - x_*) = -[wU + (w - 1)D](x^{(k)} - x_*).$$

Ostrowski proved the following theorem in 1947.

**Theorem 32.3 (Convergence of SOR).**
If $A$ is symmetric and positive-definite,

$$\rho(L_w) < 1, \quad \forall 0 < w < 2.$$

# CHAPTER 33

# THE ARNOLDI ITERATION

Despite the many names and acronyms that have proliferated in the field of Krylov subspace matrix iterations, these algorithms are built upon a common foundation of a few fundamental ideas. One can take various approaches to describing this foundation. Ours will be to consider the Arnoldi process, a Gram—Schmidt-style iteration for transforming a matrix to Hessenberg form.

## 33.1 The Arhnoli/Gram-Shmidt Analogy

The goal of Arnoldi iteration is to reduce a nonhermitian matrix to Hessenberg form by orthogonal transformations, proceeding column by column from a prescribed first column $q_1$.

We have discussed two methods for QR factorization:

- Householder reflections, triangularize $A$ by a succession of orthogonal operations;
- Gram-Schmidt orthogonalization, orthogonalize $A$ by a succession of triangular operations.

Note that the GS process has the advantage that it can be stopped aprt-way, leaving one with a reduced QR factorization of the first $n$ columns of $A$. The problem of computing a Hessenberg reduction $A = QHQ^*$ of a matrix $A$ is exactly analogous. There are two standard methods: House holder reflections and the Arnoldi iteration.

## 33.2 Mechanics of the Arnoldi Iteration

Give n $A \in \mathbb{R}^{m\times m}$, an Hessenberg form is $A = QHQ^*$. However, in dealing with iterative methods we take the view that $m$ is huge or infinite, so that computing the full reduction os out of question. Let $Q^n \in \mathbb{R}^{m\times n}$ be the first $n$ columns of $Q$ and $Q = (q_1, q_2, \ldots, q_n)$. Let $\tilde{H}_n \in \mathbb{R}^{(n+1)\times n}$ be the upper-left section of $H$, which is also Hessenberg:

$$H = \begin{bmatrix} h_{11} & & \cdots & h_{1n} \\ h_{21} & h_{22} & & \vdots \\ & \ddots & \ddots & \vdots \\ & & h_{n,n-1} & h_{nn} \\ & & & h_{n+1,n} \end{bmatrix}. \tag{33.1}$$

Then we have

$$AQ_n = Q_{n+1}\tilde{H}_n.$$

The $n$th column of this equation ca nbe written as follows:

$$Aq_n = h_{1n}q_1 + \cdots + h_{nn}q_n + h_{n+1,n}q_{n+1}. \tag{33.2}$$

Then, we have the Arnoldi iteration:

**Algorithm 33.1:** Arnoldi Iteration

```
random b, q_1 = b/||b||;
for n = 1, 2, 3, ... do
    v = A q_n;
    for j = 1 to n do
        h_jn = q_j^* v;
        v = v − h_jn q_j;
    h_{n+1,n} = ||v||;
    q_{n+1} = v / h_{n+1,n}
```

Note that the matrix $A$ only appears in the product $Aq_n$, which can be computed by a black box procedure.

## 33.3 QR Factorization of a Krylov Matrix

It is evident from this formula that the vectors $\{q_j\}$ form bases of the successive Krylov subspaces generated by $A$ and $b$, defined as follows:

$$\mathcal{K}_n = \langle b, Ab, \ldots, A^{n-1}b \rangle = \langle q_1, q_2, \ldots, q_n \rangle \subseteq \mathbb{C}^m.$$

It's obvious that $\{q_j\}_{j=1}^n$ is an orthonormal basis for $\mathcal{K}_n$. We define $K_n \in \mathbb{R}^{m\times n}$ as $K_n = (b, Ab, \ldots, A^{n-1}b)$. Then, $K_n$ must a reduced QR factorization,

$$K_n = Q_n R_n, \tag{33.3}$$

where $Q_n$ is the same matrix as above. This give an intuition explanation of why the Arnoldi process leads to effective methods for determining certain eigenvalues. Clearly $K_n$ might be expected to contain good information about the eigenvalues of $A$ with largest modules, and the QR factorization might be expected to reveal this information by peeling off one approximate eigenvector after another.

## 33.4 Projection onto Krylov Subspaces

Another way to view the Arnoldi process is as a computation of projections onto successive Krylov subspaces. Note that $Q_n^* Q_{n+1}$ is the $n \times (n+1)$ identity, and

$$H_n = Q_n^* Q_{n+1} \tilde{H}_n = \begin{bmatrix} h_{11} & \cdots & \cdots & h_{1n} \\ h_{21} & h_{22} & & \vdots \\ & \ddots & \ddots & \vdots \\ & & h_{n,n-1} & h_{nn} \end{bmatrix} \in \mathbb{R}^{n\times n}.$$

Hence,

$$H_n = Q_n^* Q_{n+1} \tilde{H}_n = Q_n^* A Q_n.$$

This matrix can be interpreted as the representation in the basis $\{q_1, \ldots, q_n\}$ of the orthogonal projection of $A$ onto $\mathcal{K}_n$. Is it clear what this interpretation means? Here is a precise statement. Consider the linear operator $\mathcal{K}_n \to \mathcal{K}_n$ defined as follows: given $v \in \mathcal{K}_n$, apply $A$ to it, then orthogonally project $Av$ back into the space $\mathcal{K}_n$. Since the orthogonal projector of $\mathbb{C}^m$ onto $\mathcal{K}_n$ is $Q_n Q_n^*$, this operator can be written $Q_n Q_n^* A$ with respect to the standard basis of $\mathbb{C}^m$. With respect to the basis of columns of $Q_n$, it can therefore be written $Q_n^* A Q_n$.

Since $H_n$ is a projection of $A$, one might imagine that its eigenvalues would be related to those of $A$ in a useful fashion. These $n$ numbers,

$$\{\theta_j\} = \{ \text{ eigenvalues of } H_n \}, \tag{33.4}$$

are called the **Arnoldi eigenvalue estimates** or **Ritz values** of $A$.

**Theorem 33.1** (Arnoldi iteration).
The matrices $Q_n$ generated by the Arnoldi iteration are reduced QR factors of the Krylov matrix:

$$K_n = Q_n R_n. \tag{33.5}$$

The Hessenberg matrices $H_n$ are the corresponding projections,

$$H_n = Q_n^* A Q_n, \tag{33.6}$$

and the successive iterates are related by the formula

$$A Q_n = Q_{n+1} \tilde{H}_n. \tag{33.7}$$

# CHAPTER 34

# HOW ARNOLDI LOCATES EIGENVALUES

The Arnoldi iteration is two things:

- The basis of many of the iterative algorithms of NLA
- A techniques for finding eigenvalues of nonhermitian matrices.

## 34.1 Computing Eigenvalues by the Arnoldi Iteration

This is how we compute the eigenvalues: at each iteration, we compute the eigenvalues of $H_n$ by QR algorithm. Some of these numbers are typically observed to converge rapidly, often geometrically. Suppose $n \ll m$, typically, the Arnoldi iteration will find extreme eigenvalues, which are near the edge of the spectrum of $A$.

## 34.2 A note of Caution: Nonnormality

If the answer is highly sensitive to perturbations, you have probably asked the wrong question. We urge anyone faced with nonhermitian eigenvalue computations involving highly sensitive eigenvalues to bear this principle in mind. If you are a numerical analyst, and the problem was given to you by a colleague in science or engineering, do not accept without scrutiny your colleague's assurances that it is truly the eigenvalues that matter physically, even though their condition numbers with respect to perturbations of the matrix entries are $10^4$.

## 34.3 Arnoldi and Polynomial Approximation

$\forall x \in \mathcal{K}_n$, we have

$$x = c_0 b + c_1 Ab + c_2 A^2 b + \cdots + c_{n-1} A^{n-1} b = q(A)b,$$

where $q(z) = c_0 + c_1 z + c_2 z^2 + \cdots + c_{n-1} z^{n-1}$. One analysis of Arnoldi iteration considers

$$P^n = \{ \text{ monic polynomials of degree } n\}.$$

We have the following problem.

**Example 34.1** (Arnoldi/Lanczos Approximation Problem).

$$\arg \min_{p^n \in P^n} \|p^n(A)b\|. \tag{34.1}$$

The Arnoldi iteration has the remarkable property that it solves this problem exactly.

**Theorem 34.2.**
As long as the Arnoldi iteration does not break down (i.e., $K_n$ is of full rank $n$), (34.1) has a unique solution $p^n$, the characteristic polynomial of $H_n$.

*Proof sketch.*
Note that $p(A)b = A^n b - Q_n y$ for some $y \in \mathbb{C}^n$. Hence, we are projecting $A^n b$ to $\mathcal{K}_n$ and we must have $p^n(A)b \perp \mathcal{K}_n \Leftrightarrow Q_n^* p^n(A)b = 0$. Now we replace $A$ with its decomposition $QHQ^*$. At step $n$, we know that

$$Q = [Q_n U], \quad H = \begin{bmatrix} H_n & X_1 \\ X_2 & X_3 \end{bmatrix}.$$

Here $(X_2)_{ij} = 0$ except $(X_2)_{11}$. Plug in this into our condition, we get

$$Q_n^* Q p^n(H) Q^* b = Q_n^* Q p^n(H) \|b\| e_1 = I_{n,m} p^n(H) \|b\| e_1 = 0.$$

Hence, the first $n$ entries of the first column of $p^n(H)$ are all zero. Note that these are also the first $n$ entries of the first column of $p^n(H_n)$. Because we can check $H \cdot H^m$ to verify this. Since the characteristic polynomial of $H_n$ satisfies the condition, this is one solution. If we we have another $p^n$, we subtract them and get a polynomial $q$ of degree $n-1$ and $q(A)b \perp \mathcal{K}_n$. Note that $q(A)b$ is also in $\mathcal{K}_n$. Hence, $q(A)b = 0$, violating the assumption that $\mathcal{K}_n$ is of full rank.

□

## 34.4 Invariance Properties

Since the family $P^n$ of monic polynomials is invariant with respect to translations $z \mapsto z + \alpha$, the Arnoldi iteration is also translation-invariant.

**Theorem 34.3** (Invariance of Arnoldi iteration).
Let the Arnoldi iteration be applied to a matrix $A \in \mathbb{C}^{m \times m}$ as described above.

Translation-invariance. If $A$ is changed to $A + \sigma I$ for some $\sigma \in \mathbb{C}$, and $b$ is left unchanged, the Ritz values $\{\theta_j\}$ at each step change to $\{\theta_j + \sigma\}$.

Scale-invariance. If $A$ is changed to $\sigma A$ for some $\sigma \in \mathbb{C}$, and $b$ is left unchanged, the Ritz values $\{\theta_j\}$ change to $\{\sigma\theta_j\}$.

Invariance under unitary similarity transformations. If $A$ is changed to $UAU^*$ for some unitary matrix $U$, and $b$ is changed to $Ub$, the Ritz values $\{\theta_j\}$ do not change.

In all three cases the Ritz vectors, namely the vectors $Q_n y_j$ corresponding to the eigenvectors $y_j$ of $H_n$, do not change under the indicated transformation.

every matrix $A$ is unitarily similar to an upper-triangular matrix. Thus the property of invariance under unitary similarity transformations implies that for understanding convergence properties of the Arnoldi iteration, it is enough in principle to consider upper-triangular matrices.

## 34.5 How Arnoldi Locates Eigenvalues

What does polynomial approximation have to do with the eigenvalues of $A$ ? A little thought shows that there is a connection between these two along the following lines. If one's aim is to find a polynomial $p^n$ with the property that $p^n(A)$ is small, an effective means to that end may be to pick $p^n$ to have zeros close to the eigenvalues of $A$.

Consider an extreme case. Suppose that $A$ is diagonalizable and has only $n \ll m$ distinct eigenvalues, hence a minimal polynomial of degree $n$. Then from Theorem 34.2 it is clear that after $n$ steps, all of these eigenvalues will be found exactly, at least if the vector $b$ contains components in directions associated with every eigenvalue. Thus after $n$ steps, the Arnoldi iteration has computed the minimal polynomial of $A$ exactly.

In practical applications, much the same phenomenon takes place. Now, however, the agreement of Ritz values with eigenvalues is approximate instead of exact, and instead of a minimal polynomial, the result is a "pseudo-minimal polynomial," i.e., a polynomial $p^n$ such that $\|p^n(A)\|$ is small.

## 34.6 Arnoldi Lemniscates

**Definition 34.4** (Lemniscate).
Given a polynomial $p$ and a constant $C$, a lemniscate is a curve or collection of curves

$$\{z \in \mathbb{C} : |p(z)| = C\}.$$

If $p$ is the Arnoldi polynomial and $C = \frac{\|p^n(A)b\|}{\|b\|}$, the curves are called Arnoldi lemniscates.

As the iteration number $n$ increases, components of these lemniscates typically appear which surround the extreme eigenvalues of $A$ and then shrink rapidly to a point, namely the eigenvalue itself.

One example is let $A \in \mathbb{R}^{100\times 100}$ and $a_{ij} \sim \mathcal{N}(0, \frac{1}{100})$. We have the eigenvalues are approximately uniformly distributed in the unit disk $|z| \leqslant 1$. Now we set $a_{11} = 1.5$.

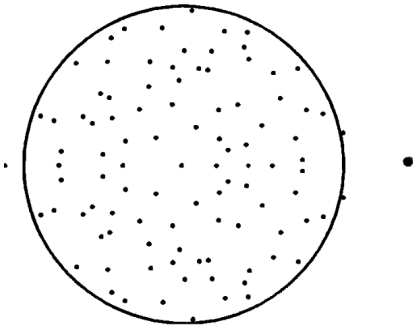


Figure 34.1: Eigenvalues of $a100 \times 100$ matrix $A$, random except in the 1,1 position. The circle is the unit circle in $\mathbb{C}$. The eigenvalues are approximately uniformly distributed in the unit disk except for the outlier $\lambda \approx 1.4852$.

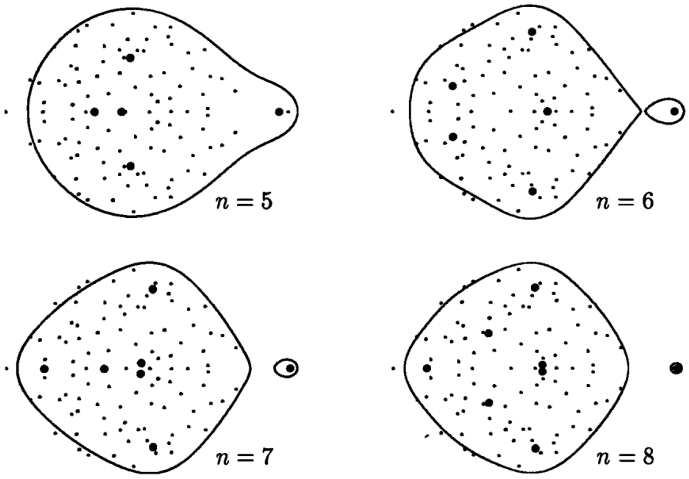



Figure 34.2: Arnoldi lemniscates at steps $n = 5, 6, 7, 8$ for the same matrix $A$. The small dots are the eigenvalues of $A$, and the large dots are the eigenvalues of $H_n$, i.e., the Ritz values. One component of the Arnoldi lemniscate first "swallows" the outlier eigenvalue, and in subsequent iterations it then shrinks to a point at a geometric rate.

## 34.7 Geometric Convergence

Under certain circumstances, the convergence of some of the Arnoldi eigenvalue estimates to eigenvalues of $A$ is geometric. These matters are incompletely understood at present, and we shall not cite theoretical results here. Instead, we shall just take a look at the convergence in the numerical example above, and give a partial explanation.

We still consider the example above and show the following convergence result:

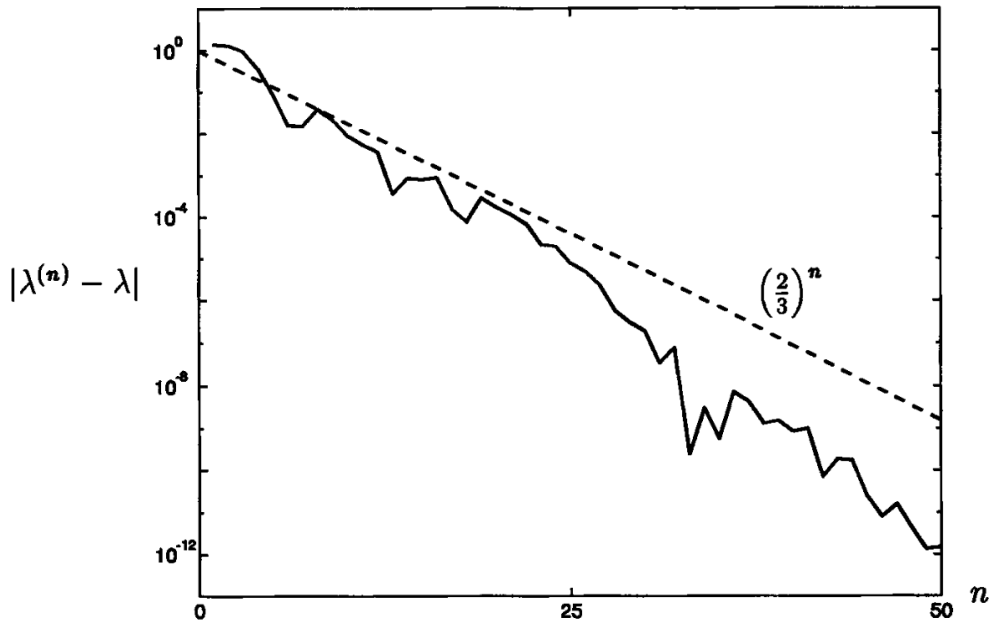



Figure 34.3: Convergence of the rightmost Arnoldi eigenvalue estimate.

Firstly, we can see that

$$|\lambda^{(n)} - \lambda| \approx \left(\frac{2}{3}\right)^n .$$

An explanation is that we consider the polynomial $p(z) = z^{n-1}(z - \tilde{\lambda})$ where $\tilde{\lambda}$ is some number close to $\lambda$. At each of the eigenvalues of $A$ in the unit disk, $|p(z)|$ is of order 1 or smaller. At $z = \lambda$, however, it has magnitude

$$|p(\lambda)| \approx \left(\frac{3}{2}\right)^n |\tilde{\lambda} - \lambda|$$

(this would be an equality if $\lambda$ were exactly equal to 3/2 ). When $n$ is large, $(3/2)^n$ is huge. For this number also to be of order 1, $|\tilde{\lambda} - \lambda|$ must be small enough to balance it, that is, of order $(2/3)^n$.

Another feature is apparent in Figure 34.4. After the initial few dozen steps, the convergence begins to accelerate, a phenomenon common in Krylov subspace iterations. What is happening here is that the iteration is beginning to resolve some of the other outer eigenvalues of $A$, near the unit circle. If the dimension of $A$ had been $m = 300$, then the cloud of eigenvalues would have filled the unit disk sufficiently densely that no such acceleration would have been visible in the fifty iterative steps.

# CHAPTER 35

# GMRES

In the last lecture we showed how the Arnoldi process can be used to find eigenvalues. Here we show that it can also be used to solve systems of equations $Ax = b$. The standard algorithm of this kind is known as GMRES, which stands for "generalized minimal residuals."

## 35.1 Residual Minimization in $\mathcal{K}_n$

Given $A \in \mathbb{C}^{m\times m}, b \in \mathbb{C}^m$ and $\mathcal{K}_n = \langle b, Ab, \dots, A^{n-1}b\rangle$. Assume $A$ nonsingular and the goal is to solve $Ax = b$. We denote $x_* = A^{-1}b$.

The idea of GMRES is to approximate $x_*$ by $\arg\min_{x\in\mathcal{K}_n} \|b - Ax_n\|$. Let $K_n$ be the $m \times n$ Krylov matrix, then the column space of $AK_n$ is $A\mathcal{K}_n$. Our problem is to

$$\arg\min_{c\in\mathbb{C}^n} \|AK_nc - b\|.$$

Once $c$ is found, we set $x_n = K_nc$. This could be done by the QR of $AK_n$.

The procedure just described is numerically unstable, and it constructs a factor $R$ that is not needed. Here is what is actually done. We use the Arnoldi iteration (Algorithm 33.1) to construct a sequence of Krylov matrices $Q_n$ whose columns $q_1, q_2, \dots$ span the successive Krylov subspaces $\mathcal{K}_n$. Then, we write $x_n = Q_ny$. Our problem is to minimize

$$\|AQ_ny - b\| = \|Q_{n+1}\tilde{H}_ny - b\| = \|\tilde{H}_ny - Q_{n+1}^*b\| = \|\tilde{H}_ny - \|b\|e_1\|.$$

At step $n$ of GMRES, we solve this problem for $y$ and then set $x_n = Q_ny$.

## 35.2 Mechanics of GMRES

**Algorithm 35.1:** GMRES

$q_1 = \frac{b}{\|b\|}$;
**for** $n = 1, 2, 3, \dots$ **do**
  $\langle$ step $n$ of Arnoldi iteration, Algorithm 33.1$\rangle$;
  Find $y$ to minimize $\|\tilde{H}_ny - \|b\|e_1\|$;
  $x_n = Q_ny$.

Note that the "Find y" step can be solved via QR factorization. Rather than construct $QR$ factorizations of the successive matrice $\tilde{H}_1, \tilde{H}_2, \dots$ independently, one can use an updating process to get the QR factorization of $\tilde{H}_n$ from that of $\tilde{H}_{n-1}$. All that is required is a single Givens rotation and $O(n)$ work.

## 35.3 GMRES and Polynomial Approximation

The GMRES iteration also solves an approximation problem, the only difference is the space of polynomials

$$P_n = \{ \text{ polynomials } p \text{ of degree } \leqslant n \text{ with } p(0) = 1\}.$$

Note that for the GMRES, the update is

$$x_n = q_n(A)b,$$

where $q$ is a polynomial of degree $n-1$. The corresponding residual $r_n = b - Ax_n$ is $r_n = (I - Aq_n(A))b = p_n(A)b$. Thus we have shown that GMRES solves the following approximation problem.

**Example 35.1** (GMRES approximation problem).

$$\arg \min_{p_n \in P_n} \|p_n(A)b\|.$$

Like the Arnoldi iteration, GMRES satisfies certain invariance properties.

**Theorem 35.2** (Properties of GMRES).
Let the GMRES iteration be applied to a matrix $A \in \mathbb{C}^{m\times m}$ as described above.

Scale-invariance. If $A$ is changed to $\sigma A$ for some $\sigma \in \mathbb{C}$, and $b$ is changed to $\sigma b$, the residuals $\{r_n\}$ change to $\{\sigma r_n\}$.

Invariance under unitary similarity transformations. If $A$ is changed to $UAU^*$ for some unitary matrix $U$, and $b$ is changed to $Ub$, the residuals $\{r_n\}$ change to $\{U^* r_n\}$.

GMRES is not invariant under translation, since the normalization $p(0) = 1$ involves the translation-dependent point 0 . On the contrary, its behavior depends strongly on the position of the origin-loosely speaking, on the condition number of $A$.

## 35.4 Convergence of GMRES

The convergence of GMRES depends on the preconditioner. Mathematically, the problem is to investigate what properties of $A$ determine the size of $\|r_n\|$.

We begin with two observations:

- $\|r_{n+1}\| \leqslant \|r_n\|$, since $\mathcal{K}_n \subset \mathcal{K}_{n+1}$.
- After at most $m$ steps, the process must converge. $\|r_m\| = 0$. For generic data $A, b$, this is true because $\mathcal{K}_m = \mathbb{C}^m$ and in special case, if $b$ lies in $\mathcal{K}_n$ for some $n < m$, converge will occur earlier.

To obtain more useful information about convergence, we must turn to the polynomial approximation. We know that $\|r_n\| = \|p_n(A)b\| \leqslant \|p_n(A)\|\|b\|$. Hence,

$$\frac{\|r_n\|}{\|b\|} \leqslant \inf_{p_n \in P_n} \|p_n(A)\|.$$

Then, we only need to bound $\inf_{p_n \in P_n} \|p_n(A)\|$.

## 35.5 Polynomials Small on the Spectrum

Given $A$ and $n$, how small can $\|p_n(A)\|$ be? Given a polynomial $p$ and a set $S$, we define the scalar $\|p\|_S$ by

$$\|p\|_S = \sup_{z\in S} |p(z)|.$$

Suppose $A = V\Lambda V^{-1}$ for some nonsingular matrix $V$ and diagonal matrix $\Lambda$. THen,

$$\|p(A)\| \leqslant \|V\|\|p(\Lambda)\|\|V^{-1}\| = \kappa(V)\|p\|_{\Lambda(A)}.$$

**Theorem 35.3.**
At step $n$ of the GMRES iteration, the residual $r_n$ satisfies

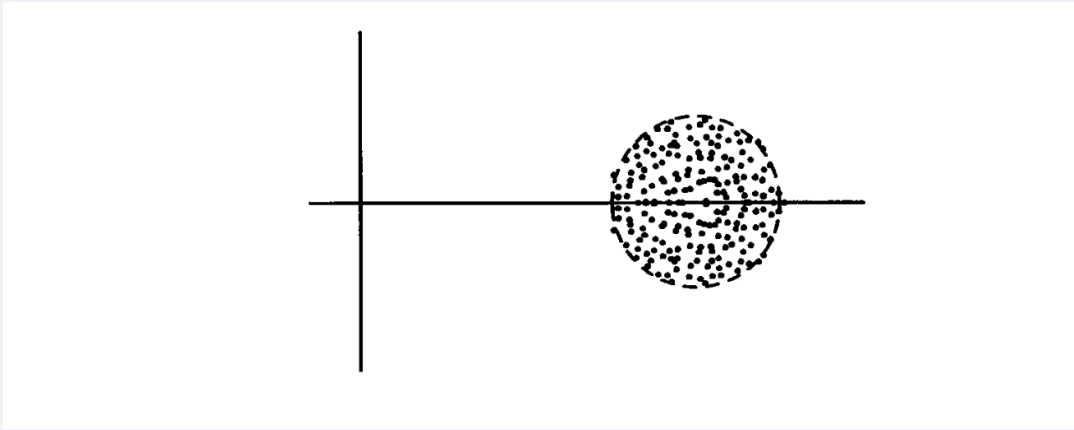
$$\frac{\|r_n\|}{\|b\|} \leqslant \inf_{p_n \in P_n} \|p_n(A)\| \leqslant \kappa(V) \inf_{p_n \in P_n} \|p_n\|_{\Lambda(A)} ,$$

where $\Lambda(A)$ is the set of eigenvalues of $A, V$ is a nonsingular matrix of eigenvectors (assuming $A$ is diagonalizable), and $\|p_n\|_{\Lambda(A)}$ is defined above.

This theorem can be summarized in words as follows. If $A$ is not too far from normal in the sense that $\kappa(V)$ is not too large, and if properly normalized degree $n$ polynomials can be found whose size on the spectrum $\Lambda(A)$ decreases quickly with $n$, then GMRES converges quickly.

**Example 35.4** (A good eg for GMRES).
Here is a numerical example. Let $A$ be a $200 \times 200$ matrix whose entries are independent samples from the real normal distribution of mean 2 and standard deviation $0.5/\sqrt{200}$. In MAtLAB,

$$\mathrm{m} = 200; \mathrm{A} = 2 * \mathrm{eye(m)} + 0.5 * \mathrm{randn(m)/\,sqrt(m)}. \tag{35.1}$$



This figure shows the eigenvalues of $A$, a set of points roughly uniformly distributed in the disk of radius $1/2$ centered at $z = 2$ .

Fig 34.1 shows the convergence curve for the GMRES iteration applied to the problem $Ax = b$, where $b = (1, 1, \ldots, 1)^*$. The convergence in this case is extraordinarily steady at a rate approximately $4^{-n}$. The reason for this is not hard to spot. Since the spectrum of $A$ approximately fills the disk indicated, $\|p(A)\|$ is approximately minimized by the choice $p(z) = (1 - z/2)^n$. Since $I - A/2$ is a random matrix scaled so that its spectrum approximately fills the disk of radius $1/4$ about 0 , we have $\|p(A)\| = \|(I - A/2)^n\| \approx 4^{-n}$. This matrix $A$ is well-conditioned, with condition number $\kappa(A) \approx 2.03$. The deviation from normality is modest, with $\kappa(V) \approx 141$.

Figure 34.1 illustrates the convergence of matrix iterations under favorable circumstances-when the matrix $A$ is well behaved (which often means a good preconditioner has been applied). We see that six-digit accuracy is achieved after ten iterations, at a cost of approximately $10 \times 2\ \mathrm{m}^2 = 8.0 \times 10^5$flops, since the work is dominated by the matrix-vector multiplication at each step. Solving the same system by Gaussian elimination would require $\frac{2}{3}m^3 \approx 5.3\times\ 10^6$ flops. This improvement by a factor close to 7 was achieved even though $A$ has no sparsity to take advantage of and even though the dimension $m = 200$ is not high. For the same example with $m = 2000$, GMRES would beat Gaussian elimination by a factor more like 70 . For a $2000 \times 2000$ matrix with similar spectral properties but 90% or 99% sparsity, the factor would improve to on the order of 700 or 7000 , respectively, and the storage required by GMRES would also diminish dramatically.

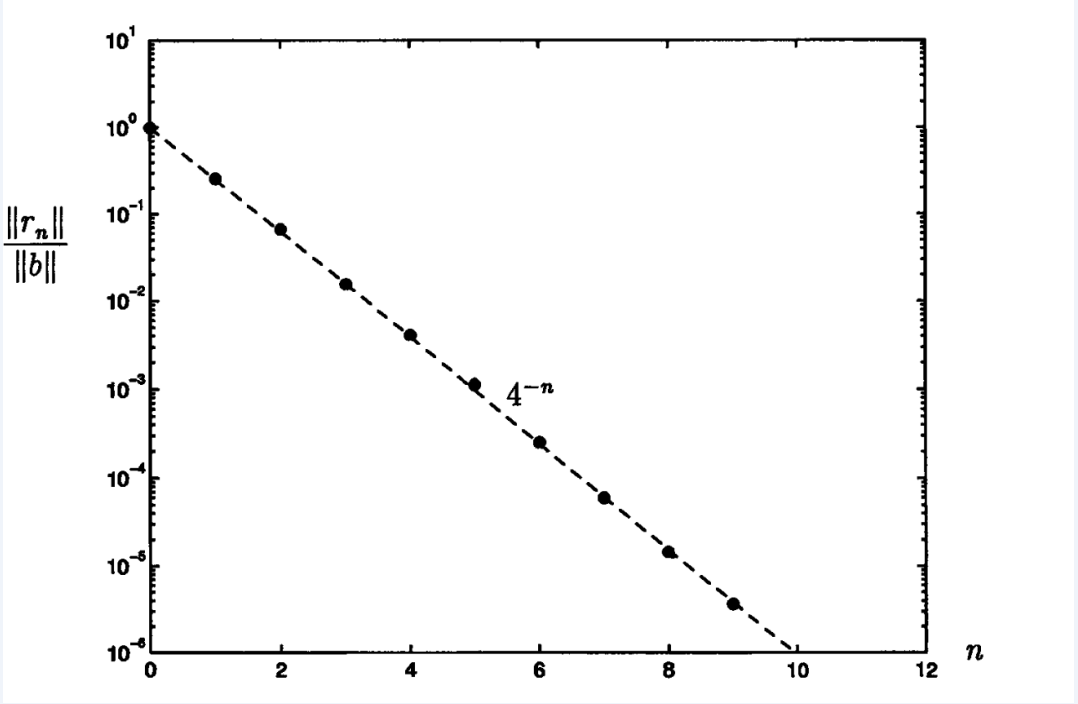
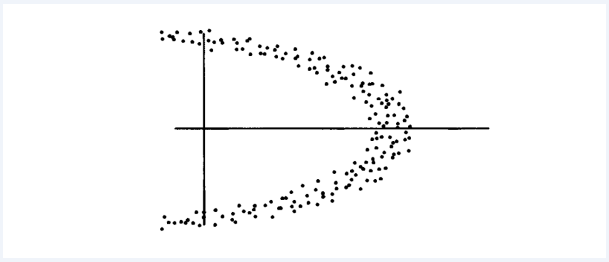
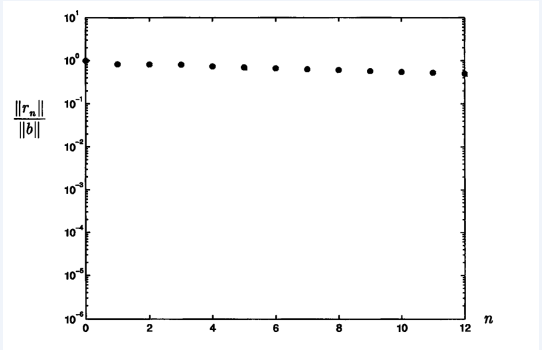



Figure 35.1: GMRES convergence curve for the same matrix A. This rapid, steady convergence is illustrative of Krylov subspace iterations under ideal circumstances, when $A$ is a well-behaved (or well-preconditioned) matrix.

**Example 35.5** (A bad eg for GMRES).
If the eigenvalues of a matrix "surround the origin," on the other hand, such rapid convergence cannot be expected. Figures 35.4-35.5 present an example. The matrix is now $A' = A + D$, where $A$ is the matrix of (35.1) and $D$ is the diagonal matrix with complex entries

$$d_k = (-2 + 2\sin\theta_k) + i\cos\theta_k, \quad \theta_k = \frac{k\pi}{m-1}, 0 \leqslant k \leqslant m-1.$$






As is evident in Figure, the eigenvalues now lie in a semicircular cloud that bends around the origin. The convergence rate is much worse than before, making the iterative computation no better than Gaussian elimination for this problem. The condition numbers are now $\kappa(A) \approx 4.32$ and $\kappa(V) \approx 54.0$, so the deterioration in convergence cannot be explained by conditioning alone; it is the locations of the eigenvalues, not their magnitudes (or those of the singular values) that are causing the trouble. If the arc extended much further around the spectrum, the convergence would worsen further.

# CHAPTER 36

# THE LANCZOS ITERATION

In the last three lectures we considered Krylov subspace iterations for nonhermitian matrix problems. We shall return to nonhermitian problems in Chapter 38, for there is more to this subject than Arnoldi and GMRES. But first, in this and the following two lectures, we specialize to the hermitian case, where a major simplification takes place.

## 36.1 Three-Term Recurrence

The Lanczos is the Arnoldi iteration specialized to the case where $A$ is hermitian. Here, we assume $A$ is real and symmetric.

For Arnoldi process in this special case, it follows from (33.6) that the Ritz matrix $H_n$ is symmetric. Hence, its eigenvalues the Ritz values or Lanczos estimates are also real.

Since $H_n$ is both symmetric and Hessenberg, it's tridiagonal. Hence, the $(n+1)$-term recurrence at step $n$ is replaced by

$$Aq_n = h_{n-1,n}q_{n-1} + h_{nn}q_n + h_{n+1,n}q_{n+1}.$$

This is just a three-term recurrence. This means the inner loop of Arnoldi iteration (algorithm 33.1), the limits 1 to $n$ can be replaced by $n-1$ to $n$. Therefore, each step of the Lanczos iteration is much cheaper than the corresponding step of the Arnoldi iteration.

The key equation for the tridiagonal property of $H_n$ is that

$$H_n = Q_n^* A Q_n.$$

In fact, this is equal to say:

$$h_{ij} = q_i^\top A q_j. \tag{36.1}$$

This implies that $h_{ij} = 0$ for $i > j+1$, since $Aq_j \in \langle q_1, q_2, \ldots, q_{j+1}\rangle$ and the Krylov vectors are orthogonal. Taking the transpose gives

$$h_{ij} = q_j^\top A^\top q_i = q_j c^\top A q_i.$$

Hence, $h_{ij} = 0$ for $j > i+1$. This simple argument leading to a three-term recurrence relation applies to arbitrary self-adjoint operator, not just to matrices.

## 36.2 The Lanczos Iteration

Let $\alpha_n = h_{nn}$ and $\beta_n = h_{n+1,n} = h_{n,n+1}$. Then, $H_n$ becomes

$$T_n = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix}.$$

**Algorithm 36.1:** Lanczos Iteration

$\beta_0 = 0, q_0 = 0, b$ random $, q_1 = \frac{b}{\|b\|}$;
**for** $n = 1, 2, 3, \ldots$ **do**
  $v = Aq_n$;
  $\alpha_n = q_n^\top v$;
  $v = v - \beta_{n-1} q_{n-1} - \alpha_n q_n$;
  $\beta_n = \|v\|$;
  $q_{n+1} = \frac{v}{\beta_n}$;

Each step consists of a matrix-vector multiplication, an inner product, and a couple of vector operations. If $A$ has enough sparsity or other structure that matrix-vector products can be computed cheaply, then such an iteration can be applied without too much difficulty to problems of dimensions in the tens or hundreds of thousands.

**Theorem 36.1** (Properties of Lanczos iteration).
The matrices $Q_n$ of vectors $q_n$ generated by the Lanczos iteration are reduced $QR$ factors of the Krylov matrix $K_n$,

$$K_n = Q_n R_n \tag{36.2}$$

The tridiagonal matrices $T_n$ are the corresponding projections

$$T_n = Q_n^* A Q_n \tag{36.3}$$

and the successive iterates are related by the formula

$$AQ_n = Q_{n+1} \tilde{T}_n, \tag{36.4}$$

which we can write in the form of a three-term recurrence at step $n$,

$$Aq_n = \beta_{n-1} q_{n-1} + \alpha_n q_n + \beta_n q_{n+1}. \tag{36.5}$$

As long as the Lanczos iteration does not break down (i.e., $K_n$ is of full rank n), the characteristic polynomial of $T_n$ is the unique polynomial $p^n \in P^n$ that solves the Arnoldi/Lanczos approximation problem (34.3), i.e., that achieves

$$\|p^n(A)b\| = \text{ minimum }. \tag{36.6}$$

## 36.3 Lanczos and Electric Charge Distributions

In practice, the Lanczos iteration is used to compute eigenvalues of large symmetric matrices just as the Arnoldi iteration is used for nonsymmetric matrices. At each step $n$, or at occasional steps, the eigenvalues of the growing tridiagonal matrix $T_n$ are determined by standard methods. Often some of these numbers are observed to converge geometrically to certain limits, which can then be expected to be eigenvalues of $A$.

As with the Arnoldi iteration, it is the outlying eigenvalues of $A$ that are most often obtained first.

- If the eigenvalues of $A$ are more evenly spaced than Chebyshev points, then the Lanczos iteration will tend to find outliers.

Here is what this statement means. Suppose the $m$ eigenvalues $\{\lambda_j\}$ of $A$ are spread reasonably densely around an interval on the real axis. Since the Lanczos iteration is scale- and translation-invariant (Theorem 36.1), we can assume without loss of generality that this interval is $[-1,1]$. The $m$ **Chebyshev points** in $[-1,1]$ are defined by the formula

$$x_j = \cos\theta_j, \quad \theta_j = \frac{\left(j-\frac{1}{2}\right)\pi}{m}, \quad 1 \leqslant j \leqslant m.$$

The exact definition is not important; what matters is that these points cluster quadratically near the endpoints, with the spacing between points $O(\,\mathrm{m}^{-1})$ in the interior and $O(\,\mathrm{m}^{-2})$ near $\pm 1$ . The rule of thumb asserts that if the eigenvalues $\{\lambda_j\}$ of $A$ are more evenly distributed than this-less clustered at the endpoints - then the Ritz values computed by a Lanczos iteration will tend to converge to the outlying eigenvalues first. In particular, an approximately uniform eigenvalue distribution will produce rapid convergence towards outliers. Conversely, if the eigenvalues of $A$ are more than quadratically clustered at the endpoints-a situation not so common in practice-then we can expect convergence to some of the "inliers."

These observations can be given a physical interpretation. Consider $m$ point charges free to move about the interval $[-1,1]$. Assume that the repulsive force between charges located at $x_j$ and $x_k$ is proportional to $|x_j - x_k|^{-1}$. (For electric charges in 3D the force would be $|x_j - x_k|^{-2}$, but this becomes $|x_j - x_k|^{-1}$ in 2D, where we can view each point as the intersection of an infinite line in 3D with the plane.) Let these charges distribute themselves in a minimal-energy equilibrium in $[-1,1]$. Then this minimal-energy distribution and the Chebyshev distribution are approximately the same, and in the limit $m \to \infty$, they both converge to a limiting continuous charge density distribution proportional to $(1-x^2)^{-1/2}$

Think of the eigenvalues of $A$ as point charges. If they are distributed approximately in a minimal-energy configuration in an interval, then the Lanczos iteration will be useless; there will be little convergence before step $n = m$. If the distribution is very different from this, however, then there is likely to be rapid convergence to some eigenvalues, namely, the eigenvalues in regions where there is "too little charge" in the sense that if the points were free to move, more would tend to cluster here. The rule of thumb can now be restated:

- The Lanczos iteration tends to converge to eigenvalues in regions of "too little charge" for an equilibrium distribution.

## 36.4 Example

Let $A \in \mathbb{R}^{203\times 203}$ and

$$A = \mathrm{diag}(0, .01, .02, \ldots, 1.99, 2, 2.5, 3.0).$$

Hence the spectrum of $A$ consists of a dense collection of eigenvalues throughout $[0,2]$ together with two outliers, 2.5 and 3.0. We carry out a Lanczos iteration beginning with a random starting vector $q_1$.

Figure 35.1 shows the Ritz values and the associated Lanczos polynomial at step $n = 9$. Seven of the Ritz values lie in [0,2], and the polynomial is uniformly small on that interval; the beginnings of a tendency for the Ritz values to cluster near the endpoints can be detected. The other two Ritz values lie near the eigenvalues at 2.5 and 3.0. The leading three Ritz values are

$$1.93, 2.48, 2.999962.$$

Evidently we have little accuracy in the lower eigenvalues but five-digit accuracy in the leading one. A plot like this gives an idea of why outliers tend to be estimated accurately. The graph of $p(x)$ is so steep for $x \approx 3$ that if $p(3)$ is to be small, there must be a root of $p$ very close to 3 . This steepness

of the graph is related to the presence of "too little charge" near this point. If the charges were free to move about $[0,3]$ to minimize energy, more points would cluster near $x = 3$ and $p(x)$ would not be so steep there.

At step 20 the leading three Ritz values are

$$1.9906,\ 2.499999999987,\quad 3.00000000000000.$$

Now we have about fifteen digits of accuracy in the leading eigenvalue and twelve digits in the second. A plot of $p(x)$ would be correspondingly steep near the points 2.5 and 3.0. Note that convergence to the third eigenvalue is also beginning to occur, a reflection of the fact that the eigenvalues in $[0,2]$ are distributed evenly rather than in a Chebyshev distribution.

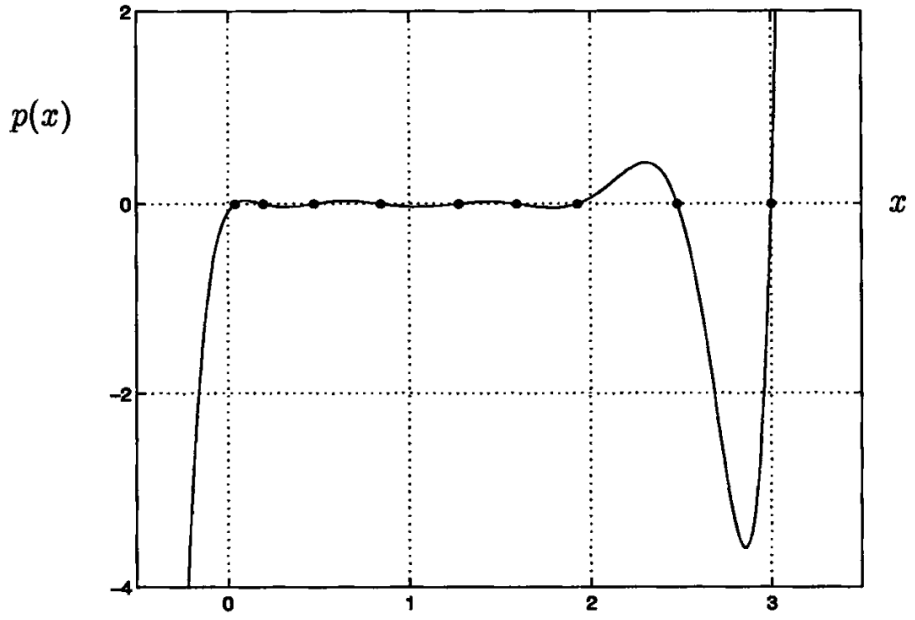



Figure 36.1: Plot of the Lanczos polynomial at step 9 of the Lanczos iteration for the matrix $A$. The roots are the Ritz values or "Lanczos eigenvalue estimates." The polynomial is small throughout $[0,2] \cup \{2.5\} \cup \{3.0\}$. To achieve this, it must place one root near 2.5 and another very near 3.0.

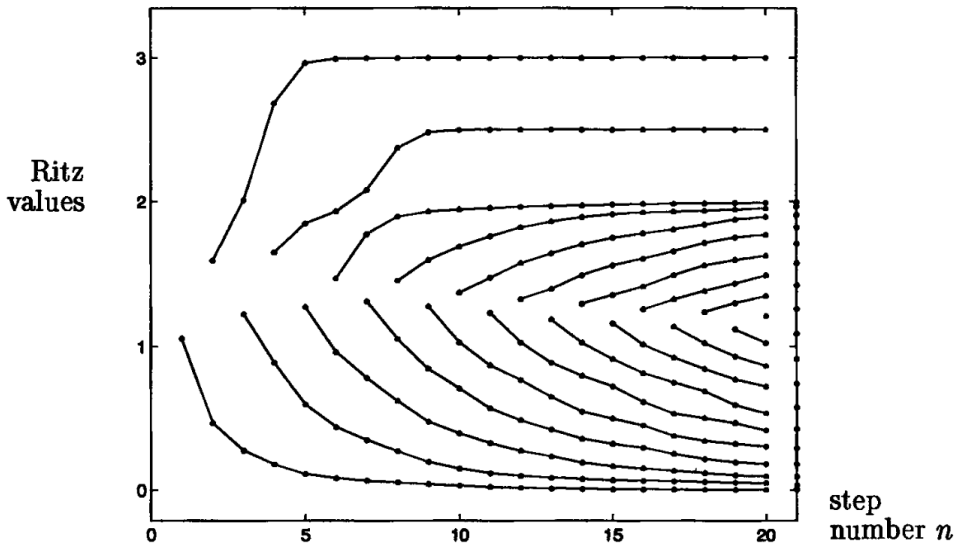



Figure 36.2: Ritz values for the first 20 steps of the Lanczos iteration applied to the same matrix. The convergence to the eigenvalues 2.5 and 3.0 is geometric. Little useful convergence to individual eigenvalues occurs in the $[0,2]$ part of the spectrum. Instead, the Ritz values in $[0,2]$ approximate Chebyshev points in that interval, marked by dots on the right-hand boundary.

An "aerial view" of the convergence process appears in Figure 36.2 , which shows the Ritz values for all steps from $n = 1$ to $n = 20$. Each vertical slice of this plot corresponds to the Ritz values at one iteration; the lines connecting the dots help the eye follow what is going on but have no precise meaning. The plot shows pronounced convergence to the leading eigenvalue after about $n = 5$ and to the next one around $n = 10$. In the interval $[0, 2]$ containing the other eigenvalues, they show a density of Ritz values approximately proportional to $(1 - x^2)^{-1/2}$, with very clear bunching at endpoints.

## 36.5 Rounding Errors and "Ghost" Eigenvalues

Rounding errors have a complex effect on the Lanczos iteration and, indeed, on all iterations of numerical linear algebra based on three-term recurrence relations. The source of the difficulty is easily identified. In an iteration based on an $n$-term recurrence relation, such as Arnoldi or GMRES, the vectors $q_1, q_2, q_3, \ldots$ are forced to be orthogonal by explicit Gram-Schmidt operations. Three-term recurrences like Lanczos and conjugate gradients, however, depend upon orthogonality of the vectors $\{q_j\}$ to arise "automatically" from a mathematical identity. In practice, such identities are not accurately preserved in the presence of rounding errors, and after a number of iterations, orthogonality is lost.

The loss of orthogonality in practical Lanczos iterations sounds wholly bad, but the situation is more subtle than that. As it happens, loss of orthogonality is connected closely with the convergence of Ritz values to eigenvalues of $A$. A great deal is known about this subject, though not as much as one might like; we shall not give details.

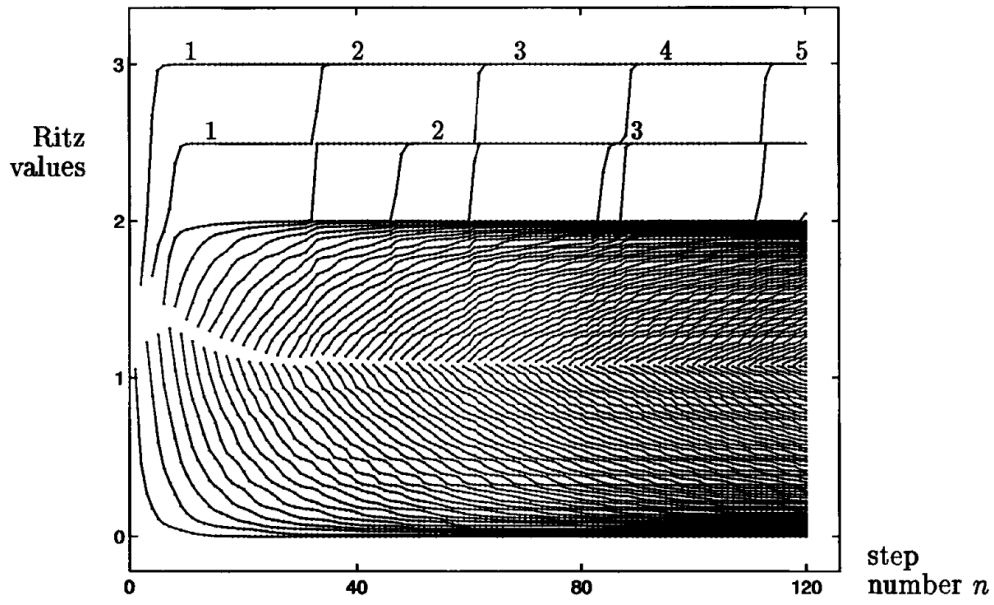



Figure 36.3: Continuation to 120 steps of the Lanczos iteration. The numbers indicate multiplicities of the Ritz values. Note the appearance of four "ghost" copies of the eigenvalue 3.0 and two "ghost" copies of the eigenvalue 2.5 .

Because of complexities like these, no straightforward theorem is known to the effect that the Lanczos or conjugate gradient iterations is stable in the sense defined in this book. Nonetheless, these iterations are extraordinarily useful in practice. Figure 35.3 gives an idea of the way in which instability is often manifested in practice without preventing the iteration from being useful. The figure is a repetition of Figure 36.2 , but for 120 instead of 20 steps of the iteration. Everything looks as expected until around step 30, when a second copy of the eigenvalue 3.0 appears among the Ritz values. A third copy appears around step 60, a fourth copy around step 90, and so on. Meanwhile, additional copies of the eigenvalue 2.5 also appear around step 40 and 80 and (just beginning to be visible) 120 . These extra Ritz values are known as "ghost" eigenvalues, and they have nothing to do with the actual multiplicities of the corresponding eigenvalues of $A$.

A rigorous analysis of the phenomenon of ghost eigenvalues is complicated. Intuitive explanations, however, are not hard to devise.

- One idea is that in the presence of rounding errors, one should think of each eigenvalue of $A$ not as a point but as a small interval of size roughly $O\left(\epsilon_{\text{machine}} \|A\|\right)$; ghost eigenvalues arise from the need for $p(z)$ to be small not just at the exact eigenvalues but throughout these small intervals.
- Another, rather different explanation is that convergence of a Ritz value to an eigenvalue of $A$ annihilates the corresponding eigenvector component in the vector being operated upon; but in the presence of rounding errors, random noise must be expected to excite that component slightly again. After sufficiently many iterations, this previously annihilated component will have been amplified enough that another Ritz value is needed to annihilate it again-and then again, and again.

Both of these explanations capture some of the truth about the behavior of the Lanczos iteration in floating point arithmetic. The second one has perhaps more quantitative accuracy.

# CHAPTER 37

# FROM LANCZOS TO GAUSS QUADRATURE

If discrete vectors become continuous functions on $[-1, 1]$, and the matrix $A$ is taken to be the operator of pointwise multiplication by $x$, then the Lanczos iteration becomes the standard procedure for constructing orthogonal polynomials via a three-term recurrence relation. From here it is a short step to Gauss quadrature formulas, whose nodes and weights can be computed by solving a symmetric tridiagonal matrix eigenvalue problem.

## 37.1 Orthogonal Polynomials

Here we consider $L^2[-1, 1]$ instead of $\mathbb{R}^m$. $L^2[-1, 1]$ is a Hilbert space with inner product

$$(u, v) = \int_{-1}^{1} u(x)v(x)\,dx, \tag{37.1}$$

$\forall u, v \in L^2[-1, 1]$. Now we define $A$ as the linear operator by

$$(Au)(x) = xu(x)$$

$\forall u \in L^2[-1, 1]$. This operator is analogous to a diagonal matrix. In particular, $A$ is symmetric, and because of the symmetry, the Arnoldi process specializes to the Lanczos process. Furthermore, we assume the initial function $b(x)$ is a nonzero constant. The corresponding normalized initial function $q_1(x)$ will be $q_1(x) = \frac{1}{\sqrt{2}}$.

**Algorithm 37.1:** Construction of Orthogonal Polynomials

$\beta_0 = 0, q_0(x) = 0, q_1(x) = \frac{1}{\sqrt{2}}$;
**for** $n = 1, 2, 3, \ldots$ **do**
  $v(x) = xq_n(x)$;
  $\alpha_n = (q_n, v)$;
  $v(x) = v(x) - \beta_{n-1}q_{n-1}(x) - \alpha_n q_n(x)$;
  $\beta_n = \|v\|$;
  $q_{n+1}(x) = v(x)/\beta_n$;

Note that the polynomials $q_1, q_2, q_3, \ldots$ are of degrees $0, 1, 2, \ldots$ and thus the polynomial of degree $n$ in this sequence is $q_{n+1}$.

Algo 37.1 will construct the sequence of Legendre polynomials. We labeled Algo **??** "Construction of orthogonal polynomials" rather than "Construction of Legendre polynomials" because it is, in fact, more general. If (37.1) is modified by the inclusion of a nonconstant positive weight function $w(x)$ in the integrand, then one obtains other families of orthogonal polynomials such as Chebyshev polynomials and Jacobi polynomials. All of the developments of this lecture apply to these more general families, but we shall not give details.

## 37.2 Jacobi Matrices

Right now we have $K_n = (1, x, \ldots, x^{n-1})$ and $Q_n = (q_1(x), q_2(x), \ldots, q_n(x))$. The tridiagonal matrices $T_n$ described in the preivous chapter are parituclar important. There are still $n \times n$ discrete matrices, related to $Q_n$ and $A$ by (36.4) and (36.3). Their entries are given by the analogue of (36.1),

$$t_{ij} = (q_i(x), xq_j(x)).$$

In the context of orthogonal polynomials, the matrices $\{T_n\}$ are known as **Jacobi matrices.** Now the three-term recurrence (36.5) takes the form

$$xq_n(x) = \beta_{n-1}q_{n-1}(x) + \alpha_n q_n(x) + \beta_n q_{n+1}(x).$$

The statement of algorithm 37.1 is perhaps misleading as written. It would appear that nontrivial computations are involved at each step of this algorithm: the evaluation of the inner product $(q_n, v)$ and norm $\|v\|$ that define $\alpha_n$ and $\beta_n$. If (37.1) contained an arbitrary weight function $w(x)$, these computations would indeed be nontrivial. However, for the particular choice $w(x) = 1$ associated with Legendre polynomials, and also for various choices associated with other classical families of polynomials, the entries $\{\alpha_n\}$ and $\{\beta_n\}$ are known analytically. We have

$$\alpha_n = 0, \beta_n = \frac{1}{2}(1 - (2n)^{-2})^{-\frac{1}{2}} \tag{37.2}$$

for Legendre polynomials. With the use of these formulas algorithm 37.1 becomes a trivial mechanical procedure.

## 37.3 The Characteristic Polynomial

In this context of orthogonal polynomials, the Arnoldi approximation problem becomes the following problem:

**Example 37.1** (Orthogonal Polynomials Approximation Problem).

$$\arg \min_{p^n \in P^n} \|p^n(x)\|.$$

According to Thm 36.1, the solution is the characteristic polynomial of the matrix $T_n$.

We should notice that for any $p \in P^n$, $p(x) = Cq_{n+1}(x) + Q_n y$, where $C$ is a constant-the inverse of the leading coefficient of $q_{n+1}(x)$. Note that $\{q_n(x)\}$ are orthogonal, we have $\|p\| = (C^2 + \|y\|^2)^{\frac{1}{2}}$. It's obvious that the minimum is achieved by setting $y = 0$. Hence, $p^n(x) = Cq_{n+1}(x)$.

**Theorem 37.2.**
Let $\{q_n(x)\}$ be the sequence of orthogonal polynomials generated by algorithm 37.1, let $\{T_n\}$ be the associated sequence of tridiagonal Jacobi matrices, and let $p^n$ be the characteristic polynomial of $T_n$. Then for $n = 0, 1, 2, \ldots$,

$$p^n(x) = C_n q_{n+1}(x), \tag{37.3}$$

where $C_n$ is a constant. In particular, the zeros of $q_{n+1}(x)$ are the eigenvalues of $T_n$. These $n$ zeros are distinct and lie in the open interval $(-1, 1)$.

This theorem is of great computational importance. To determine the zeros of the Legendre polynomials, all one has to do is compute the eigenvalues of the associated Jacobi matrices, whose entries are given in closed form by (37.2). As we have seen in previous lectures, the eigenvalue problem for an $n \times n$ symmetric tridiagonal matrix is well-conditioned and can be solved very quickly, requiring only $O(n^2)$ flops. By contrast, computing the zeros of the Legendre polynomials directly, starting from the coefficients of the polynomials rather than the Jacobi matrices, is inefficient and numerically unstable.

## 37.4 Quadrature Formulas

There is a reason why the zeros of the Legendre polynomials are of computational interest: they are the nodes of the Gauss—Legendre quadrature formulas.

Let us briefly review the idea of numerical quadrature. Given $f(x)$ on $[-1,1]$, we want to compute

$$I(f) = \int_{-1}^{1} f(x)\,dx.$$

It's natural to consider

$$I_n(f) = \sum_{j=1}^{n} w_j f(x_j) \tag{37.4}$$

defined by a set of $n$ nodes or abscissas $x_j \in [-1,1]$ and corresponding weights $w_j$, chosen independently of $f$. This is an $n$-point **quadrature formula**. Various forms of such formulas, often coupled with adaptive error estimation, interval subdivision, and order control, are the basis of most numerical integration carried out on computers today.

Any set of nodes is a candidate for a quadrature formula. The following result is a consequence of the nonsingularity of Vandermonde matrices.

**Theorem 37.3.**
Let the nodes $\{x_j\}$ be an arbitrary set of $n$ distinct points in $[-1,1]$. Then there is a unique choice of weights $\{w_j\}$ with the property that the quadrature formula (37.4) has order of accuracy at least $n-1$ in the sense that it is exact if $f(x)$ is any polynomial of degree $\leqslant n-1$.

If the nodes $\{x_j\}$ are taken equally spaced from -1 to 1 , the quadrature formula provided by this theorem is known as a **Newton-Cotes** formula, the $n$-point generalization of the familiar trapezoid and Simpson rules. NewtonCotes formulas have the order of accuracy guaranteed by this theorem but no higher. These formulas are simple and useful, especially for lower values of $n$. For larger $n$, their weights $w_j$ have oscillating signs and huge amplitudes, of order $2^n$, causing numerical instability.

## 37.5 Gauss Quadrature

The idea of Gauss quadrature is to pick not just the weights $\{w_j\}$ but also the nodes $\{x_j\}$ optimally, so as to raise the order of accuracy of (37.4) as high as possible. As it happens, there is a unique choice of nodes and weights that achieves this, and the resulting formula has order $2n-1$. This is a dramatic improvement over order $n-1$, a doubling of the number of digits of accuracy typically attainable for smooth functions and a fixed number of function evaluations. Moreover, the weights $w_j$ are all positive, making these formulas stable even for high $n$.

The **Gauss-Legendre quadrature formula** is defined as the quadrature formula (37.4) provided by Thm 36.1 whose nodes $x_1, \ldots, x_n$ are the zeros of $q_{n+1}(x)$.

**Theorem 37.4** (Accuracy of Gauss-Legendre quadrature).
Theorem 37.3. The n-point Gauss-Legendre quadrature formula has order of accuracy exactly $2n-1$, and no quadrature formula (37.4) has order of accuracy higher than this.

*Proof.*
Given any set of distinct points $\{x_j\}$, let $f(x)$ be the polynomial $\Pi_{j=1}^{n}(x-x_j)^2$ of degree $2n$. Then $I(f) > 0$, but $I_n(f) = 0$ since $f(x_j) = 0$ for each node $x_j$. Thus the quadrature formula is not exact for polynomials of degree $2n$.

On the other hand, suppose $f(x)$ is any polynomial of degree $\leqslant 2n-1$, and take $\{x_j\}$ to be the Gauss quadrature nodes, the zeros of $\{q_{n+1}(x)\}$. The function $f(x)$ can be factored in the form

$$f(x) = g(x)q_{n+1}(x) + r(x),$$

where $g(x)$ is a polynomial of degree $\leqslant n-1$ and $r(x)$, the remainder term, is also a polynomial of degree $\leqslant n-1$. (In fact, $r(x)$ is the degree $n-1$ polynomial interpolant to $f$ in the points $\{x_j\}$.) Now since $q_{n+1}(x)$ is orthogonal to all polynomials of lower degree, we have $I\left(gq_{n+1}\right)=0$. At the same time, since $g\left(x_j\right) q_{n+1}\left(x_j\right)=0$ for each node $x_j$, we have $I_n\left(gq_{n+1}\right)=0$. Since $I$ and $I_n$ are linear operators, these identities imply $I(f)=I(r)$ and $I_n(f)=I_n(r)$. But since $r(x)$ is of degree $\leqslant n-1$, we have $I(r)=I_n(r)$ by Theorem 37.3 and combining these results gives $I(f)=I_n(f)$, as claimed. □

## 37.6 Gauss Quadrature via Jacobi Matrices

We have gone from the Lanczos iteration to Legendre polynomials and from Legendre polynomials to Gauss quadrature. We have even provided a fast and stable algorithm for determining the nodes of Gauss quadrature formulas: just set up the Jacobi matrices $\{T_n\}$ and compute their eigenvalues.

One final observation will finish the story. Not only the nodes but also the weights of Gauss quadrature formulas can be obtained from the eigenvalue problem for $T_n$.The $j$ th weight turns out to be simply $w_j=2(v_j)_1^2$, that is, twice the square of the first component of the $j$th eigenvector of $T_n$. This can be extended to general weight function $w(x)$.

**Theorem 37.5** (Gauss-Legendre quadrature formula).
Let $T_n$ be the $n \times n$ Jacobi matrix defined by Algorithm 37.1 with entries $\beta_1, \ldots, \beta_{n-1}$ given by (37.2) . Let $T_n=VDV^T$ be an orthogonal diagonalization of $T_n$ with $V=\left[v_1|\cdots|v_n\right]$ and $D=\operatorname{diag}\left(\lambda_1, \ldots, \lambda_n\right)$. Then the nodes and weights of the Gauss-Legendre quadrature formula are given by

$$x_j=\lambda_j, \quad w_j=2\left(v_j\right)_1^2, \quad j=1, \ldots, n.$$

## 37.7 Example

As an illustration of the power of Gauss quadrature for integrating smooth functions, suppose we wish to evaluate the integral

$$I\left(e^x\right)=\int_{-1}^1 e^x dx=2.35040239.$$

Taking $n=4$, we find that the Jacobi matrix for four-point Gauss-Legendre quadrature is

$$T_4=\begin{bmatrix} 0 & 0.577350269 & & \\ 0.577350269 & 0 & 0.516397779 & \\ & 0.516397779 & 0 & 0.507092553 \\ & & 0.507092553 & 0 \end{bmatrix}.$$

The eigenvalues of this matrix give the nodes

$$x_1=-x_4=0.861136312, \quad x_2=-x_3=0.339981044,$$

and the first components of the eigenvectors give the corresponding weights

$$w_1=w_4=0.347854845, \quad w_2=w_3=0.652145155.$$

Evaluating the sum (37.4) gives

$$I_n\left(e^x\right)=2.35040209,$$

which agrees with the exact result to about seven digits. The four-point Newton-Cotes formula, by contrast, gives $I_n\left(e^x\right) \approx 2.3556$, accurate to only about three digits.

# CHAPTER 38

# CONJUGATE GRADIENTS

The conjugate gradient iteration is the "original" Krylov subspace iteration, the most famous of these methods and one of the mainstays of scientific computing. Discovered by Hestenes and Stiefel in 1952, it solves symmetric positive definite systems of equations amazingly quickly if the eigenvalues are well distributed.

## 38.1 Minimizing the 2-Norm of the Residual

We have learned about GMRES. At step $n$, $x_*$ is approximated by the vector $x_n \in \mathcal{K}_n$ that minimizes $\|r_n\|_2$, where $r_n = b - Ax$. Since $A$ is symmetric, faster algorithms are available based on three-term instead of $(n+1)$-term recurrences at step $n$. One of these goes by the names of conjugate residuals or MINRES("minimal residuals").

## 38.2 Minimizing the $A$-Norm of the Error

Assume $A$ is also positive definite. Then we can define $A$ norm by

$$\|x\|_A = \sqrt{x^\top A x}.$$

Now we care about the vector $e_n = x_* x_n$. The conjugate gradient iteration can be described as follows. **It is a system of recurrence formulas that generates the unique sequence of iterates $\{x_n \in \mathcal{K}_n\}$ with the property that at step $n$, $\|e_n\|_A$ is minimized.**

## 38.3 The Conjugate Gradient Iteration

**Algorithm 38.1:** Conjugate Gradient (CG) Iteration

$x_0 = 0, r_0 = b, p_0 = r_0$;
**for** $n = 1, 2, 3, \ldots$ **do**
  $\alpha_n = (r_{n-1}^\top r_{n-1})/(p_{n-1}^\top A P_{n-1})$ ; // step length
  $x_n = x_{n-1} + \alpha_n p_{n-1}$ ; // approximate solution
  $r_n = r_{n-1} - \alpha_n A p_{n-1}$ ; // residual
  $\beta_n = (r_n^\top r_n)/(r_{n-1}^\top r_{n-1})$ ; // improvement this step
  $p_n = r_n + \beta_n p_{n-1}$ ; // search direction

Before analyzing the mathematical properties of these formulas, let us examine them operationally. The algorithm is extraordinarily simple. The only complication-which we shall not address—is the choice of a convergence criterion.

From the five lines that define the algorithm, the following properties can be deduced. Like all the theorems in this book that do not explicitly mention rounding errors, this one assumes that the computation is performed in exact arithmetic. If there are rounding errors, these properties fail, and it becomes a subtle matter to explain the still very impressive performance of CG.

**Theorem 38.1** (Properties of CG).
Let the CG iteration (algorithm 37.1) be applied to a symmetric positive definite matrix problem $Ax = b$. As long as the iteration has not yet converged, the algorithm proceeds without divisions by zero, and we have the following identities of subspaces:

$$\begin{aligned} \mathcal{K}_n &= \langle x_1, x_2, \dots, x_n \rangle = \langle p_0, p_1, \dots, p_{n-1} \rangle \\ &= \langle r_0, r_1, \dots r_{n-1} \rangle = \langle b, Ab, \dots, A^{n-1}b \rangle. \end{aligned} \tag{38.1}$$

Moreover, the residuals are orthogonal,

$$r_n^\top r_j = 0, \quad \forall j < n, \tag{38.2}$$

and the search direction are "A-conjugate,"

$$p_n^\top A p_j = 0, \quad \forall j < n. \tag{38.3}$$

*Proof sketch.*
We can prove all these by induction. □

## 38.4 Optimality of CG

**Theorem 38.2** (Optimality of CG).
Let the CG iteration be applied to a symmetric positive definite matrix problem $Ax = b$. If the iteration has not already converged (i.e., $r_{n-1} \neq 0$), then $x_n$ is the unique point in $\mathcal{K}_n$ that minimizes $\|e_n\|_A$. The convergence is monotonic,

$$\|e_n\|_A \leqslant \|e_{n-1}\|_A, \tag{38.4}$$

and $e_n = 0$ is achieved for some $n \leqslant m$.

*Proof.*
From Thm 38.1, we know that $x_n$ belongs to $\mathcal{K}_n$. Proof. To show that it is the unique point in $\mathcal{K}_n$ that minimizes $\|e\|_A$, consider an arbitrary point $x = x_n - \Delta x \in \mathcal{K}_n$, with error $e = x_* - x = e_n + \Delta x$. We calculate

$$\begin{aligned} \|e\|_A^2 &= (e_n + \Delta x)^T A (e_n + \Delta x) \\ &= e_n^T A e_n + (\Delta x)^T A (\Delta x) + 2 e_n^T A (\Delta x). \end{aligned}$$

The final term in this equation is $2r_n^T(\Delta x)$, an inner product of $r_n$ with a vector in $\mathcal{K}_n$, and by Thm 38.1 any such inner product is zero. This is the crucial orthogonality property that makes the CG iteration so powerful. It implies that we have

$$\|e\|_A^2 = e_n^T A e_n + (\Delta x)^T A (\Delta x)$$

Only the second of these terms depends on $\Delta x$, and since $A$ is positive definite, that term is $\geqslant 0$, attaining the value 0 if and only if $\Delta x = 0$, i.e., $x_n = x$. Thus $\|e\|_A$ is minimal if and only if $x_n = x$, as claimed.

The remaining statements of the theorem now follow readily. The monotonicity property (38.4) is a consequence of the inclusion $\mathcal{K}_n \subseteq \mathcal{K}_{n+1}$, and since $\mathcal{K}_n$ is a subset of $\mathbb{R}^m$ of dimension $n$ as long as convergence has not yet been achieved, convergence must be achieved in at most $m$ steps. □

The guarantee that the CG iteration converges in at most $m$ steps is void in floating point arithmetic. For arbitrary matrices $A$ on a real computer, no decisive reduction in $\|e_n\|_A$ will necessarily be observed at all when $n = m$. In practice, however, CG is used not for arbitrary matrices but for matrices whose spectra, perhaps thanks to preconditioning, are well-enough behaved that convergence to a desired accuracy is achieved for $n \ll m$.

## 38.5 CG as an Optimization Algorithm

We have just shown that the CG iteration has a certain optimality property: it minimizes $\|e_n\|_A$ at step $n$ over all vectors $x \in \mathcal{K}_n$. In fact, as foreshadowed already by the use of such terms as "step length" and "search direction," this iteration can be interpreted as an algorithm of a standard form for minimizing a nonlinear function of $x \in \mathbb{R}^m$. At the heart of the iteration is the formula

$$x_n = x_{n-1} + \alpha_n p_{n-1}.$$

This is a familiar equation in optimization, in which a current approximation $x_{n-1}$ is updated to a new approximation $x_n$ by moving a distance $\alpha_n$ (the step length) in the direction $p_{n-1}$ (the search direction). By a succession of such steps, the CG iteration attempts to find a minimum of a nonlinear function.

In fact, this function is

$$\varphi(x) = \frac{1}{2} x^\top A x - x^\top b. \tag{38.5}$$

In fact, a short computation now reveals

$$\begin{aligned} \|e_n\|_A^2 &= e_n^T A e_n = (x_* - x_n)^T A (x_* - x_n) \\ &= x_n^T A x_n - 2 x_n^T A x_* + x_*^T A x_* \\ &= x_n^T A x_n - 2 x_n^T b + x_*^T b = 2\varphi(x_n) + \text{ constant}. \end{aligned}$$

Thus $\varphi(x)$ is the same as $\|e\|_A^2$ except for a factor of 2 and the constant $x_*^\top b$. This function must achieve its minimum uniquely at $x = x_*$.

At each step, an iterate $x_n = x_{n-1} + \alpha_n p_{n-1}$ is computed that minimize $\varphi(x)$ over all $x$ in the one-dimensional space $x_{n-1} + \langle p_{n-1} \rangle$. What makes the CG iteration remarkable is the choice of the search direction $p_{n-1}$, which has the special property that minimizing $\varphi(x)$ over $x_n + \langle p_{n-1} \rangle$ actually minimizes it over all of $\mathcal{K}_n$.

There is a close analogy between the CG iteration for solving $Ax = b$ and the Lanczos iteration for finding eigenvalues. The eigenvalues of $A$, as discussed in Chapter 26, are the stationary values for $x \in \mathbb{R}^m$ of the Rayleigh quotient, $r(x) = \left(x^T A x\right) / \left(x^T x\right)$. The eigenvalue estimates (Ritz values) associated with step $n$ of the Lanczos iteration are the stationary values of the same function $r(x)$ if $x$ is restricted to the Krylov subspace $\mathcal{K}_n$. This is a perfect parallel of what we have shown in the last two pages, that the solution $x_*$ of $Ax = b$ is the minimal point in $\mathbb{R}^m$ of the scalar function $\varphi(x)$, and the CG iterate $x_n$ is the minimal point of the same function $\varphi(x)$ if $x$ is restricted to $\mathcal{K}_n$.

## 38.6 CG and Polynomial Approximation

For the CG, the approximation problem is:

**Example 38.3** (CG Approximation problem).

$$\arg\min_{p_n \in P_n} \|p_n(A) e_0\|_A.$$

Here $e_0 = x_* - x_0 = x_*$ and $P_n = \{p \in \mathbb{R}[x] \,|\, \deg(p) \leqslant n, p(0) = 1\}$.

We have the following convergence theorem.

**Theorem 38.4.**
If the CG iteration has not already converged before step $n$ (i.e., $r_{n-1} \neq 0$ ), then the CG approximation problem has a unique solution $p_n \in P_n$, and the iterate $x_n$ has error $e_n = p_n(A)e_0$ for this same polynomial $p_n$. Consequently we have

$$\frac{\|e_n\|_A}{\|e_0\|_A} = \inf_{p \in P_n} \frac{\|p(A)e_0\|_A}{\|e_0\|_A} \leqslant \inf_{p \in P_n} \max_{\lambda \in \Lambda(A)} |p(\lambda)|$$

where $\Lambda(A)$ denotes the spectrum of $A$.

## 38.7 Rate of Convergence

Thm 38.4 establishes that the rate of convergence of the CG iteration is determined by the location of the spectrum of $A$. A good spectrum is one on which polynomials $p_n \in P_n$ can be very small, with size decreasing rapidly with $n$. Roughly speaking, this may happen for either or both of two reasons: the eigenvalues may be grouped in small clusters, or they may lie well separated in a relative sense from the origin.

First, we suppose the eigenvalues are perfectly clustered.

**Corollary 38.5.**
If $A$ has only $n$ distinct eigenvalues, then the CG iteration converges in at most $n$ steps.

At the other extreme, suppose we know nothing about any clustering of the eigenvalues but only that heir distances from the origin vary by at most a factor $\kappa \geqslant 1$. $\kappa = \lambda_{\max}/\lambda_{\min}$.

**Corollary 38.6.**
Let the CG iteration be applied to a symmetric positive definite matrix problem $Ax = b$, where $A$ has 2-norm condition number $\kappa$. Then the A-norms of the errors satisfy

$$\frac{\|e_n\|_A}{\|e_0\|_A} \leqslant 2/\left[\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^n + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^{-n}\right] \leqslant 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^n.$$

*Proof sketch.*
Consider the shifted Chebyshev polynomial

$$p(x) = \frac{T_n\left(\gamma - \frac{2x}{\lambda_{\max}-\lambda_{\min}}\right)}{T_n(\gamma)},$$

where $\gamma$ takes the special value $\gamma = \frac{\lambda_{\max}+\lambda_{\min}}{\lambda_{\max}-\lambda_{\min}} = \frac{\kappa+1}{\kappa-1}$. For $x \in [\lambda_{\min}, \lambda_{\max}]$, the argument of $T_n$ in the numerator of $p(x)$ lies in $[-1, 1]$, which means the magnitude of that numerator is $\leqslant 1$. Therefore, we only need to show

$$T_n(\gamma) = T_n\left(\frac{\kappa+1}{\kappa-1}\right) = \frac{1}{2}\left[\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^n + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^{-n}\right].$$

Since we know if $x = \frac{1}{2}(z + z^{-1}), T_n(x) = \frac{1}{2}(z^n + z^{-n})$, we can show this with a change of variable. □

Corollary 38.6 is the most famous result about convergence of the CG iteration. Since

$$\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} \sim 1 - \frac{2}{\sqrt{\kappa}}$$

as $\kappa \to \infty$, it implies that if $\kappa$ is large but not too large, convergence to a specified tolerance can be expected in $O(\sqrt{\kappa})$ iterations. One must remember that this is only an upper bound. Convergence may be faster for special righthand sides (not so common) or if the spectrum is clustered (more common).

## 38.8 Example

For an example of the convergence of CG, consider a $500 \times 500$ sparse matrix $A$ constructed as follows. First we put 1 at each diagonal position and a random number from the uniform distribution on $[-1, 1]$ at each off-diagonal position (maintaining the symmetry $A = A^T$ ). Then we replace each off-diagonal entry with $|a_{ij}| > \tau$ by zero, where $\tau$ is a parameter. For $\tau$ close to zero, the result is a well-conditioned positive definite matrix whose density of nonzero entries is approximately $\tau$. As $\tau$ increases, both the condition number and the sparsity deteriorate.

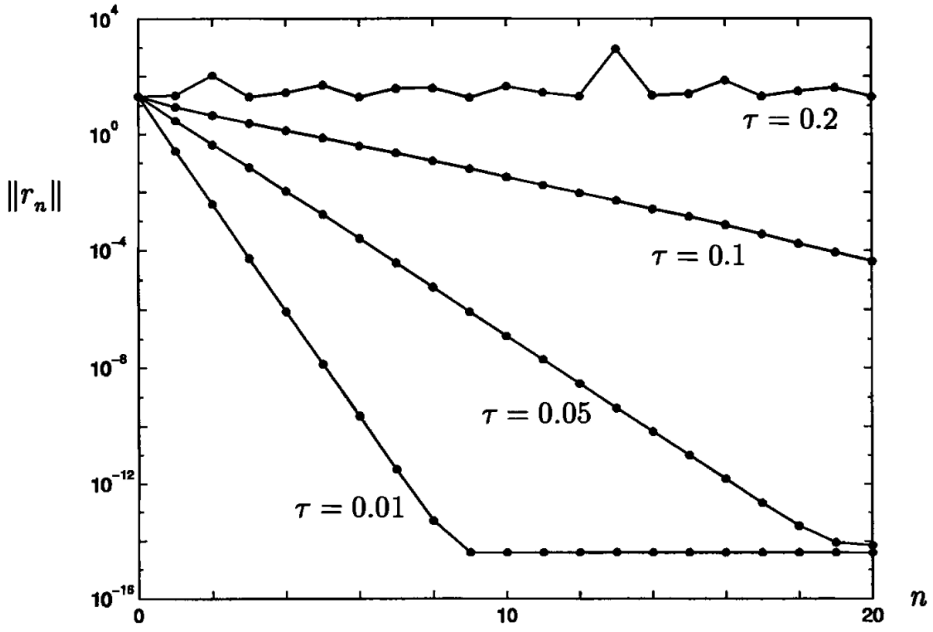



Figure 38.1: CG convergence curves for the $500 \times 500$ sparse matrices $A$ described in the text. For $\tau = 0.01$, the system is solved about 700 times faster by *CG* than by Cholesky factorization. For $\tau = 0.2$, the matrix is not positive definite and there is no convergence.

Note that for $\tau = 0.2$, with 50,834 nonzeros, there is no convergence at all. The lowest eigenvalue is now negative, so $A$ is no longer positive definite and the use of the CG iteration is inappropriate. (In fact, the CG iteration often succeeds with indefinite matrices, but in this case the matrix is not only indefinite but ill-conditioned.)

When $\tau = 0.01$, the operation count of $6 \times 10^4$ flops beats Cholesky factorization (23.4) by a factor of about 700. Unfortunately, not every matrix arising in practice has such a well-behaved spectrum, even after the best efforts to find a good preconditioner.

# CHAPTER 39

# BIORTHOGONALIZATION METHODS

Not all Krylov subspace iterations for nonsymmetric systems involve recurrences of growing length and growing cost. Methods based on three-term recurrences have also been devised, and they are the most powerful nonsymmetric iterations available today. The price to be paid, at least for some of the iterations in this category, is that one must work with two Krylov subspaces rather than one, generated by multiplications by $A^*$ as well as $A$.

## 39.1 Where We Stand

We have shown a table of Krylov subspace matrix iterations:

| | $Ax = b$ | $Ax = \lambda x$ |
|---|---|---|
| $A = A^*$ | CG | Lanczos |
| $A \neq A^*$ | GMRES<br>CGN<br>BCG et al. | Arnoldi |

Our discussions of three of these boxes are now complete, and as for the fourth, lower-left position, we have already discussed GMRES. In this lecture we turn to the final two lines of the table. We spend just a moment on CGN, a simple and easily analyzed algorithm, and then move to our main subject, the biorthogonalization methods represented by the entry "BCG et al."

## 39.2 CGN = CG Applied to the Normal Equations

Let $A \in \mathbb{C}^{m\times m}$ be nonsingular but not hermitian, and we want to solve $Ax = b$. One of the simplest methods is to apply the CG iteration to the normal equations:

$$A^*Ax = A^*b. \tag{39.1}$$

(The matrix $A^*A$ is not formed explicitly, which would require $m^3$ flops. Instead, each matrix-vector product $A^*Av$ is evaluated in two steps as $A^*(Av)$.) Now we can apply the CG method. This method goes by the name of CGN (also CGNR), which roughly stands for "CG applied to the normal equations."

Similarly, from Thm 38.1, we have

$$x_n \in \langle A^*b, (A^*A)A^*b, \ldots, (A^*A)^{n-1}A^*b \rangle.$$

From Thm 38.2, we know that the $A^*A$-norm of the error is minimized over this space at each step, and since $\|e_n\|^2_{A^*A} = e_n^* A^* A e_n = \|Ae_n\|_2^2 = \|r_n\|^2$, this is another way of saying that the 2-norm of the residual $r_n = b - Ax_n$ is minimized. Thus CGN, like GMRES, is a minimal residual method, but since their Krylov subspaces are different, these tw methods are by no means equivalent.

According to Theorem 38.4 the convergence of CGN is controlled by the eigenvalues of $A^*A$. These numbers are equal to the squares of the singular values of $A$. Thus the convergence of CGN is determined by the singular values of $A$, and in principle has nothing to do with the eigenvalues of $A$. The fact that squares are involved is unfortunate, however. If $A$ has condition number $\kappa$, then $A^*A$ has condition number $\kappa^2$, and the rate for CGN becomes

$$\frac{\|r_n\|_2}{\|r_0\|_2} \leqslant 2\left(\frac{\kappa - 1}{\kappa + 1}\right)^n.$$

For large $\kappa$, this is much more worse; it implies that $O(\kappa)$ iterations are required for convergence to a fixed accuracy, not $O(\sqrt{\kappa})$.

This "squaring of the condition number" has given the CGN iteration a poor reputation, which, on balance, may be deserved. Nevertheless, for some problems CGN vastly outperforms alternative methods, since their convergence depends on eigenvalues rather than singular values. All one needs is a matrix whose singular values are well behaved but whose eigenvalues are not, such as a well-conditioned matrix whose spectrum surrounds the origin in the complex plane. An extreme example is provided by the $m \times m$ circulant matrix of the form

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

The singular values of this matrix are all equal to 1, but the eigenvalues are the $m$th roots of unity. GMRES requires $m$ steps for convergence for a general right-hand side $b$, while CGN converges in one step.

Another virtue of the CGN iteration is that since it is based on the normal equations, it applies without modification to least squares problems, while $A$ is no longer square.

## 39.3 Tridiagonal Biorthogonalization

The Lanczos iteration is a process of tridiagonal orthogonalization. It $A$ is not hermitian, such a reduction is not possible in general. The Arnoldi iteration, a process of Hessenberg orthogonalization, doe the latter.

Biorthogonalization methods are based on the opposite choice. If we insist on a tridiagonal result but give up the use of unitary transformations, we have a process of **tridiagonal biorthogonalization**: $A = VTV^{-1}$, where $V$ is nonsingular but not unitary.

To begin to make this idea into an iterative algorithm, we must see what is involved for $n < m$. Assume $V$ is nonsingular and $A = VTV^{-1}$ with $T$ tridiagonal, and let $W = V^{-*}$. Let $v_j$ and $w_j$ denote the $j$th columns of $V$ and $W$, respectively. We can see that

$$W^*V = I \Rightarrow w_i^* v_j = \delta_{ij}.$$

Now we define $V_n = (v_1, \ldots, v_n), W_n = (w_1, \ldots, w_n) \in \mathbb{R}^{m \times n}$. Then,

$$W_n^* V_n = V_n^* W_n = I_n.$$

Recall that for the Lanczos iterations, we have

$$AQ_n = Q_{n+1}\tilde{T}_n, \quad T_n = Q_n^* A Q_n.$$

For the Arnoldi iteration, we have

$$AQ_n = Q_{n+1}\tilde{H}_n, \quad H_n = Q_n^* A Q_n.$$

Then for the biorthogonalization methods, we have

$$AV_n = V_{n+1}\tilde{T}_n, \tag{39.2}$$

$$A^*W_n = W_{n+1}\tilde{S}_n, \tag{39.3}$$

$$T_n = S_n^* = W_n^* A V_n. \tag{39.4}$$

Note that here $\tilde{T}_{n+1}$ and $\tilde{S}_{n+1}$ are tridiagonal matrices with dimension $(n+1)\times n$, and $T_n = S_n^*$ is the $n \times n$ matrix obtained by deleting the last row of $\tilde{T}_{n+1}$ or the last column of $\tilde{S}_{n+1}^*$. NOte that (39.2) can displayed as

$$A(v_1,\dots,v_n) = (v_1, v_2, \dots, v_{n+1}) \begin{bmatrix} \alpha_1 & \gamma_1 & & & \\ \beta_1 & \alpha_2 & \gamma_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \gamma_{n-1} \\ & & & \beta_{n-1} & \alpha_n \\ & & & & \beta_n \end{bmatrix},$$

which corresponds to the three-term recurrence relation

$$Av_n = \gamma_{n-1}v_{n-1} + \alpha_n v_n + \beta_n v_{n+1}. \tag{39.5}$$

Similarly, (39.3) takes the form

$$A^*(w_1,\dots,w_n) = (w_1, \dots, w_{n+1}) \begin{bmatrix} \bar{\alpha}_1 & \bar{\beta}_1 & & & \\ \bar{\gamma}_1 & \bar{\alpha}_2 & \bar{\beta}_2 & & \\ & \bar{\gamma}_2 & \bar{\alpha}_3 & \ddots & \\ & & \ddots & \ddots & \bar{\beta}_{n-1} \\ & & & \bar{\gamma}_{n-1} & \bar{\alpha}_n \\ & & & & \bar{\gamma}_n \end{bmatrix},$$

corresponding to

$$A^*w_n = \bar{\beta}_{n-1}w_{n-1} + \bar{\alpha}_n w_n + \bar{\gamma}_n w_{n+1}. \tag{39.6}$$

As usual with Krylov subspace iterations, these equations suggest an algorithm. Begin with vectors $v_1$ and $w_1$ that are arbitrary except for satisfying $v_1^* w_1 = 1$, and set $\beta_0 = \gamma_0 = 0$ and $v_0 = w_0 = 0$. Now, for each $n = 1, 2, \dots$, set $\alpha_n = w_n^* A v_n$, as follows from (39.4). The vectors $v_{n+1}$ and $w_{n+1}$ are then determined by (39.5) and (39.6) up to scalar factors. These factors may be chosen arbitrarily, subject to the normalization $w_{n+1}^* v_{n+1} = 1$, whereupon $\beta_{n+1}$ and $\gamma_{n+1}$ are determined by (39.5) and (39.6).

For a generic matrix, in exact arithmetic, the procedure will run to completion after $m$ steps, but for certain special matrices there may also be a breakdown of the process before this point. If $v_n = 0$ or $w_n = 0$ at some step, an invariant subspace of $A$ or $A^*$ has been found: the tridiagonal matrix $T$ is reducible. Alternatively, it may also happen that $v_n \neq 0$ and $w_n \neq 0$ but $w_n^* v_n = 0$. The possibility of this more serious kind of breakdown is present in most biorthogonalization methods. As in other areas of numerical analysis, the fact that exact breakdown is possible for certain problems implies that near-breakdown may occur for many other problems, with potentially adverse consequences in floating point arithmetic. Some methods for coping with these phenomena are mentioned at the end of this lecture.

## 39.4 BCG=Biconjugate Gradients

One way to use the biorthogonalization process just described is to compute eigenvalues: as $n \to \infty$, some eigenvalues of $T_n$ may converge rapidly to some eigenvalues of $A$. Another application, which we shall now briefly discuss, is the solution of nonsingular systems of equations $Ax = b$. The classic algorithm of this type is known as biconjugate gradients or *BCG*.

The principle of BCG is as follows. We take $v_1 = b$, so that the first Krylov subspace becomes $\mathcal{K}_n = \langle b, Ab, \dots, A^{n-1}b \rangle$. Recall that the principle of GMRES is to pick $x_n \in \mathcal{K}_n$ so that the orthogonality condition GMRES: $r_n \perp \langle Ab, A^2b, \dots, A^nb \rangle = A\mathcal{K}_n$ is satisfied, where $r_n = b - Ax_n$ is the residual corresponding to $x_n$. This choice has the effect of minimizing $\|r_n\|$, the 2-norm of the residual. The principle of the BCG algorithm is to pick $x_n$ in the same subspace, $x_n \in \mathcal{K}_n$, but to enforce the orthogonality condition GMRES: $r_n \perp \langle Ab, A^2b, \dots, A^nb \rangle = A\mathcal{K}_n$ is satisfied, where $r_n = b - Ax_n$ is the residual corresponding to $x_n$. This choice has the effect of minimizing $\|r_n\|$, the 2 -norm of the residual. The principle of the BCG algorithm is to pick $x_n$ in the same subspace, $x_n \in \mathcal{K}_n$, but to enforce the orthogonality condition

$$\text{BCG:} \quad r_n \perp \left\langle w_1, A^*w_1, \dots, (A^*)^{n-1} w_1 \right\rangle.$$

Here $w_1 \in \mathbb{C}^m$ is an arbitrary vector satisfying $w_1^* v_1 = 1$; in applications one sometimes takes $w_1 = v_1 / \|v_1\|_2$. Unlike GMRES, this choice does not minimize $\|r_n\|_2$, and it is not optimal from the point of view of minimizing the number of iterations. Its advantage is that it can be implemented with three-term recurrences rather than the $(n+1)$-term recurrences of GMRES.

Without giving details of the derivation, we now record the BCG algorithm in its standard form. What follows should be compared with Algorithm 38.1, the CG algorithm. The two are the same except that the sequence of search directions $\{p_n\}$ of CG has become two sequences $\{p_n\}$ and $\{q_n\}$, and the sequence of residuals $\{r_n\}$ of CG has become two sequences $\{r_n\}$ and $\{s_n\}$.

**Algorithm 39.1:** Biconjugate Gradient (BCG) iteration

$x_0 = 0, p_0 = r_0 = b, q_0 = s_0 =$ arbitrary ;
**for** $n = 1, 2, 3, \dots$ **do**
  $\alpha_n = s_{n-1}^* r_{n-1}) / (q_{n-1}^* A p_{n-1})$;
  $x_n = x_{n-1} + \alpha_n p_{n-1}$;
  $r_n = r_{n-1} - \alpha_n A p_{n-1}$;
  $s_n = s_{n-1} - \alpha_n A^* q_{n-1}$;
  $\beta_n = (s_n^* r_n) / (s_{n-1}^* r_{n-1})$;
  $p_n = r_n + \beta_n p_{n-1}$;
  $q_n = s_n + \beta_n q_{n-1}$;

## 39.5 Example

We consider the same example in chapter 37 except with one change: the sign of all the entries are randomized. This makes the matrix no longer hermitian, and it changes the dominat entries on the diagonal to 1 and $-1$ at random, rather than all 1. Now the eigenvalues are clustered around 1 and $-1$ instead of just 1.

In this figure we set $\tau = 0.01$. with $\tau = 0.01$. Considering first the GMRES curve, we note that the convergence is half as fast as in Chapter 37, with essentially no progress at each odd-numbered step, but steady progress at each even step. This odd-even effect is a result of the approximate $\pm 1$ symmetry of the matrix: a polynomial $p(z)$ of degree $2k+1$ with $p(0) = 1$ can be no smaller at 1 and -1 than a corresponding polynomial of degree $2k$. Turning now to the BCG curve, we see that the convergence is comparable in an overall sense, but it is no longer monotonic, showing spikes of magnitude as great as about $10^2$ at each oddnumbered step. The accuracy attained at the end has also suffered by more than a digit. All of these features are typical of BCG computations.

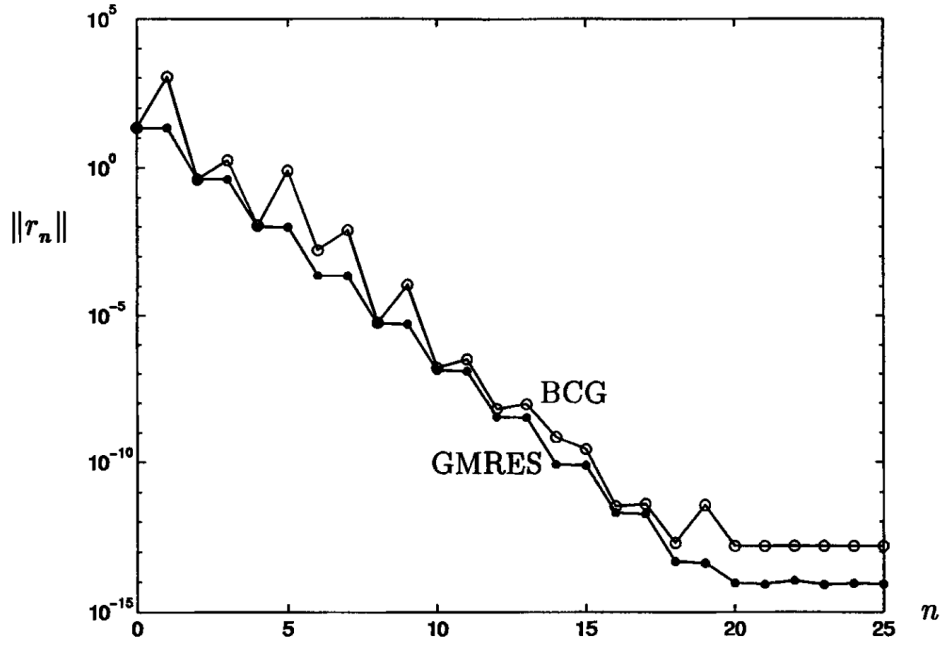



Figure 39.1: Comparison of GMRES and BCG for the $500 \times 500$ matrix labeled $\tau = 0.01$ in Figure 38.1 , but with the signs of the entries randomized.

The horizontal axis in Figure 39.2 is the step number $n$, which is not the same as the computational cost. At each step, GMRES requires one matrixvector multiplication involving $A$, whereas BCG requires multiplications involving both $A$ and $A^*$. For problems where matrix-vector multiplications dominate the work and enough storage is available, GMRES may consequently be twice as fast as BCG or faster.

## 39.6 QMR and Other Variants

BCG has one great advantage over GMRES: it involves three-term recurrences, enabling the work per step and the storage requirements to remain under control even when many steps are needed. On the other hand, it has two disadvantages.

- One is that in comparison to the monotonic and often rapid convergence of GMRES as a function of step number, its convergence is slower and often erratic, sometimes far more erratic than in Figure 39.2. Irregular convergence is unattractive, and it may have the consequence of reducing the ultimately attainable accuracy because of rounding errors.
- The other problem with BCG is that it requires multiplication by $A^*$ as well as $A$. Depending on how these products are implemented both mathematically and in terms of computer architecture, this may be anything from a minor additional burden to effectively impossible.

In response to these two problems, beginning in the 1980 s, more than a dozen variants of BCG have been proposed. Here are some of the best known of these; references are given in the Notes.

- Look-ahead Lanczos (Parlett, Taylor, and Liu, 1985)
- $CGS$ = conjugate gradients squared (Sonneveld, 1989)
- $QMR$ = quasi-minimal residuals (Freund and Nachtigal, 1991)
- $Bi$-CGSTAB = stabilized $BCG$ (van der Vorst, 1992)
- $TFQMR$ = transpose - free $QMR$( Freund, 1993)

The look-ahead Lanczos algorithm is based on the fact that when a breakdown is about to take place, it can be avoided by taking two or more steps of the iteration at once rather than a single step. The original idea of Parlett et al. has been developed extensively by later authors and is incorporated, for example, in the version of the QMR algorithm recommended by its authors. The phenomenon of breakdown can be shown to be equivalent to the phenomenon of square blocks of identical entries in the

table of Padé approximants to a function, and the look-ahead idea amounts to a method of stepping across such blocks in one step. In practice, of course, one does not just test for exact breakdowns; a notion of near-breakdown defined by appropriate tolerances is involved.

The CGS algorithm is based on the discovery that if two steps of the BCG are combined into one in a different manner, so that the algorithm is "squared," then multiplication by $A^*$ can be avoided. The result is a "transpose-free" method that sometimes converges up to twice as quickly as BCG, though the convergence is also twice as erratic.

The QMR algorithm is based on the observation that although three-term recurrences cannot be used to minimize $\|r_n\|$, they can be used to minimize a different, data-dependent norm that in practice is usually not so far from $\|r_n\|$. This may have a pronounced effect on the smoothness of convergence, significantly reducing the impact of rounding errors. The Bi-CGSTAB algorithm is another method that also significantly smooths the convergence rate of BCG, and TFQMR is a variant of QMR that combines its smooth convergence with the avoidance of the need for $A^*$.

Most recently, efforts have been directed at combining these three virtues of smoothed convergence curves, look-ahead to avoid breakdowns, and transposefree operation. So far, all three have not yet been combined fully satisfactorily in a single algorithm, but this research area is young.

# CHAPTER 40

# PRECONDITIONING

Most recently, efforts have been directed at combining these three virtues of smoothed convergence curves, look-ahead to avoid breakdowns, and transposefree operation. So far, all three have not yet been combined fully satisfactorily in a single algorithm, but this research area is young.

## 40.1 Preconditioners for $Ax = b$

Given $A \in \mathbb{R}^{m\times m}, b \in \mathbb{R}^m$, we want to solve

$$Ax = b.$$

For any nonsingular $M \in \mathbb{R}^{m\times m}$, the system

$$M^{-1}Ax = M^{-1}b$$

has the same solution. However this system depends on the properties of $M^{-1}A$ instead of just $A$. We call $M$ the preconditioned. We don't need an explicit construction of the inverse $M^{-1}$, but the solution of systems of equations of the form

$$My = c.$$

The idea is that we find an $M$ that we can solve $My = c$ quickly, and $M$ is very close to $A$ which makes $M^{-1}Ax = M^{-1}b$ easy.

What does it mean for $M$ to be "close enough to $A$?" Answering this question is the matter that has occupied our attention throughout this part of the book. If the eigenvalues of $M^{-1}A$ are close to 1 and $\|M^{-1}A - I\|_2$ is small, then any of the iterations we have discussed can be expected to converge quickly. However, preconditioners that do not satisfy such a strong condition may also perform well. For example, the eigenvalues of $M^{-1}A$ could be clustered about a number other than 1 , and there might be some outlier eigenvalues far from the others.

For most problems involving iterations other than CGN, fortunately, a simple rule of thumb is adequate. A preconditioner $M$ is good if $M^{-1}A$ is not too far from normal and its eigenvalues are clustered.

## 40.2 Left, Right, Hermitian Preconditioners

What we have described is a **left preconditioner**. Another idea is to transform $Ax = b$ into $AM^{-1}y = b$ with $x = M^{-1}y$, in which case $M$ is called a right preconditioner.

If $A$ is hermitian positive definite, then it is usal to preserve this property in preconditioning. Suppose $M$ is also PSD, with $M = CC^*$. Then the left preconditioner is equivalent to

$$[C^{-1}AC^{-*}]C^*x = C^{-1}b.$$

This can be solve by CG. Besides, $C^{-1}AC^{-*} \sim M^{-1}A$. Hence, it's enough to examine the eigenvalues of $M^{-1}A$.

## 40.3 Examples

Figure 40.1 presents an example of a preconditioned CG iteration for a symmetric positive definite matrix. The matrix $A$ isis the $1000 \times 1000$ symmetric matrix whose entries are all zero except for $a_{ij} = 0.5 + \sqrt{i}$ on the diagonal, $a_{ij} = 1$ on the sub- and superdiagonals, and $a_{ij} = 1$ on the 100 th sub- and superdiagonals, i.e., for $|i - j| = 100$. The righthand side is $b = (1, 1, \ldots, 1)^T$. As the figure shows, a straight CG iteration for this matrix converges slowly, achieving about five-digit residual reduction after forty iterations. Since the matrix is very sparse, this is an improvement over a direct method, but one would like to do better.

As it happens, we can do much better with a simple diagonal preconditioner. Take $M = \mathrm{diag}(A)$, the diagonal matrix with entries $m_{ii} = 0.5 + \sqrt{i}$. To preserve symmetry, set $C = \sqrt{M}$ and consider a new iteration. The figure shows that thirty steps of the iteration now give convergence to fifteen digits.

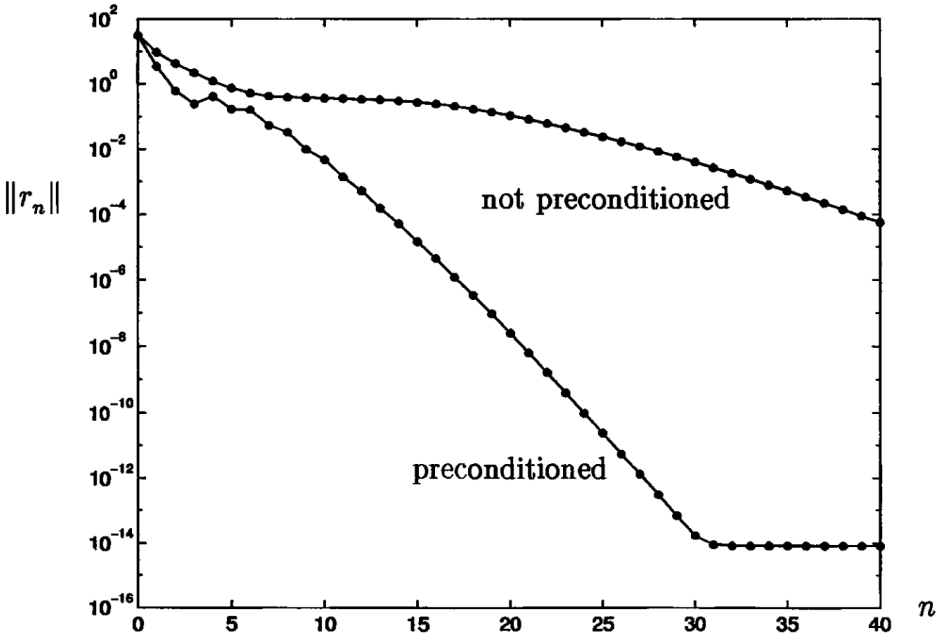



Figure 40.1: CG and preconditioned CG convergence curves for the $1000 \times 1000$ sparse matrix $A$ described in the text.

## 40.4 Survey of Preconditioners for $Ax = b$

The preconditioners used in practice are sometimes as simple as this one, but they are often far more complicated. Rather than consider one or two examples in detail, we shall take the opposite course and survey at a high level the wide range of preconditioning ideas that have been found useful over the years. Details can be found in the references listed in the Notes.

- **Diagonal Scaling or Jacobi.** Perhaps the most important preconditioner is the one just mentioned in the example: $M = \mathrm{diag}(A)$, provided that this matrix is nonsingular. For certain problems, this transformation alone is enough to make a slow iteration into a fast one. More generally, one may take $M = \mathrm{diag}(c)$ for a suitably chosen vector $c \in \mathbb{C}^m$. It is a hard mathematical problem to determine a vector $c$ such that $\kappa\,(M^{-1}A)$ is exactly minimized, but fortunately, nothing like the exact minimum is needed in practice, and in any case, as the rule of thumb above shows, there is more to preconditioning than minimizing the condition number.
- **Incomplete Cholesky or $LU$ factorization.** Another star preconditioner is the one that made the idea of preconditioning famous in the 1970s. Suppose $A$ is sparse, having just a few nonzeros per row. The difficulty with methods such as Gaussian elimination or Cholesky factorization is that these processes destroy zeros, so that if $A = R^*R$, for example, then the factor $R$ will usually not be very sparse. However, suppose a matrix $\tilde{R}$ is computed by Choleskylike formulas but

allowed to have nonzeros only in positions where $A$ has nonzeros, and we define $M = \tilde{R}^* \tilde{R}$. This incomplete Cholesky preconditioner may be highly effective for some problems; the acronym ICCG for incomplete Cholesky conjugate gradients is used. Similar $ILU$ or incomplete $LU$ preconditioners are useful in nonsymmetric cases. Numerous variants of the idea of incomplete factorization have been proposed and developed extensively.

- **Coarse-grid approximation.** A discretization of a partial differential or integral equation on a fine grid may lead to a huge system of equations. The analogous discretization on a coarser grid, however, may lead to a small system that is easy to solve. If a method can be found to transfer solutions on the coarse grid to the fine grid and back again, e.g. by interpolation, then a powerful preconditioner may be obtained of the following schematic form:

$$M = \langle \text{ transfer to fine grid } \rangle \circ A_{\text{coarse}} \circ \langle \text{ transfer to coarse grid } \rangle.$$

  Typically a preconditioner of this kind does a good job of handling the low-frequency components of the original problem, leaving the high frequencies to be treated by the Krylov subspace iteration. When this technique is iterated, resulting in a sequence of coarser and coarser grids, we obtain the idea of multigrid iteration.
- **Local approximation.** A coarse-grid approximation takes into account some of the larger-scale structure of a problem while ignoring some of the finer structure. A kind of a converse to this idea is relevant to problems $Ax = b$ where $A$ represents coupling between elements both near and far from one another. The elements may be physical objects such as particles, or they may be numerical objects such as the panels introduced in a boundary element discretization. In any case, it may be worth considering the operator $M$ analogous to $A$ but with the longer-range interactions omitted-a short-range approximation to $A$. In the simplest cases of this kind, $M$ may consist simply of a few of the diagonals of $A$ near the main diagonal, making this a generalization of the idea of a diagonal preconditioner.
- **Block preconditioners and domain decomposition.** Throughout numerical linear algebra, most algorithms expressed in terms of the scalar entries of a matrix have analogues involving block matrices. An example is that a diagonal or Jacobi preconditioner may be generalized to block-diagonal or block-Jacobi form. This is another kind of local approximation, in that local effects within certain components are considered while connections to other components are ignored. In the past decade ideas of this kind have been widely generalized in the field of domain decomposition, in which solvers for certain subdomains of a problem are composed in flexible ways to form preconditioners for the global problem. These methods combine mathematical power with natural parallelizability.
- **Low-order discretization.** Often a differential or integral equation is discretized by a higher-order method such as a fourth-order finite difference formula or a spectral method, bringing a gain in accuracy but making the discretization stencils bigger and the matrix less sparse. A lower-order approximation of the same problem, with its sparser matrix, may be an effective preconditioner. Thus, for example, one commonly encounters finite difference and finite element preconditioners for spectral discretizations.
- **Constant-coefficient or symmetric approximation.** Special techniques, like fast Poisson solvers, are available for certain partial differential equations with constant coefficients. For a problem with variable coefficients, a constantcoefficient approximation implemented by a fast solver may make a good preconditioner. Analogously, if a differential equation is not self-adjoint but is close in some sense to a self-adjoint equation that can be solved more easily, then the latter may sometimes serve as a preconditioner.
- **Splitting of a multi-term operator.** Many applications involve combinations of physical effects, such as the diffusion and convection that combine to make up the Navier-Stokes equations of fluid mechanics. The linear algebra result may be a matrix problem $Ax = b$ with $A = A_1 + A_2$ (or with more than two terms, of course), often embedded in a nonlinear iteration. If $A_1$ or $A_2$ is easily invertible, it may serve as a good preconditioner.
- **Dimensional splitting or ADI.** Another kind of splitting takes advantage of the fact that an operator such as the Laplacian in two or three dimensions is composed of analogous operators in

each of the dimensions separately. This idea may form the basis of a preconditioner, and in one form goes by the name of $ADI$ or alternating direction implicit methods.

- **One step of a classical iterative method.** In this book we have not discussed the "classical iterations" such as Jacobi, Gauss-Seidel, SOR, or SSOR, but one or more steps of these iterations-particularly Jacobi and SSOR-often serve excellently as preconditioners. This is also one of the key ideas behind multigrid methods.
- **Periodic or convolution approximation.** Throughout the mathematical sciences, boundary conditions are a source of analytical and computational difficulty. If only there were no boundary conditions, so that the problem were posed on a periodic domain! This idea can sometimes be the basis of a good preconditioner. In the simplest linear algebra context, it becomes the idea of preconditioning a problem involving a Toeplitz matrix $A$ (i.e., $a_{i,j} = a_{i-j}$ ) by a related circulant matrix $M$ $\left(m_{i,j} = m_{(i-j)(\bmod m)}\right)$, which can be inverted in $O(m \log m)$ operations by a fast Fourier transform. This is a particularly well studied example in which $M^{-1}A$ may be far from the identity in norm but have highly clustered eigenvalues.
- **Unstable direct method.** Certain numerical methods, such as Gaussian elimination without pivoting, deliver inaccurate answers because of instability. If the unstable method is fast, however, why not use it as a preconditioner? This is the "fly by wire" approach to numerical computation: solve the problem carelessly but quickly, and embed that solution in a robust control system. It is a powerful idea.
- **Polynomial preconditioners.** Finally, we mention a technique that is different from the others in that it is essentially $A^{-1}$ rather than $A$ itself that is approximated by the preconditioner. A polynomial preconditioner is a matrix polynomial $M^{-1} = p(A)$ with the property that $p(A)A$ has better properties for iteration than $A$ itself. For example, $p(A)$ might be obtained from the first few terms of the Neumann series $A^{-1} = I + (I - A) + (I - A)^2 + \cdots$, or from some other expression, often motivated by approximation theory in the complex plane. Implementation is easy, based on the same "black box" used for the Krylov subspace iteration itself, and the coefficients of the preconditioner may sometimes be determined adaptively.

## 40.5 Preconditioners for Eigenvalue Problems

Though the idea has been developed more recently and is not yet as famous, preconditioners can be effective for eigenvalue problems as well as systems of equations. Some of the best-known techniques in this area are polynomial acceleration, analogous to the polynomial preconditioning just described for systems of equations, shift-and-invert Arnoldi or the related rational Krylov iteration, which employ rational functions of $A$ instead of polynomials, and the Davidson and Jacobi-Davidson methods, based on a kind of diagonal preconditioner. For example, shift-and-invert and rational Krylov methods are based on the fact that if $r(z)$ is a rational function and $\{\lambda_j\}$ are the eigenvalues of $A$, then the eigenvalues of $r(A)$ are $\{r(\lambda_j)\}$. If $r(A)$ can be computed with reasonable speed and its eigenvalues are better distributed for iteration than those of $A$, this may be a route to fast calculation of eigenvalues.

## 40.6 A Closing Note

In ending this book with the subject of preconditioners, we find ourselves at the philosophical center of the scientific computing of the future. The traditional view of computer scientists is that a computational problem is finite: after a short or long calculation, one obtains the solution exactly. Over the years, however, this view has come to be appropriate to fewer and fewer problems. The best methods for large-scale computational problems are usually approximate ones, methods that obtain a satisfactorily accurate solution in a short time rather than an exact one in a much longer or infinite time. Nu− merical analysis is indeed a branch of analysis, primarily, not algebra-even when the problems to be solved are from linear algebra. Further speculations on this phenomenon are presented in the Appendix.

Nothing will be more central to computational science in the next century than the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly. For Krylov subspace matrix iterations, this is preconditioning. For the great range of computational problems, both continuous and discrete, we can only guess where this idea will take us.

# APPENDIX A

# THE DEFINITION OF NUMERICAL ANALYSIS

**Note A.1.**
This essay by Lloyd N. Trefethen is reprinted from the November 1992 issue of SIAM News. It was reprinted previously in the March/April 1993 issue of the Bulletin of the Institute of Mathematics and Its Applications.

What is numerical analysis? I believe that this is more than a philosophical question. A certain wrong answer has taken hold among both outsiders to the field and insiders, distorting the image of a subject at the heart of the mathematical sciences. Here is the wrong answer:

Numerical analysis is the study of rounding errors. (D1)

The reader will agree that it would be hard to devise a more uninviting description of a field. Rounding errors are inevitable, yes, but they are complicated and tedious and - not fundamental. If (D1) is a common perception, it is hardly surprising that numerical analysis is widely regarded as an unglamorous subject. In fact, mathematicians, physicists, and computer scientists have all tended to hold numerical analysis in low esteem for many years-a most unusual consensus.

Of course nobody believes or asserts (D1) quite as baldly as written. But consider the following opening chapter headings from some standard numerical analysis texts:

- Isaacson & Keller (1966): 1. Norms, arithmetic, and well-posed computations.
- Hamming (1971): 1. Roundoff and function evaluation.
- Dahlquist & Björck (1974): 1. Some general principles of numerical calculation. 2. How to obtain and estimate accuracy....
- Stoer & Bulirsch (1980): 1. Error analysis.
- Conte & de Boor (1980): 1. Number systems and errors.
- Atkinson (1987): 1. Error: its sources, propagation, and analysis.
- Kahaner, Moler & Nash (1989): 1. Introduction. 2. Computer arithmetic and computational errors.

"Error" ... "roundoff"... "computer arithmetic" - these are the words that keep reappearing. What impression does an inquisitive college student get upon opening such books? Or consider the definitions of numerical analysis in some dictionaries:

- Webster's New Collegiate Dictionary (1973): "The study of quantitative approximations to the solutions of mathematical problems including consideration of the errors and bounds to the errors involved."
- Chambers 20th Century Dictionary (1983): "The study of methods of approximation and their accuracy, etc."
- The American Heritage Dictionary (1992): "The study of approximate solutions to mathematical problems, taking into account the extent of possible errors."

"Approximations" ... "accuracy" ... "errors" again. It seems to me that these definitions would serve most effectively to deter the curious from investigating further.

The singular value decomposition (SVD) affords another example of the perception of numerical analysis as the science of rounding errors. Although the roots of the SVD go back more than 100 years, it is mainly since the 1960s, through the work of Gene Golub and other numerical analysts, that it has achieved its present degree of prominence. The SVD is as fundamental an idea as the eigenvalue decomposition; it is the natural language for discussing all kinds of questions of norms and extrema involving nonsymmetric matrices or operators. Yet today, thirty years later, most mathematical scientists and even many applied mathematicians do not have a working knowledge of the SVD. Most of them have heard of it, but the impression seems to be widespread that the SVD is just a tool for combating rounding errors. A glance at a few numerical analysis textbooks suggests why. In one case after another, the SVD is buried deep in the book, typically in an advanced section on rank-deficient least squares problems, and recommended mainly for its stability properties. I am convinced that consciously or unconsciously, many people think that (D1) is at least half true. In actuality, it is a very small part of the truth. And although there are historical explanations for the influence of (D1) in the past, it is a less appropriate definition today and is destined to become still less appropriate in the future.

I propose the following alternative definition with which to enter the new century:

Numerical analysis is the study of algorithms for the problems of continuous mathematics. (D2)

Boundaries between fields are always fuzzy; no definition can be perfect. But it seems to me that (D2) is as sharp a characterization as you could come up with for most disciplines.

The pivotal word is algorithms. Where was this word in those chapter headings and dictionary definitions? Hidden between the lines, at best, and yet surely this is the center of numerical analysis: devising and analyzing algorithms to solve a certain class of problems.

These are the problems of continuous mathematics. "Continuous" means that real or complex variables are involved; its opposite is "discrete." A dozen qualifications aside, numerical analysts are broadly concerned with continuous problems, while algorithms for discrete problems are the concern of other computer scientists.

Let us consider the implications of (D2). First of all it is clear that since real and complex numbers cannot be represented exactly on computers, (D2) implies that part of the business of numerical analysis must be to approximate them. This is where the rounding errors come in. Now for a certain set of problems, namely the ones that are solved by algorithms that take a finite number of steps, that is all there is to it. The premier example is Gaussian elimination for solving a linear system of equations $Ax = b$. To understand Gaussian elimination, you have to understand computer science issues such as operation counts and machine architectures, and you have to understand the propagation of rounding errors-stability. That's all you have to understand, and if somebody claims that (D2) is just a more polite restatement of (D1), you can't prove him or her wrong with the example of Gaussian elimination. But most problems of continuous mathematics cannot be solved by finite algorithms! Unlike $Ax = b$, and unlike the discrete problems of computer science, most of the problems of numerical analysis could not be solved exactlyeven if we could work in exact arithmetic. Numerical analysts know this, and mention it along with a few words about Abel and Galois when they teach algorithms for computing matrix eigenvalues. Too often they forget to mention that the same conclusion extends to virtually any problem with a nonlinear term or a derivative in it-zerofinding, quadrature, differential equations, integral equations, optimization, you name it.

Even if rounding errors vanished, numerical analysis would remain. Approximating mere numbers, the task of floating point arithmetic, is indeed a rather small topic and maybe even a tedious one. The deeper business of numerical analysis is approximating unknowns, not knowns. Rapid convergence of approximations is the aim, and the pride of our field is that, for many problems, we have invented algorithms that converge exceedingly fast.

These points are sometimes overlooked by enthusiasts of symbolic computing, especially recent converts, who are apt to think that the existence of Maple or Mathematica renders Matlab and Fortran obsolete. It is true that rounding errors can be made to vanish in the sense that in principle, any finite

sequence of algebraic operations can be represented exactly on a computer by means of appropriate symbolic operations. Unless the problem being solved is a finite one, however, this only defers the inevitable approximations to the end of the calculation, by which point the quantities one is working with may have become extraordinarily cumbersome. Floating point arithmetic is a name for numerical analysts' habit of doing their pruning at every step along the way of a calculation rather than in a single act at the end. Whichever way one proceeds, in floating point or symbolically, the main problem of finding a rapidly convergent algorithm is the same.

In summary, it is a corollary of (D2) that numerical analysis is concerned with rounding errors and also with the deeper kinds of errors associated with convergence of approximations, which go by various names (truncation, discretization, iteration). Of course one could choose to make (D2) more explicit by adding words to describe these approximations and errors. But once words begin to be added it is hard to know where to stop, for (D2) also fails to mention some other important matters: that these algorithms are implemented on computers, whose architecture may be an important part of the problem; that reliability and efficiency are paramount goals; that some numerical analysts write programs and others prove theorems; and most important, that all of this work is applied, applied daily and successfully to thousands of applications on millions of computers around the world. "The problems of continuous mathematics" are the problems that science and engineering are built upon; without numerical methods, science and engineering as practiced today would come quickly to a halt. They are also the problems that preoccupied most mathematicians from the time of Newton to the twentieth century. As much as any pure mathematicians, numerical analysts are cthe heirs to the great tradition of Euler, Lagrange, Gauss and the rest. If Euler were alive today, he wouldn't be proving existence theorems.

---

Ten years ago, I would have stopped at this point. But the evolution of computing in the past decade has given the difference between (D1) and (D2) a new topicality.

Let us return to $Ax = b$. Much of numerical computation depends on linear algebra, and this highly developed subject has been the core of numerical analysis since the beginning. Numerical linear algebra served as the subject with respect to which the now standard concepts of stability, conditioning, and backward error analysis were defined and sharpened, and the central figure in these developments, from the 1950s to his death in 1986, was Jim Wilkinson. I have mentioned that $Ax = b$ has the unusual feature that it can be solved in a finite sequence of operations. In fact, $Ax = b$ is more unusual than that, for the standard algorithm for solving it, Gaussian elimination, turns out to have extraordinarily complicated stability properties. Von Neumann wrote 180 pages of mathematics on this topic; Turing wrote one of his major papers; Wilkinson developed a theory that grew into two books and a career. Yet the fact remains that for certain $n \times n$ matrices, Gaussian elimination with partial pivoting amplifies rounding errors by a factor of order $2^n$, making it a useless algorithm in the worst case. It seems that Gaussian elimination works so.

In manifold ways, then, Gaussian elimination is atypical. Few numerical algorithms have such subtle stability properties, and certainly no other was scrutinized in such depth by von Neumann, Turing, and Wilkinson. The effect? Gaussian elimination, which should have been a sideshow, lingered in the spotlight while our field was young and grew into the canonical algorithm of numerical analysis. Gaussian elimination set the agenda, Wilkinson set the tone, and the distressing result has been (D1).

Of course there is more than this to the history of how (D1) acquired currency. In the early years of computers, it was inevitable that arithmetic issues would receive concerted attention. Fixed point computation required careful thought and novel hardware; floating point computation arrived as a second revolution a few years later. Until these matters were well understood it was natural that arithmetic issues should be a central topic of numerical analysis, and, besides this, another force was at work. There is a general principle of computing that seems to have no name: the faster the computer, the more important the speed of algorithms. In the early years, with the early computers, the dangers of instability were nearly as great as they are today, and far less familiar. The gaps between fast and slow algorithms, however, were narrower.

A development has occurred in recent years that reflects how far we have come from that time. Instances have been accumulating in which, even though a finite algorithm exists for a problem, an

infinite algorithm may be better. The distinction that seems absolute from a logical point of view turns out to have little importance in practice - and in fact, Abel and Galois notwithstanding, large-scale matrix eigenvalue problems are about as easy to solve in practice as linear systems of equations. For $Ax = b$, iterative methods are becoming more and more often the methods of choice as computers grow faster, matrices grow larger and less sparse (because of the advance from 2D to 3D simulations), and the $O(N^3)$ operation counts of the usual direct (= finite) algorithms become ever more painful. The name of the new game is iteration with preconditioning. Increasingly often it is not optimal to try to solve a problem exactly in one pass; instead, solve it approximately, then iterate. Multigrid methods, perhaps the most important development in numerical computation in the past twenty years, are based on a recursive application of this idea.

Even direct algorithms have been affected by the new manner of computing. Thanks to the work of Skeel and others, it has been noticed that the expense of making a direct method stable-say, of pivoting in Gaussian elimination-may in certain contexts be cost-ineffective. Instead, skip that step-solve the problem directly but unstably, then do one or two steps of iterative refinement. "Exact" Gaussian elimination becomes just another preconditioner!

Other problems besides $Ax = b$ have undergone analogous changes, and the famous example is linear programming. Linear programming problems are mathematically finite, and for decades, people solved them by a finite algorithm: the simplex method. Then Karmarkar announced in 1984 that iterative, infinite algorithms are sometimes better. The result has been controversy, intellectual excitement, and a perceptible shift of the entire field of linear programming away from the rather anomalous position it has traditionally occupied towards the mainstream of numerical computation.

I believe that the existence of finite algorithms for certain problems, together with other historical forces, has distracted us for decades from a balanced view of numerical analysis. Rounding errors and instability are important, and numerical analysts will always be the experts in these subjects and at pains to ensure that the unwary are not tripped up by them. But our central mission is to compute quantities that are typically uncomputable, from an analytical point of view, and to do it with lightning speed. For guidance to the future we should study not Gaussian elimination and its beguiling stability properties, but the diabolically fast conjugate gradient iteration-or Greengard and Rokhlin's $O(N)$ multipole algorithm for particle simulations-or the exponential convergence of spectral methods for solving certain PDEs-or the convergence in $O(1)$ iteration achieved by multigrid methods for many kinds of problems-or even Borwein and Borwein's magical AGM iteration for determining $1,000,000$ digits of $\pi$ in the blink of an eye. That is the heart of numerical analysis.