Fall Quarter 2024

ECE133A Applied Numerical Computing
Additional Lecture Notes

L. Vandenberghe

Contents

Chapter 1

LU factorization

In this chapter we discuss the standard method for solving a square set of linear equations $Ax = b$ with nonsingular coefficient matrix A.

1.1 Definition

Every nonsingular $n \times n$ matrix A can be factored as

$$A = PLU$$

where P is a permutation matrix, L is a unit lower triangular matrix, and U is a nonsingular upper triangular matrix. The matrices P, L, and U have size $n \times n$. This is called the *LU factorization* of A. The factorization can also be written as $P^T A = LU$, where the matrix $P^T A$ is obtained from A by reordering the rows.

An example of a 3×3 LU factorization is

$$\begin{bmatrix} 0 & 5 & 5 \\ 2 & 9 & 0 \\ 6 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ 0 & 15/19 & 1 \end{bmatrix} \begin{bmatrix} 6 & 8 & 8 \\ 0 & 19/3 & -8/3 \\ 0 & 0 & 135/19 \end{bmatrix}. \tag{1.1}$$

Another factorization of the same matrix is

$$\begin{bmatrix} 0 & 5 & 5 \\ 2 & 9 & 0 \\ 6 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & -19/5 & 1 \end{bmatrix} \begin{bmatrix} 2 & 9 & 0 \\ 0 & 5 & 5 \\ 0 & 0 & 27 \end{bmatrix}.$$

This shows that the LU factorization is not always unique.

The standard algorithm for computing an LU factorization is called *Gaussian elimination with partial pivoting* (GEPP) or *Gaussian elimination with row pivoting*, and will be described in section 1.4. The complexity is $(2/3)n^3$ flops.

1.2 Nonsingular sets of linear equations

If we use the LU factorization in the factor–solve method we obtain an algorithm for solving linear equations with a general nonsingular coefficient matrix.

Algorithm 1.1. Solving linear equations by LU factorization.

```
given a set of linear equations $Ax = b$ with $A$ $n \times n$ and nonsingular.

1. LU factorization: factor $A$ as $A = PLU$ ($(2/3)n^3$ flops).
2. Permutation: $v = P^T b$ (0 flops).
3. Forward substitution: solve $Lw = v$ ($n^2$ flops).
4. Back substitution: solve $Ux = w$ ($n^2$ flops).
```

The complexity of the factorization step is $(2/3)n^3$, and the complexity of the three other steps is $2n^2$. The total complexity is $(2/3)n^3 + 2n^2$, or $(2/3)n^3$ flops if we keep only the leading term. This algorithm is the standard method for solving linear equations.

Example We solve the equations

$$\begin{bmatrix} 0 & 5 & 5 \\ 2 & 9 & 0 \\ 6 & 8 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 15 \\ 7 \\ 18 \end{bmatrix}$$

using the factorization of the coefficient matrix given in (1.1). In the permutation step we solve

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 15 \\ 7 \\ 18 \end{bmatrix},$$

which yields $v = (18, 7, 15)$. Next, we solve

$$\begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ 0 & 15/19 & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 18 \\ 7 \\ 15 \end{bmatrix}$$

by forward substitution. The solution is $w = (18, 1, 270/19)$. Finally, we solve

$$\begin{bmatrix} 6 & 8 & 8 \\ 0 & 19/3 & -8/3 \\ 0 & 0 & 135/19 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 18 \\ 1 \\ 270/19 \end{bmatrix}$$

by backward substitution, and find the solution $x = (-1, 1, 2)$.

Equations with multiple right-hand sides Multiple sets of linear equations with different right-hand sides,

$$Ax_1 = b_1, \qquad Ax_2 = b_2, \qquad \ldots, \qquad Ax_m = b_m,$$

with a nonsingular coefficient matrix A, can be solved in

$$(2/3)n^3 + 2mn^2$$

flops, since we factor A once, and carry out m pairs of forward and backward substitutions. For example, we can solve two sets of linear equations, with the same coefficient matrix but different right-hand sides, at essentially the same cost as solving one.

1.3 Inverse of a nonsingular matrix

If A is nonsingular with LU factorization $A = PLU$ then its inverse can be expressed as

$$A^{-1} = U^{-1}L^{-1}P^T.$$

This expression gives another interpretation of algorithm 1.1. In steps 2–4, we evaluate $x = A^{-1}b = U^{-1}L^{-1}P^Tb$ as $x = U^{-1}(L^{-1}(P^Tb))$. First $v = P^Tb$ is computed, then $w = L^{-1}v$, and finally $x = U^{-1}w$.

Computing the inverse The inverse A^{-1} can be computed by solving the matrix equation

$$AX = I$$

or, equivalently, the n equations $Ax_i = e_i$, for $i = 1, \ldots, n$, where x_i is the ith column of X, and e_i is the ith unit vector. This requires one LU factorization and n pairs of forward and backward substitutions. The complexity is $(2/3)n^3 + n(2n^2) = (8/3)n^3$ flops, or about $3n^3$ flops.

The complexity can be reduced by taking advantage of the zero structure in the right-hand sides e_i. Note that forward substitution with the unit vector e_j as right-hand side takes $(n - j + 1)^2$ flops. Therefore $L^{-1}P^T$ can be computed in

$$\sum_{j=1}^{n}(n - j + 1)^2 \sim \frac{n^3}{3}$$

flops, reducing the complexity for computing A^{-1} to $2n^3$ flops.

1.4 Computing the LU factorization without pivoting

Before we describe the general algorithm for LU factorization, it is useful to consider the simpler factorization

$$A = LU$$

where L is unit lower triangular and U is upper triangular and nonsingular. We refer to this factorization as the *LU factorization without permutations* (or without pivoting). It is a special case of the general LU factorization with $P = I$.

To see how we can calculate L and U, we partition the matrices on both sides of $A = LU$ as

$$\begin{bmatrix} A_{11} & A_{1,2:n} \\ A_{2:n,1} & A_{2:n,2:n} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{2:n,1} & L_{2:n,2:n} \end{bmatrix} \begin{bmatrix} U_{11} & U_{1,2:n} \\ 0 & U_{2:n,2:n} \end{bmatrix}.$$

If we work out the product on the right-hand side, we obtain

$$\begin{bmatrix} A_{11} & A_{1,2:n} \\ A_{2:n,1} & A_{2:n,2:n} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{1,2:n} \\ U_{11}L_{2:n,1} & L_{2:n,1}U_{1,2:n} + L_{2:n,2:n}U_{2:n,2:n} \end{bmatrix}.$$

Equating both sides allows us to determine the first row of U and the first column of L:

$$U_{11} = A_{11}, \qquad U_{1,2:n} = A_{1,2:n}, \qquad L_{2:n,1} = \frac{1}{A_{11}} A_{2:n,1}.$$

Furthermore,

$$L_{2:n,2:n}U_{2,2:n} = A_{2:n,2:n} - L_{2:n,1}U_{1,2:n} = A_{2:n,2:n} - \frac{1}{A_{11}} A_{2:n,1}A_{1,2:n},$$

so we can calculate $L_{2:n,2:n}$ and $U_{2:n,2:n}$ by factoring $A_{2:n,2:n} - L_{2:n,1}U_{1,2:n}$ as

$$A_{2:n,2:n} - L_{2:n,1}U_{1,2:n} = L_{2:n,2:n}U_{2:n,2:n},$$

which is an LU factorization of size $(n-1) \times (n-1)$. This suggests a recursive algorithm: to factor a matrix of size $n \times n$, we calculate the first column of L and the first row of U, and then factor a matrix of order $n-1$. Continuing recursively, we arrive at a factorization of a 1×1 matrix, which is trivial.

Algorithm 1.2. LU FACTORIZATION WITHOUT PIVOTING.

given an $n \times n$-matrix A that can be factored as $A = LU$.

1. Calculate the first row of U: $U_{11} = A_{11}$ and $U_{1,2:n} = A_{1,2:n}$.
2. Calculate the first column of L: $L_{2:n,1} = (1/A_{11})A_{2:n,1}$.
3. Calculate the LU factorization $A_{2:n,2:n} - L_{2:n,1}U_{1,2:n} = L_{2:n,2:n}U_{2:n,2:n}$.

It can be shown that the complexity is $(2/3)n^3$ flops.

Example As an example, let us factor the matrix

$$A = \begin{bmatrix} 8 & 2 & 9 \\ 4 & 9 & 4 \\ 6 & 7 & 9 \end{bmatrix},$$

as $A = LU$, *i.e.*,

$$A = \begin{bmatrix} 8 & 2 & 9 \\ 4 & 9 & 4 \\ 6 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}.$$

We start by calculating the first row of U and the first column of L:

$$\begin{bmatrix} 8 & 2 & 9 \\ 4 & 9 & 4 \\ 6 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 3/4 & L_{32} & 1 \end{bmatrix} \begin{bmatrix} 8 & 2 & 9 \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}.$$

The remaining elements are obtained from the identity

$$\begin{aligned} \begin{bmatrix} 9 & 4 \\ 7 & 9 \end{bmatrix} &= \begin{bmatrix} 1/2 & 1 & 0 \\ 3/4 & L_{32} & 1 \end{bmatrix} \begin{bmatrix} 2 & 9 \\ U_{22} & U_{23} \\ 0 & U_{33} \end{bmatrix} \\ &= \begin{bmatrix} 1/2 \\ 3/4 \end{bmatrix} \begin{bmatrix} 2 & 9 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{22} & U_{23} \\ 0 & U_{33} \end{bmatrix} \end{aligned}$$

or

$$\begin{bmatrix} 8 & -1/2 \\ 11/2 & 9/4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{22} & U_{23} \\ 0 & U_{33} \end{bmatrix}.$$

This is a 2×2 LU factorization. Again it is easy to find the first rows and columns of the triangular matrices on the right:

$$\begin{bmatrix} 8 & -1/2 \\ 11/2 & 9/4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 11/16 & 1 \end{bmatrix} \begin{bmatrix} 8 & -1/2 \\ 0 & U_{33} \end{bmatrix}.$$

Finally, the remaining element U_{33} follows from

$$9/4 = -(11/16) \cdot (1/2) + U_{33},$$

i.e., $U_{33} = 83/32$. Putting everything together, we obtain the following factorization:

$$A = \begin{bmatrix} 8 & 2 & 9 \\ 4 & 9 & 4 \\ 6 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 3/4 & 11/16 & 1 \end{bmatrix} \begin{bmatrix} 8 & 2 & 9 \\ 0 & 8 & -1/2 \\ 0 & 0 & 83/32 \end{bmatrix}.$$

Existence of LU factorization without permutations Simple examples show that not every nonsingular matrix can be factored as $A = LU$. The matrix

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

for example, is nonsingular, but it cannot be factored as

$$\begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ L_{21} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}.$$

If we apply algorithm 1.2, we first find $U_{11} = 0$, $U_{12} = 1$. Hence, L_{21} must satisfy $1 = L_{21} \cdot 0$, which is impossible. In this example the factorization algorithm breaks down in the first step because $A_{11} = 0$. Even if the first step succeeds ($A_{11} \neq 0$), we might run into the same problem (division by zero) in any of the subsequent recursive steps.

1.5 Computing the LU factorization (with pivoting)

We now extend algorithm 1.2 to the full LU factorization $A = PLU$. We will see that such a factorization exists if A is nonsingular, and we will describe a recursive algorithm to compute it.

We first note that a nonsingular 1×1 matrix A is simply a nonzero scalar, so its LU factorization is trivial: $A = PLU$ with $P = 1$, $L = 1$, $U = A$. Next we show that if it is true that every nonsingular $(n-1) \times (n-1)$ matrix has an LU factorization, then the same is true for nonsingular $n \times n$ matrices. By induction, this proves that every nonsingular matrix has an LU factorization.

Let us therefore assume that every nonsingular matrix of size $(n-1) \times (n-1)$ has an LU factorization. Suppose we want to factor a nonsingular matrix A of size $n \times n$. At least one element in the first column of A must be nonzero (since a matrix with a zero column is singular). We can therefore apply a row permutation that makes the 1,1 element nonzero. In other words, there exists a permutation matrix P_1 such that the matrix $B = P_1^T A$ satisfies $B_{11} \neq 0$. Clearly, the matrix B is nonsingular if A is nonsingular. We partition the permuted matrix B as

$$B = P_1^T A = \left[\begin{array}{cc} B_{11} & B_{1,2:n} \\ B_{2:n,1} & B_{2:n,2:n} \end{array}\right].$$

Now consider the $(n-1) \times (n-1)$ matrix

$$B_{2:n,2:n} - \frac{1}{B_{11}} B_{2:n,1} B_{1,2:n}. \tag{1.2}$$

We show that this matrix is nonsingular if B is. Suppose an $(n-1)$-vector x satisfies

$$\left(B_{2:n,2:n} - \frac{1}{B_{11}} B_{2:n,1} B_{1,2:n}\right) x = 0. \tag{1.3}$$

Then x and $y = -(1/B_{11}) B_{1,2:n} x$ satisfy

$$\left[\begin{array}{cc} B_{11} & B_{1,2:n} \\ B_{2:n,1} & B_{2:n,2:n} \end{array}\right] \left[\begin{array}{c} y \\ x \end{array}\right] = \left[\begin{array}{c} 0 \\ 0 \end{array}\right].$$

Since B is nonsingular, y and x must be zero. Hence, (1.3) holds only if $x = 0$, which means that the matrix (1.2) is nonsingular.

By the induction assumption, this means that it can be factored as

$$B_{2:n,2:n} - \frac{1}{B_{11}} B_{2:n,1} B_{1,2:n} = P_2 L_2 U_2.$$

This provides the desired LU factorization of A:

$$\begin{aligned}
A &= P_1 \left[\begin{array}{cc} B_{11} & B_{1,2:n} \\ B_{2:n,1} & B_{2:n,2:n} \end{array}\right] \\
&= P_1 \left[\begin{array}{cc} 1 & 0 \\ 0 & P_2 \end{array}\right] \left[\begin{array}{cc} B_{11} & B_{1,2:n} \\ P_2^T B_{2:n,1} & P_2^T B_{2:n,2:n} \end{array}\right] \\
&= P_1 \left[\begin{array}{cc} 1 & 0 \\ 0 & P_2 \end{array}\right] \left[\begin{array}{cc} B_{11} & B_{1,2:n} \\ P_2^T B_{2:n,1} & L_2 U_2 + (1/B_{11}) P_2^T B_{2:n,1} B_{1,2:n} \end{array}\right] \\
&= P_1 \left[\begin{array}{cc} 1 & 0 \\ 0 & P_2 \end{array}\right] \left[\begin{array}{cc} 1 & 0 \\ (1/B_{11}) P_2^T B_{2:n,1} & L_2 \end{array}\right] \left[\begin{array}{cc} B_{11} & B_{1,2:n} \\ 0 & U_2 \end{array}\right].
\end{aligned}$$

If we define

$$P = P_1 \begin{bmatrix} 1 & 0 \\ 0 & P_2 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 \\ (1/B_{11})P_2^T B_{2:n,1} & L_2 \end{bmatrix}, \quad U = \begin{bmatrix} B_{11} & B_{1,2:n} \\ 0 & U_2 \end{bmatrix},$$

then P is a permutation matrix, L is unit lower triangular, U is upper triangular, and $A = PLU$.

We summarize the idea by stating the algorithm recursively.

Algorithm 1.3. LU FACTORIZATION.

given a nonsingular $n \times n$-matrix A.

1. Choose P_1 such that $B = P_1^T A$ satisfies $B_{11} \neq 0$.
2. Compute the LU factorization $B_{2:n,2:n} - (1/B_{11})B_{2:n,1}B_{1,2;n} = P_2 L_2 U_2$.
3. The LU factorization of A is $A = PLU$ with

$$P = P_1 \begin{bmatrix} 1 & 0 \\ 0 & P_2 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 \\ (1/B_{11})P_2^T B_{2:n,1} & L_2 \end{bmatrix}, \quad U = \begin{bmatrix} B_{11} & B_{1,2:n} \\ 0 & U_2 \end{bmatrix}.$$

It can be shown that the total complexity is $(2/3)n^3$ flops.

Example Let us factor the matrix

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

Since $A_{11} = 0$ we need a permutation that brings the second row or the third row in the first position. For example, we can choose

$$P_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix},$$

and proceed with the factorization of

$$B = P_1^T A = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 5 & 5 \\ 2 & 3 & 0 \\ 6 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 9 & 8 \\ 2 & 3 & 0 \\ 0 & 5 & 5 \end{bmatrix}.$$

To factor B we need the LU factorization of

$$\begin{aligned} A^{(1)} = B_{2:n,2:n} - (1/B_{11})B_{2:n,1}B_{1,2:n} &= \begin{bmatrix} 3 & 0 \\ 5 & 5 \end{bmatrix} - (1/6) \begin{bmatrix} 2 \\ 0 \end{bmatrix} \begin{bmatrix} 9 & 8 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -8/3 \\ 5 & 5 \end{bmatrix}. \end{aligned}$$

The first element is zero, so we need to apply a permutation

$$P_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

in order to switch the two rows. We denote the permuted matrix as $B^{(1)}$:

$$B^{(1)} = P_2^T A^{(1)} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -8/3 \\ 5 & 5 \end{bmatrix} = \begin{bmatrix} 5 & 5 \\ 0 & -8/3 \end{bmatrix}.$$

This matrix is upper triangular, so we do not have to go to the next level of the recursion to find its factorization: $B^{(1)} = L_2 U_2$ and $A^{(1)} = P_2 L_2 U_2$ where

$$L_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad U_2 = \begin{bmatrix} 5 & 5 \\ 0 & -8/3 \end{bmatrix}.$$

The LU factorization $A = PLU$ can now be assembled as in step 3 of the algorithm outline. The permutation matrix is

$$P = P_1 \begin{bmatrix} 1 & 0 \\ 0 & P_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

The lower triangular matrix is

$$L = \begin{bmatrix} 1 & 0 \\ (1/B_{11}) P_2^T B_{2:n,1} & L_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1/3 & 0 & 1 \end{bmatrix}.$$

The upper triangular matrix is

$$U = \begin{bmatrix} B_{11} & B_{1,2:n} \\ 0 & U_2 \end{bmatrix} = \begin{bmatrix} 6 & 9 & 8 \\ 0 & 5 & 5 \\ 0 & 0 & -8/3 \end{bmatrix}.$$

In summary, an LU factorization of A is

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

1.6 Effect of rounding error

In this section we discuss the effect of rounding errors on the accuracy of the LU factorization method for solving linear equations. As an example, we consider two equations in two variables

$$\begin{aligned} 10^{-5} x_1 + x_2 &= 1 \\ x_1 + x_2 &= 0. \end{aligned}$$

The solution is

$$x_1 = \frac{-1}{1 - 10^{-5}}, \qquad x_2 = \frac{1}{1 - 10^{-5}} \tag{1.4}$$

as is easily verified by substitution. We will solve the equations using the LU factorization, and introduce small errors by rounding some of the intermediate results to four significant decimal digits.

The matrix

$$A = \begin{bmatrix} 10^{-5} & 1 \\ 1 & 1 \end{bmatrix}$$

has two LU factorizations:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 10^5 & 1 \end{bmatrix} \begin{bmatrix} 10^{-5} & 1 \\ 0 & 1-10^5 \end{bmatrix} \tag{1.5}$$

$$= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 10^{-5} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1-10^{-5} \end{bmatrix}. \tag{1.6}$$

Suppose we use the first factorization, and round the $2, 2$ element of U,

$$U_{22} = 1 - 10^5 = -0.99999 \cdot 10^5,$$

to four significant digits, *i.e.*, replace it with $-1.0000 \cdot 10^5 = -10^5$. (The other elements of L and U do not change if we round them to four significant digits.) We proceed with the solve step, using the approximate factorization

$$A \approx \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 10^5 & 1 \end{bmatrix} \begin{bmatrix} 10^{-5} & 1 \\ 0 & -10^5 \end{bmatrix}.$$

In the forward substitution step we solve

$$\begin{bmatrix} 1 & 0 \\ 10^5 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

which gives $z_1 = 1$, $z_2 = -10^5$. In the back substitution step we solve

$$\begin{bmatrix} 10^{-5} & 1 \\ 0 & -10^5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -10^5 \end{bmatrix}.$$

The solution is $x_1 = 0$, $x_2 = 1$. Comparing this with the exact solution (1.4), we see that the error in x_1 is 100%! One small rounding error (replacing -0.99999 by -1) has caused a huge error in the result.

This phenomenon is called *numerical instability*. An algorithm is numerically unstable if small rounding errors can cause a very large error in the result. The example shows that solving linear equations using the LU factorization can be numerically unstable.

In a similar way, we can examine the effect of rounding errors on the second factorization (1.6). Suppose we round the $2, 2$ element $U_{22} = 1 - 10^{-5} = 0.99999$ to 1, and use the approximate factorization

$$A \approx \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 10^{-5} & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

In the forward substitution step we solve

$$\begin{bmatrix} 1 & 0 \\ 10^{-5} & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

which gives $z_1 = 0$, $z_2 = 1$. Solving

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

by back substitution, yields $x_1 = -1$, $x_2 = 1$. In this case the error in the result is very small (about 10^{-5}, *i.e.*, of the same order as the rounding error that we introduced).

We conclude that choosing the wrong permutation matrix P can have a disastrous effect on the accuracy of the solution.

An error analysis and extensive practical experience have provided a good remedy for numerical instability in the LU factorization: when selecting the permutation matrix P in step 1 of the algorithm of section 1.5, choose P so that B_{11} is the element with the largest absolute value in the first column of A. (This is consistent with our observation in the small example: the largest element of the first column of A is 1, so according to this guideline, we should permute the two rows of A.) With this selection rule for P, the algorithm is stable, except in rare cases.

1.7 Sparse linear equations

When the matrix A is sparse, the LU factorization usually includes both row and column permutations, *i.e.*, A is factored as

$$A = P_1 L U P_2, \tag{1.7}$$

where P_1 and P_2 are permutation matrices, L is unit lower triangular, and U is upper triangular. If the factors L and U are sparse, the forward and backward substitutions can be carried out efficiently, and we have an efficient method for solving $Ax = b$.

Sparse LU factorization algorithms must take into account two criteria when selecting the permutation matrices P_1 and P_2:

1. *Numerical stability.* As we have seen in section 1.6 the LU factorization algorithm may be numerically unstable (or may not exist) for certain choices of the permutation matrix. The same is true for the factorization (1.7).

2. *Sparsity of L and U.* The sparsity of the factors L and U depends on the permutations P_1 and P_2, which have to be chosen in a way that tends to yield sparse L and U.

Achieving a good compromise between these two objectives is quite difficult, and codes for sparse LU factorization are much more complicated than codes for dense LU factorization.

The complexity of computing the sparse LU factorization depends in a complicated way on the size of A, the number of nonzero elements, its sparsity pattern, and the particular algorithm used, but is often much smaller than the complexity of a dense LU factorization. In many cases the complexity grows approximately linearly with n, when n is large. This means that when A is sparse, we can solve $Ax = b$ very efficiently, often with an order approximately n.

Chapter 2

Cholesky factorization

2.1 Positive definite matrices

2.1.1 Definition

A square matrix A is *positive definite* if it is symmetric ($A = A^T$) and

$$x^T Ax > 0 \text{ for all nonzero } x.$$

In other words, $x^T Ax \geq 0$ for all x and $x^T Ax = 0$ only if $x = 0$. A slightly larger class is the set of *positive semidefinite* matrices: a matrix A is positive semidefinite if it is symmetric and

$$x^T Ax \geq 0 \text{ for all } x.$$

All positive definite matrices are positive semidefinite but not vice-versa, because the definition of positive definite matrix includes the extra requirement that $x = 0$ is the only vector for which $x^T Ax = 0$.

Examples Practical methods for checking positive definiteness of a matrix will be discussed later in this chapter. For small matrices, one can directly apply the definition and examine the sign of the product $x^T Ax$. It is useful to note that for a general square matrix A, the product $x^T Ax$ can be expanded as

$$x^T Ax = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j,$$

i.e., $x^T Ax$ is a sum of all products $x_i x_j$ multiplied with the i, j element of A. Each product $x_i x_j$ for $i \neq j$ appears twice in the sum, and if A is symmetric the coefficients of the two terms are equal ($A_{ij} = A_{ji}$). Therefore, for a symmetric matrix,

$$x^T Ax = \sum_{i=1}^{n} A_{ii} x_i^2 + 2 \sum_{i=1}^{n} \sum_{j=1}^{i-1} A_{ij} x_i x_j.$$

The first sum is over the diagonal elements of A; the second sum is over the strictly lower-triangular elements.

For example, with

$$A = \begin{bmatrix} 9 & 6 \\ 6 & 5 \end{bmatrix}$$

we get $x^TAx = 9x_1^2 + 12x_1x_2 + 5x_2^2$. To check the sign of x^TAx, we complete the squares:

$$\begin{array}{rcl} x^TAx & = & 9x_1^2 + 12x_1x_2 + 5x_2^2 \\ & = & (3x_1 + 2x_2)^2 - 4x_2^2 + 5x_2^2 \\ & = & (3x_1 + 2x_2)^2 + x_2^2. \end{array}$$

Clearly $x^TAx \geq 0$ for all x. Moreover $x^TAx = 0$ only if $x_1 = x_2 = 0$. Therefore $x^TAx > 0$ for all nonzero x and we conclude that A is positive definite.

The matrix

$$A = \begin{bmatrix} 9 & 6 \\ 6 & 3 \end{bmatrix}, \tag{2.1}$$

on the other hand, is not positive semidefinite, because

$$\begin{array}{rcl} x^TAx & = & 9x_1^2 + 12x_1x_2 + 3x_2^2 \\ & = & (3x_1 + 2x_2)^2 - 4x_2^2 + 3x_2^2 \\ & = & (3x_1 + 2x_2)^2 - x_2^2. \end{array}$$

From this expression it is easy to find values of x for which $x^TAx < 0$. For example, for $x = (-2/3, 1)$, we get $x^TAx = -1$.

The matrix

$$A = \begin{bmatrix} 9 & 6 \\ 6 & 4 \end{bmatrix}$$

is positive semidefinite because

$$\begin{array}{rcl} x^TAx & = & 9x_1^2 + 12x_1x_2 + 4x_2^2 \\ & = & (3x_1 + 2x_2)^2 \end{array}$$

and this is nonnegative for all x. A is not positive definite because $x^TAx = 0$ for some nonzero x, *e.g.*, $x = (2, -3)$.

2.1.2 Properties

Diagonal The diagonal elements of a positive definite matrix are all positive. This is easy to show from the definition. If A is positive definite, then $x^TAx > 0$ for all nonzero x. In particular, $x^TAx > 0$ for $x = e_i$ (the ith unit vector). Therefore,

$$e_i^TAe_i = A_{ii} > 0$$

for $i = 1, \ldots, n$. Positivity of the diagonal elements is a *necessary* condition for positive definitess of A, but it is far from sufficient. For example, the matrix defined in (2.1) is not positive definite, although it has positive diagonal elements.

Schur complement Consider a positive definite matrix A of size $n \times n$. Partition A as

$$A = \left[\begin{array}{cc} A_{11} & A_{2:n,1}^T \\ A_{2:n,1} & A_{2:n,2:n} \end{array} \right].$$

(Since A is symmetric, we wrote its $1,2$ block as the transpose of the $2,1$ block.) The matrix

$$S = A_{2:n,2:n} - \frac{1}{A_{11}} A_{2:n,1} A_{2:n,1}^T,$$

is called the *Schur complement* of A_{11} and is defined because $A_{11} \neq 0$. To show that S is positive definite, pick any nonzero $(n-1)$-vector v. Then the vector

$$x = \left[\begin{array}{c} -(1/A_{11}) A_{2:n,1}^T v \\ v \end{array} \right]$$

is nonzero and satisfies

$$\begin{array}{rl} x^T A x & \\ = & \left[\begin{array}{cc} -(1/A_{11}) v^T A_{2:n,1} & v^T \end{array} \right] \left[\begin{array}{cc} A_{11} & A_{2:n,1}^T \\ A_{2:n,1} & A_{2:n,2:n} \end{array} \right] \left[\begin{array}{c} -(1/A_{11}) A_{2:n,1}^T v \\ v \end{array} \right] \\ = & \left[\begin{array}{cc} -(1/A_{11}) v^T A_{2:n,1} & v^T \end{array} \right] \left[\begin{array}{c} 0 \\ Sv \end{array} \right] \\ = & v^T S v. \end{array}$$

Since A is positive definite, we have $x^T A x = v^T S v > 0$ for all nonzero v. Therefore S is positive definite.

Positive definite matrices are nonsingular Positive definite matrices are nonsingular: if $Ax = 0$ then $x^T A x = 0$ and if A is positive definite, this implies $x = 0$.

2.1.3 Gram matrices

A *Gram matrix* is a matrix of the form $B^T B$. All Gram matrices are positive semidefinite since

$$x^T B^T B x = (Bx)^T (Bx) = \|Bx\|^2 \geq 0 \qquad (2.2)$$

for all x. This holds regardless of the values and dimensions of B. A Gram matrix is not necessarily positive definite and (2.2) shows what the required properties of B are: the matrix $B^T B$ is positive definite if $\|Bx\| = 0$ implies $x = 0$. In other words, $B^T B$ is positive definite if and only if B has linearly independent columns.

2.1.4 Singular positive semidefinite matrices

We have seen that positive definite matrices are nonsingular. We now show that if A is positive semidefinite but not positive definite, then it is singular.

Assume that A is positive semidefinite ($x^T Ax \geq 0$ for all x) but not positive definite, *i.e.*, there exists a nonzero x with $x^T Ax = 0$. We will show that $Ax = 0$ and therefore A is singular.

Take an arbitrary y and define the function

$$f(t) = (x + ty)^T A(x + ty)$$

where t is a scalar variable. We have $f(t) \geq 0$ for all t because A is positive semidefinite. Expanding the products in the definition of f and using $x^T Ax = 0$, we obtain

$$\begin{aligned} f(t) &= x^T Ax + tx^T Ay + ty^T Ax + t^2 y^T Ay \\ &= 2ty^T Ax + t^2 y^T Ay. \end{aligned}$$

The function f is quadratic in t, with $f(0) = 0$. Since $f(t) \geq 0$ for all t the derivative of f at $t = 0$ must be zero, *i.e.*, $f'(0) = 2y^T Ax = 0$. Since y is arbitrary, this is only possible if $Ax = 0$.

2.2 Cholesky factorization

Every positive definite matrix A can be factored as

$$A = R^T R$$

where R is upper triangular with positive diagonal elements. This is called the *Cholesky factorization* of A. The matrix R, which is uniquely determined by A, is called the *Cholesky factor* of A. If $n = 1$ (*i.e.*, A is a positive scalar), then R is just the square root of A. So we can also interpret the Cholesky factor as a kind of "square root" of a positive definite matrix A.

An example of a 3×3 Cholesky factorization is

$$\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 \\ 3 & 3 & 0 \\ -1 & 1 & 3 \end{bmatrix} \begin{bmatrix} 5 & 3 & -1 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{bmatrix}. \qquad (2.3)$$

In section 2.3 we will discuss how the Cholesky factorization is computed, and we will find that the complexity of factoring a matrix of order n is $(1/3)n^3$ flops.

2.2.1 Positive definite sets of linear equations

It is straightforward to solve $Ax = b$, with A positive definite, if we first compute the Cholesky factorization $A = R^T R$. To solve $R^T Rx = b$, we introduce an intermediate variable $w = Rx$, and compute w and x by solving two triangular sets of equations. We first solve $R^T w = b$ for w. Then we solve $Rx = w$ to find x. These two equations are solvable, because R is upper triangular with positive diagonal elements.

Algorithm 2.1. Solving linear equations by Cholesky factorization.

```
given a set of linear equations with $A$ positive definite of size $n \times n$.
  1. Cholesky factorization: factor $A$ as $A = R^T R$ ($(1/3)n^3$ flops).
  2. Forward substitution: solve $R^T w = b$ ($n^2$ flops).
  3. Back substitution: solve $Rx = w$ ($n^2$ flops).
```

The total complexity is $(1/3)n^3 + 2n^2$, or roughly $(1/3)n^3$ flops.

Example We compute the solution of

$$\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 30 \\ 15 \\ -16 \end{bmatrix},$$

using the factorization in (2.3). We first solve

$$\begin{bmatrix} 5 & 0 & 0 \\ 3 & 3 & 0 \\ -1 & 1 & 3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 30 \\ 15 \\ -16 \end{bmatrix}$$

by forward substitution. The solution is $w = (6, -1, -3)$. Then we solve

$$\begin{bmatrix} 5 & 3 & -1 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -1 \\ -3 \end{bmatrix}$$

by back substitution and obtain the solution $x = (1, 0, -1)$.

Equations with multiple right-hand sides Suppose we need to solve m equations

$$Ax_1 = b_1, \qquad Ax_2 = b_2, \qquad \ldots, \qquad Ax_m = b_m,$$

where A is positive definite of size $n \times n$, and $b_1, \ldots, b_m$ are n-vectors. These m sets of equations have the same coefficient matrix but different right-hand sides $b_1, \ldots, b_m$. Alternatively, we can think of the problem as solving the matrix equation

$$AX = B$$

where X and B are the $n \times m$-matrices

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}, \qquad B = \begin{bmatrix} b_1 & b_2 & \cdots & b_m \end{bmatrix}.$$

We first compute the Cholesky factorization of A, which costs $(1/3)n^3$. Then for $i = 1, \ldots, m$, we solve $Ax_i = b_i$ using forward and backward substitution (this costs $2n^2$ flops per right-hand side). Since we only factor A once, the total effort is

$$(1/3)n^3 + 2mn^2$$

flops. Had we (needlessly) repeated the factorization step for each right-hand side, the cost would have been $m((1/3)n^3 + 2n^2)$. Since the factorization cost ($(1/3)n^3$) dominates the cost of forward and backward substitution ($2n^2$), this method allows us to solve a small number of linear systems, with the same coefficient matrix, at roughly the cost of solving one.

2.2.2 Inverse of a positive definite matrix

If A is positive definite with Cholesky factorization $A = R^T R$, then its inverse can be expressed as

$$A^{-1} = R^{-1} R^{-T}.$$

This follows from the expressions for the inverse of a product of nonsingular matrices and for the inverse of a transpose, and the fact that R is invertible because it is upper-triangular with nonzero diagonal elements.

This expression provides another interpretation of algorithm 2.1. If we multiply both sides of the equation $Ax = b$ on the left by A^{-1}, we get

$$x = A^{-1} b = R^{-1} R^{-T} b.$$

Steps 2 and 3 of the algorithm evaluate this expression by first computing $w = R^{-T} b$, and then $x = R^{-1} w = R^{-1} R^{-T} b$.

Computing the inverse The inverse of a positive definite matrix can be computed by solving

$$AX = I,$$

using the method for solving equations with multiple right-hand sides described in the previous section. Only one Cholesky factorization of A is required. Given the Cholesky factors, we first compute $Y = R^{-T}$ by solving $R^T Y = I$, column by column, via n forward substitutions. Then we compute $X = R^{-1} Y$ by solving $RX = Y$, column by column, via n backward substitutions The complexity of computing the inverse using this method is $(1/3)n^3 + 2n^3 = (7/3)n^3$, or about $2n^3$.

This number can be reduced by taking advantage of two properties. First, when solving $R^T Y = I$, we can use the fact that the solution Y is lower triangular. Therefore the kth column can be computed by a forward substitution of length $n - k + 1$, and the complexity of solving $R^T Y = I$ is

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

Second, the solution $X = A^{-1}$ is a symmetric matrix, so it is sufficient to compute its lower triangular part. When computing column k of X, we can stop the back substitution once X_{nk}, $X_{n-1,k}$, $\ldots$, X_{kk} have been computed, and this is a back substitution of length $n - k + 1$. The lower triangular part of the solution of $RX = Y$ can therefore be computed in roughly $(1/3)n^3$ operations. With these two improvements, the complexity of computing A^{-1} is roughly n^3 flops.

2.3 Computing the Cholesky factorization

Algorithm 2.2 below is a recursive method for computing the Cholesky factorization of a positive definite matrix A of order n. It refers to the following block matrix partitioning of A and its Cholesky factor R:

$$A = \begin{bmatrix} A_{11} & A_{1,2:n} \\ A_{2:n,1} & A_{2:n,2:n} \end{bmatrix}, \qquad R = \begin{bmatrix} R_{11} & R_{1,2:n} \\ 0 & R_{2:n,2:n} \end{bmatrix}$$

where A_{11} and R_{11} are scalars. By definition the Cholesky factor R is upper triangular with positive diagonal elements, so $R_{11} > 0$ and $R_{2:n,2:n}$ is upper triangular with positive diagonal elements.

Algorithm 2.2. CHOLESKY FACTORIZATION.

```
given a positive definite matrix $A$ of size $n \times n$.
  1. Calculate the first row of $R$:
       $R_{11} = \sqrt{A_{11}}, \qquad R_{1,2:n} = \frac{1}{R_{11}} A_{1,2:n}.$
  2. Compute the Cholesky factorization
       $A_{2:n,2:n} - R_{1,2:n}^T R_{1,2:n} = R_{2:n,2:n}^T R_{2:n,2:n}.$
```

It can be shown that the complexity of this algorithm is $(1/3)n^3$ flops.

To verify the correctness of the algorithm, we write out the equality $A = R^T R$ for the partitioned matrices:

$$\begin{aligned} \begin{bmatrix} A_{11} & A_{1,2:n} \\ A_{2:n,1} & A_{2:n,2:n} \end{bmatrix} &= \begin{bmatrix} R_{11} & 0 \\ R_{1,2:n}^T & R_{2:n,2:n}^T \end{bmatrix} \begin{bmatrix} R_{11} & R_{1,2:n} \\ 0 & R_{2:n,2:n} \end{bmatrix} \\ &= \begin{bmatrix} R_{11}^2 & R_{11} R_{1,2:n} \\ R_{11} R_{1,2:n}^T & R_{1,2:n}^T R_{1,2:n} + R_{2:n,2:n}^T R_{2:n,2:n} \end{bmatrix}. \end{aligned}$$

The equality allows us to determine the first row of R:

$$R_{11} = \sqrt{A_{11}}, \qquad R_{1,2:n} = \frac{1}{R_{11}} A_{1,2:n}.$$

(Recall that $A_{11} > 0$ if A is positive definite.) Furthermore,

$$R_{2:n,2:n}^T R_{2:n,2:n} = A_{2:n,2:n} - R_{1,2:n}^T R_{1,2:n},$$

so we must choose $R_{2:n,2:n}$ so that $R_{2:n,2:n}^T R_{2:n,2:n}$ is the Cholesky factorization of the matrix

$$A_{2:n,2:n} - R_{1,2:n}^T R_{1,2:n} = A_{2:n,2:n} - \frac{1}{A_{11}} A_{2:n,1} A_{2:n,1}^T.$$

This is the Schur complement of A_{11} and, as we have seen in section 2.1.2, it is a positive definite matrix of size $(n-1) \times (n-1)$.

If we continue recursively, we arrive at a Cholesky factorization of a 1×1 positive definite matrix, which is trivial (it is just the square root). Algorithm 2.2 therefore works for all positive definite matrices A. It also provides a practical way of checking whether a given matrix is positive definite. If we try to factor a matrix that is not positive definite, we will encounter a nonpositive leading diagonal element at some point during the recursion.

Example As an example, we derive the factorization (2.3).

$$\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix} = \begin{bmatrix} R_{11} & 0 & 0 \\ R_{12} & R_{22} & 0 \\ R_{13} & R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix}.$$

We start with the first row of R:

$$\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 \\ 3 & R_{22} & 0 \\ -1 & R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} 5 & 3 & -1 \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix}.$$

The remainder of the factorization follows from

$$\begin{aligned} \begin{bmatrix} 18 & 0 \\ 0 & 11 \end{bmatrix} &= \begin{bmatrix} 3 & R_{22} & 0 \\ -1 & R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} 3 & -1 \\ R_{22} & R_{23} \\ 0 & R_{33} \end{bmatrix} \\ &= \begin{bmatrix} 3 \\ -1 \end{bmatrix} \begin{bmatrix} 3 & -1 \end{bmatrix} + \begin{bmatrix} R_{22} & 0 \\ R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} R_{22} & R_{23} \\ 0 & R_{33} \end{bmatrix}, \\ \begin{bmatrix} 9 & 3 \\ 3 & 10 \end{bmatrix} &= \begin{bmatrix} R_{22} & 0 \\ R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} R_{22} & R_{23} \\ 0 & R_{33} \end{bmatrix}. \end{aligned}$$

To factor this 2×2-matrix, we first determine R_{22} and R_{23}:

$$\begin{bmatrix} 9 & 3 \\ 3 & 10 \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 1 & R_{33} \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 0 & R_{33} \end{bmatrix}.$$

Finally, the last element follows from

$$10 = 1 + R_{33}^2$$

i.e., $R_{33} = 3$.

2.4 Sparse positive definite matrices

If A is very sparse (most of its elements are zero), then the Cholesky factor R is usually quite sparse as well, and the Cholesky factorization can be computed in much less than $(1/3)n^3$ flops (provided we store A in a format that allows us to easily skip multiplications and additions with zero). If R is sparse, the associated forward and backward substitutions can also be carried out in much less than $2n^2$ flops.

The sparsity pattern of R (*i.e.*, the location of its zeros and nonzeros) can be determined from the sparsity pattern of A. Let us examine the first two steps of algorithm 2.2. The sparsity pattern of $R_{1,2:n}$, computed in the first step, is exactly the same as the sparsity pattern of $A_{1,2:n}$. In step 2, the matrix $A_{2:n,2:n} - R_{1,2:n}^T R_{1,2:n}$ is formed, and this matrix will usually have some nonzero elements in positions where $A_{2:n,2:n}$ is zero. More precisely, the i, j element of $A_{2:n,2:n} -$

$R_{1,2:n}^T R_{1,2:n}$ can be nonzero if the i, j element of $A_{2:n,2:n}$ is nonzero, but also if the ith and the jth elements of $R_{1,2:n}$ are nonzero. If we then proceed to the next cycle in the recursion and determine the first column of the Cholesky factor of $A_{2:n,2:n} - R_{1,2:n}^T R_{1,2:n}$, some nonzero elements may be introduced in positions of R where the original matrix A had a zero element. The creation of new nonzero elements in the sparsity pattern of R, compared to the sparsity pattern of A, is called *fill-in*. If the amount of fill-in is small, R will be sparse, and the Cholesky factorization can be computed very efficiently.

In some cases the fill-in is very extensive, or even complete. Consider for example, the set of linear equations

$$\begin{bmatrix} 1 & a^T \\ a & I \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} b \\ c \end{bmatrix} \tag{2.4}$$

where a is an n-vector with $\|a\| < 1$. It can be shown that the coefficient matrix is positive definite (see exercises). Its Cholesky factorization is

$$\begin{bmatrix} 1 & a^T \\ a & I \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a & R_{2:n,2:n}^T \end{bmatrix} \begin{bmatrix} 1 & a^T \\ 0 & R_{2:n,2:n} \end{bmatrix}$$

where $I - aa^T = R_{2:n,2:n}^T R_{2:n,2:n}$ is the factorization of $I - aa^T$. Now if all the elements of a are nonzero, then $I - aa^T$ is a dense matrix. Therefore $R_{2:n,2:n}$ will be dense and we have 100% fill-in.

If, on the other hand, we first rewrite the equations (2.4) as

$$\begin{bmatrix} I & a \\ a^T & 1 \end{bmatrix} \begin{bmatrix} v \\ u \end{bmatrix} = \begin{bmatrix} c \\ b \end{bmatrix}$$

(by reordering the variables and right-hand sides) and then factor the reordered coefficient matrix, we obtain a factorization with zero fill-in:

$$\begin{bmatrix} I & a \\ a^T & 1 \end{bmatrix} = \begin{bmatrix} I & 0 \\ a^T & \sqrt{1 - a^T a} \end{bmatrix} \begin{bmatrix} I & a \\ 0 & \sqrt{1 - a^T a} \end{bmatrix}.$$

This simple example shows that reordering the equations and variables can have a dramatic effect on the amount of fill-in, and hence on the efficiency of solving a sparse set of equations.

Sparse Cholesky factorization When A is symmetric positive definite and sparse, it is usually factored as

$$A = PR^T RP^T \tag{2.5}$$

where P is a permutation matrix and R is upper triangular with positive diagonal elements. We can express this as $P^T AP = R^T R$, *i.e.*, $R^T R$ is the Cholesky factorization of $P^T AP$, the matrix A with its rows and columns permuted by the same permutation.

The matrix $P^T AP$ arises when we reorder the equations and the variables in $Ax = b$. If we define new variables $\tilde{x} = P^T x$ and apply the same permutation to the right-hand side to get $\tilde{b} = P^T b$, then $\tilde{x}$ satisfies the equation $(P^T AP)\tilde{x} = \tilde{b}$.

Since $P^T AP$ is positive definite for any permutation matrix P, we are free to choose any permutation matrix; for every P there is a unique Cholesky factor R. The choice of P, however, can greatly affect the sparsity of R, which in turn can greatly affect the efficiency of solving $Ax = b$. Very effective heuristic methods are known to select a permutation matrix P that leads to a sparse factor R.

Chapter 3

Complexity of iterative algorithms

3.1 Iterative algorithms

The matrix algorithms we discussed so far are non-iterative. They require a finite number of floating-point operations, that can be counted and expressed as a polynomial function of the problem dimensions. In chapters 4 and 5 we discuss problems that are solved by *iterative* algorithms. By this is meant an algorithm that computes a sequence of values $x^{(0)},\ x^{(1)}, x^{(2)}, \ldots$, with

$$x^{(k)} \to x^*$$

as $k \to \infty$, where the scalar or vector x^* is a solution of the problem. $x^{(0)}$ is called the *starting point* of the algorithm, and $x^{(k)}$ is called the kth *iterate*. Moving from $x^{(k)}$ to $x^{(k+1)}$ is called an *iteration* of the algorithm. The algorithm is terminated when $\|x^{(k)} - x^*\| \leq \epsilon$, where $\epsilon > 0$ is some specified tolerance, or when it is determined that the sequence is not converging (for example, when a limit on the number of iterations is exceeded).

The total cost of an iterative algorithm is more difficult to estimate than the cost of a non-iterative algorithm, because the number of iterations depends on the problem parameters and on the starting point. The efficiency of an iterative algorithm is therefore usually not expressed by giving its flop count, but by giving upper bounds on the number of iterations to reach a given accuracy. Deriving such bounds is the purpose of *convergence analysis.*

Iterative algorithms are often classified according to their rate of convergence. In the following paragraphs we give an overview of the most common definitions. For simplicity we will assume x^* is a scalar, and $x^{(k)}$ $(k = 0, 1, 2, \ldots)$ is a sequence of numbers converging to $x^\star$.

Absolute and relative error The error after k iterations can be expressed as the *absolute error*, $|x^{(k)} - x^\star|$, or as the *relative error* $|x^{(k)} - x^\star|/|x^\star|$ (which is defined only if $x^\star \neq 0$). The relative error can also be expressed as the *number of correct* *digits*, for which several slightly different definitions exist. The definition that we will adopt is the following: if the relative error $|x^{(k)} - x^\star|/|x^\star|$ is less than one, then the number of correct digits in $x^{(k)}$ is defined as

$$\left\lfloor -\log_{10}\left(\frac{|x^{(k)} - x^\star|}{|x^\star|}\right)\right\rfloor$$

(by $\lfloor \alpha \rfloor$ we mean the largest integer less than or equal to α). In other words, we take the logarithm with base 10 of the relative error (which is a negative number if the relative error is less than one), change its sign, and round it down to the nearest integer. This means that r correct digits correspond to relative errors in the interval $[10^{-r}, 10^{-r-1})$.

As an example, the table shows three numbers x close to $x^\star = \pi = 3.141592\ldots$, the relative error $|x - \pi|/\pi$, and the number of correct digits $\lfloor -\log_{10}(|x - \pi|/\pi)\rfloor$.

$x^{(k)}$	rel. error	#correct digits
3.14150	$2.9 \cdot 10^{-5}$	4
3.14160	$2.3 \cdot 10^{-6}$	5
3.14165	$1.8 \cdot 10^{-5}$	4

Note that other, perhaps more intuitive, definitions of correct digits might give slightly different results. For example, if we simply count the number of leading digits where π and x agree, we would say there are 5 correct digits in 3.14150, and 4 correct digits in 3.14160 and 3.14165. On the other hand, we might also argue that the number of correct digits in $x = 3.14150$ is actually 4, because if we round x and π to 4 digits, we obtain the same number (3.142), but if we round them to 5 digits, they are different (3.1415 and 3.1416).

These discrepancies do not matter in practice. If we use the number of correct digits to describe the accuracy of an approximation, it is understood that we are only giving a rough indication of the relative error. “Four correct digits” means that the relative error is roughly 10^{-4}. It could be a little more or little less, depending on the definition, but that does not matter, because if we want to say exactly what the accuracy is, we can simply give the relative (or absolute) error.

3.2 Linear and R-linear convergence

A sequence $x^{(k)}$ with limit $x^\star$ is *linearly convergent* if there exists a constant $c \in (0, 1)$ such that

$$|x^{(k)} - x^\star| \leq c\,|x^{(k-1)} - x^\star| \tag{3.1}$$

for k sufficiently large. For example, the sequence $x^{(k)} = 1 + (1/2)^k$ converges linearly to $x^\star = 1$, because

$$|x^{(k+1)} - x^\star| = (1/2)^{k+1} = \frac{1}{2}|x^{(k)} - x^\star|$$

so the definition is satisfied with $c = 1/2$.

k	$1 + 0.5^k$
0	2.00000000000000
1	1.50000000000000
2	1.25000000000000
3	1.12500000000000
4	1.06250000000000
5	1.03125000000000
6	1.01562500000000
7	1.00781250000000
8	1.00390625000000
9	1.00195313125000
10	1.00097656250000

Table 3.1 The first ten values of $x^{(k)} = 1 + 1/2^k$.

If $x^\star \neq 0$, we can give an intuitive interpretation of linear convergence in terms of the number of correct digits in $x^{(k)}$. Let

$$r^{(k)} = -\log_{10} \frac{|x^{(k)} - x^\star|}{|x^\star|}. \tag{3.2}$$

Except for rounding to an integer, $r^{(k)}$ is the number of correct digits in $x^{(k)}$. If we divide both sides of the inequality (3.1) by $|x^\star|$ and take logarithms, we obtain

$$r^{(k+1)} \geq r^{(k)} - \log_{10} c.$$

Ignoring the effect of rounding, we can say we gain at least $-\log_{10} c$ correct digits per iteration.

We can verify this using the example $x^{(k)} = 1 + 1/2^k$. As we have seen, this sequence is linearly convergent with $c = 1/2$, so we expect to gain roughly $-\log_{10} 1/2 = 0.3$ correct digits per iteration, or in other words, one correct digit per three or four iterations. This is confirmed by table 3.1, which shows the first ten values of $x^{(k)}$.

R-linear convergence Linear convergence is also sometimes defined as follows. A sequence $x^{(k)}$ with limit $x^\star$ is *R-linearly convergent* if there exists a positive M and $c \in (0, 1)$ such that

$$|x^{(k)} - x^\star| \leq M c^k \tag{3.3}$$

for sufficiently large k. This means that for large k the error decreases at least as fast as the geometric series Mc^k. We refer to this as R-linear convergence to distinguish it from the first definition. Every linearly convergent sequence is also R-linearly convergent, but the converse is not true. For example, the error in an R-linearly convergent sequence does not necessarily decrease monotonically, while the inequality (3.1) implies that $|x^{(k)} - x^\star| < |x^{(k-1)} - x^\star|$ for sufficiently large k.

k	$1 + 0.5^{2^k}$
0	1.50000000000000
1	1.25000000000000
2	1.06250000000000
3	1.00390625000000
4	1.00001525878906
5	1.00000000023283
6	1.00000000000000

Table 3.2 The first six values of $x^{(k)} = 1 + (1/2)^{2^k}$.

3.3 Quadratic convergence

A sequence $x^{(k)}$ with limit $x^\star$ is *quadratically convergent* if there exists a constant $c > 0$ such that

$$|x^{(k)} - x^\star| \leq c\, |x^{(k-1)} - x^\star|^2 \tag{3.4}$$

for k sufficiently large. The sequence $x^{(k)} = 1 + (1/2)^{2^k}$ converges quadratically to $x^\star = 1$, because

$$|x^{(k+1)} - x^\star| = (1/2)^{2^{k+1}} = \left((1/2)^{2^k}\right)^2 = |x^{(k)} - x^\star|^2,$$

so the definition is satisfied with $c = 1$.

If $x^\star \neq 0$, we can relate the definition to the number of correct digits in $x^{(k)}$. If we define $r^{(k)}$ as in (3.2), then we can write the inequality (3.4) as

$$r^{(k)} \geq 2r^{(k-1)} - \log_{10}(|x^\star c|).$$

Since $|x^{(k)} - x^\star| \to 0$, we must have $r^{(k)} \to +\infty$, so sooner or later the first term on the right-hand side will dominate the second term, which is constant. For sufficiently large k, the number of correct digits roughly doubles in each iteration.

Table 3.2 shows the first few values of the sequence $x^{(k)} = 1 + (1/2)^{2^k}$ which converges quadratically with $c = 1$, and $x^\star = 1$. We start with one correct digit. It takes two iterations to get the second correct digit. The next iteration we gain one digit, then we gain two in one iteration, etc.

3.4 Superlinear convergence

A sequence $x^{(k)}$ with limit $x^\star$ is *superlinearly convergent* if there exists a sequence $c_k > 0$ with $c_k \to 0$ such that

$$|x^{(k)} - x^\star| \leq c_k\, |x^{(k-1)} - x^\star| \tag{3.5}$$

for sufficiently large k.

The sequence $x^{(k)} = 1 + (1/(k+1))^k$ is superlinearly convergent because

$$|x^{(k)} - x^\star| = \frac{1}{(k+1)^k} = \frac{k^{k-1}}{(k+1)^k}\frac{1}{k^{k-1}} = \frac{k^{k-1}}{(k+1)^k}|x^{(k-1)} - x^\star|,$$

so the definition is satisfied with $c_k = k^{k-1}/(k+1)^k$, which indeed goes to zero.

If we define $r^{(k)}$ as above, we can write the inequality (3.4) as

$$r^{(k)} \geq r^{(k-1)} - \log_{10}(c_k),$$

and since $c_k \to 0$, $-\log_{10} c_k \to \infty$. For sufficiently large k, the number of correct digits gained per iteration ($-\log_{10}(c_k)$) increases with k.

Table 3.3 shows the first values of $1+(1/(k+1))^k$. The number of correct digits increases faster than linearly, but does not quite double per iteration.

k	$1 + (1/(k+1)^k)$
0	2.00000000000000
1	1.50000000000000
2	1.11111111111111
3	1.01562500000000
4	1.00160000000000
5	1.00012860082305
6	1.00000849985975
7	1.00000047683716
8	1.00000002323057
9	1.00000000100000
10	1.00000000003855

Table 3.3 The first ten values of $x^{(k)} = 1 + (1/(k+1))^k$.

Chapter 4

Nonlinear equations

In this chapter we discuss methods for finding a solution of n nonlinear equations in n variables

$$\begin{array}{rcl} f_1(x_1, x_2, \ldots, x_n) & = & 0 \\ f_2(x_1, x_2, \ldots, x_n) & = & 0 \\ & \vdots & \\ f_n(x_1, x_2, \ldots, x_n) & = & 0. \end{array}$$

To simplify notation, we will often express this as

$$f(x) = 0$$

where

$$f(x) = \left[\begin{array}{c} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{array} \right], \qquad x = \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right].$$

We assume that f is at least continuous (*i.e.*, $\lim_{y \to x} f(y) = f(x)$ for all x), and for some algorithms, stronger assumptions will be needed (for example, differentiability).

4.1 Bisection method

The first method we discuss only applies to problems with $n = 1$.

We start with an interval $[l, u]$ that satisfies $f(l)f(u) < 0$ (the function values at the end points of the interval have opposite signs). Since f is continuous, this guarantees that the interval contains at least one solution of $f(x) = 0$. In each iteration we evaluate f at the midpoint $(l + u)/2$ of the interval, and depending on the sign of $f((l + u)/2)$, replace l or u with $(l + u)/2$. If $f((u + l)/2)$ has the same sign as $f(l)$, we replace l with $(u+l)/2$. Otherwise we replace u. Thus we obtain a new interval that still satisfies $f(l)f(u) < 0$. The method is called bisection because the interval is replaced by either its left or right half at each iteration.

Algorithm 4.1. Bisection algorithm.

```
given $l$, $u$ with $l < u$ and $f(l)f(u) < 0$; a required tolerance $\epsilon > 0$
repeat
    1. $x := (l+u)/2$.
    2. Compute $f(x)$.
    3. if $f(x) = 0$, return $x$.
    4. if $f(x)f(l) < 0$, $u := x$, else, $l := x$.
until $u - l \leq \epsilon$
```

The convergence of the bisection method is easy to analyze. If we denote by $[l^{(0)}, u^{(0)}]$ the initial interval, and by $[l^{(k)}, u^{(k)}]$ the interval after iteration k, then

$$u^{(k)} - l^{(k)} = \frac{u^{(0)} - l^{(0)}}{2^k}, \tag{4.1}$$

because the length of the interval is divided by two at each iteration. This means that the exit condition $u^{(k)} - l^{(k)} \leq \epsilon$ will be satisfied if

$$\log_2(\frac{u^{(0)} - l^{(0)}}{2^k}) = \log_2(u^{(0)} - l^{(0)}) - k \leq \log_2 \epsilon,$$

i.e., as soon as $k \geq \log_2((u^{(0)} - l^{(0)})/\epsilon)$. The algorithm therefore terminates after

$$\left\lceil \log_2 \left(\frac{u^{(0)} - l^{(0)}}{\epsilon} \right) \right\rceil$$

iterations. (By $\lceil \alpha \rceil$ we mean the smallest integer greater than or equal to α.)

Since the final interval contains at least one solution $x^\star$, we are guaranteed that its midpoint

$$x^{(k)} = \frac{1}{2}(l^{(k)} + u^{(k)})$$

is no more than a distance $u^{(k)} - l^{(k)}$ from $x^\star$. Thus we have

$$|x^{(k)} - x^\star| \leq \epsilon,$$

when the algorithm terminates.

The advantages of the bisection method are its simplicity and the fact that it does not require derivatives. It also does not require a starting point close to $x^\star$. The disadvantages are that it is not very fast, and that it does not extend to $n > 1$. Selecting an initial interval that satisfies $f(l)f(u) < 0$ may also be difficult.

Convergence rate The bisection method is *R-linearly convergent* (as defined in section 3.2). After k iterations, the midpoint $x^{(k)} = (u^{(k)} + l^{(k)})/2$ satisfies

$$|x^{(k)} - x^\star| \leq u^{(k)} - l^{(k)} \leq (1/2)^k (u^{(0)} - l^{(0)}),$$

(see equation (4.1)). Therefore the definition (3.3) is satisfied with $c = 1/2$ and $M = u^{(0)} - l^{(0)}$.

4.2 Newton's method for one equation with one variable

Newton's method is the most popular method for solving nonlinear equations. We first explain the method for $n = 1$, and then extend it to $n > 1$. We assume that f is differentiable.

Algorithm 4.2. Newton's method for one equation with one variable.

```
given initial $x$, required tolerance $\epsilon > 0$
repeat
    1. Compute $f(x)$ and $f'(x)$.
    2. if $|f(x)| \leq \epsilon$, return $x$.
    3. $x := x - f(x)/f'(x)$.
until maximum number of iterations is exceeded.
```

For simplicity we assume that $f'(x) \neq 0$ in step 3. (In a practical implementation we would have to make sure that the code handles the case $f'(x) = 0$ gracefully.)

The algorithm starts at some initial value $x^{(0)}$, and then computes iterates $x^{(k)}$ by repeating

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}, \quad k = 0, 1, 2, \ldots.$$

This update has a simple interpretation. After evaluating the function value $f(x^{(k)})$ and the derivative $f'(x^{(k)})$, we construct the first-order Taylor approximation to f around $x^{(k)}$:

$$\hat{f}(y) = f(x^{(k)}) + f'(x^{(k)})(y - x^{(k)}).$$

We then solve $\hat{f}(y) = 0$, *i.e.*,

$$f(x^{(k)}) + f'(x^{(k)})(y - x^{(k)}) = 0,$$

for the variable y. This is called the *linearized equation*. If $f'(x^{(k)}) \neq 0$, the solution exists and is given by

$$y = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}.$$

We then take y as the next value $x^{(k+1)}$. This is illustrated in figure 4.1.

Examples We first consider the nonlinear equation

$$f(x) = e^x - e^{-x} - 1 = 0.$$

The derivative is $f'(x) = e^x + e^{-x}$, so the Newton iteration is

$$x^{(k+1)} = x^{(k)} - \frac{e^{x^{(k)}} - e^{-x^{(k)}} - 1}{e^{x^{(k)}} + e^{-x^{(k)}}}, \quad k = 0, 1, \ldots$$

If we start at $x = 4$, the algorithm converges very quickly to $x^\star = 0.4812$ (see figure 4.2).

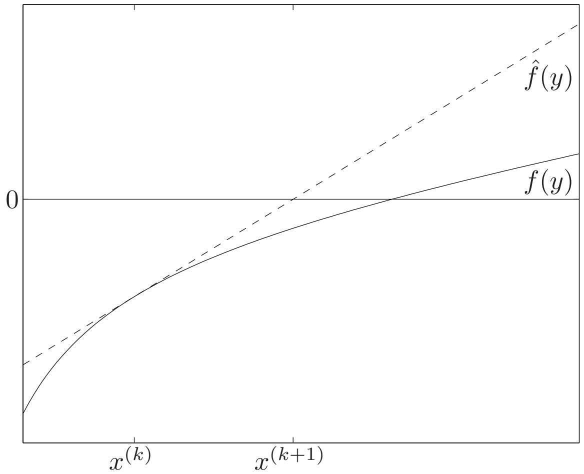

Figure 4.1 One iteration of Newton's method. The iterate $x^{(k+1)}$ is the zero-crossing of the first-order Taylor approximation of f at $x^{(k)}$.

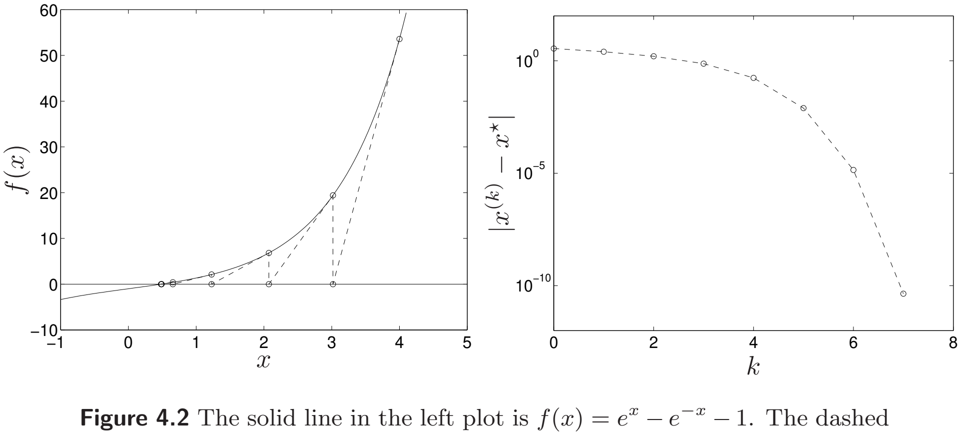

Figure 4.2 The solid line in the left plot is $f(x) = e^x - e^{-x} - 1$. The dashed line and the circles indicate the iterates in Newton's method for solving $f(x) = 0$, starting at $x^{(0)} = 4$. The right plot shows the error $|x^{(k)} - x^\star|$ versus k.

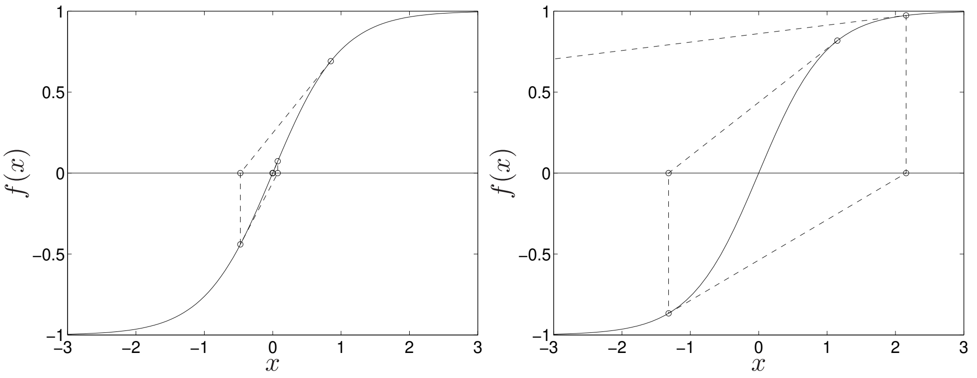

Figure 4.3 The solid line in the left plot is $f(x) = (e^x - e^{-x})/(e^x + e^{-x})$. The dashed line and the circles indicate the iterates in Newton's method for solving $f(x) = 0$, starting at $x^{(0)} = 0.85$ (left) and $x^{(0)} = 1.15$ (right). In the first case the method converges rapidly to $x^\star = 0$. In the second case it does not converge.

As a second example, we consider the equation

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 0.$$

The derivative is $f'(x) = 4/(e^x + e^{-x})^2$, so the Newton iteration is

$$x^{(k+1)} = x^{(k)} - \frac{1}{4}(e^{2x^{(k)}} - e^{-2x^{(k)}}), \quad k = 0, 1, \ldots$$

Figure 4.3 shows the iteration, starting at two starting points, $x^{(0)} = 0.85$, and $x^{(0)} = 1.15$. The method converges rapidly from $x^{(0)} = 0.85$, but does not converge from $x^{(0)} = 1.15$.

The two examples are typical for the convergence behavior of Newton's method: it works very well if started near a solution; it may not work when started far from a solution.

4.3 Newton's method for sets of nonlinear equations

We now extend Newton's method to a nonlinear equation $f(x) = 0$ where $f : \mathbf{R}^n \to \mathbf{R}^n$ (a function that maps an n-vector x to an n-vector $f(x)$).

We start with an initial point $x^{(0)}$. At iteration k we evaluate $f(x^{(k)})$, the derivative matrix $Df(x^{(k)})$, and form the first-order approximation of f at $x^{(k)}$:

$$\hat{f}(y) = f(x^{(k)}) + Df(x^{(k)})(y - x^{(k)}).$$

We set $\hat{f}(y) = 0$ and solve for y, which gives

$$y = x^{(k)} - Df(x^{(k)})^{-1} f(x^{(k)})$$

(assuming $Df(x^{(k)})$ is nonsingular). This value is taken as the next iterate $x^{(k+1)}$. In summary,

$$x^{(k+1)} = x^{(k)} - Df(x^{(k)})^{-1} f(x^{(k)}), \quad k = 0, 1, 2, \ldots$$

Algorithm 4.3. Newton's method for sets of nonlinear equations.

```
given an initial $x$, a required tolerance $\epsilon > 0$
repeat
    1. Evaluate $g = f(x)$ and $H = Df(x)$.
    2. if $\|g\| \leq \epsilon$, return $x$.
    3. Solve $Hv = -g$.
    4. $x := x + v$.
until maximum number of iterations is exceeded.
```

Example As an example, we take a problem with two variables

$$f_1(x_1, x_2) = \log(x_1^2 + 2x_2^2 + 1) - 0.5, \qquad f_2(x_1, x_2) = -x_1^2 + x_2 + 0.2. \tag{4.2}$$

There are two solutions, $(0.70, 0.29)$ and $(-0.70, 0.29)$. The derivative matrix is

$$Df(x) = \left[\begin{array}{cc} 2x_1/(x_1^2 + 2x_2^2 + 1) & 4x_2/(x_1^2 + 2x_2^2 + 1) \\ -2x_1 & 1 \end{array}\right].$$

Figure 4.4 shows what happens if we use three different starting points.

The example confirms the behavior we observed for problems with one variable. Newton's method does not always work, but if started near a solution, it takes only a few iterations. The main advantage of the method is its very fast local convergence. The disadvantages are that it requires a good starting point, and that it requires the n^2 partial derivatives of f.

Many techniques have been proposed to improve the convergence properties. (A simple idea for $n = 1$ is to combine it with the bisection method.) These *globally convergent* Newton methods are designed in such a way that locally, in the neighborhood of a solution, they automatically switch to the standard Newton method.

A variation on Newton's method that does not require derivatives is the secant method, discussed in the next paragraph.

4.4 Secant method

Although the idea of the secant method extends to problems with several variables, we will describe it only for $n = 1$.

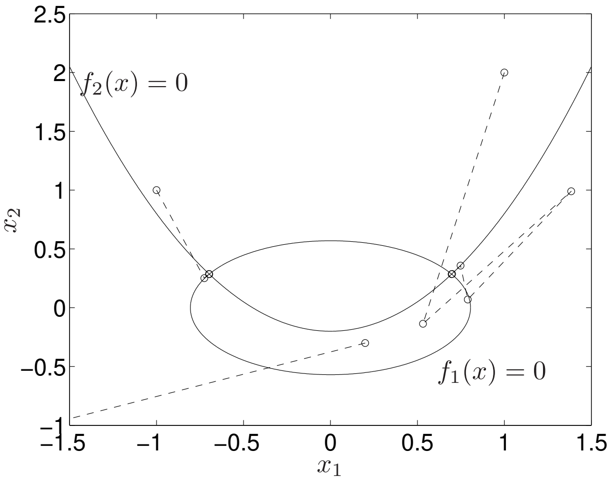

Figure 4.4 The solid lines are the zero-level curves of the functions f_1 and f_2 in (4.2). The circles are the iterates of Newton's method from three starting points. With $x^{(0)} = (-1, 1)$, the method converges to the solution $(-0.70, 0.29)$ in 2 or 3 steps. With $x^{(0)} = (1, 2)$, it converges to the solution $(0.70, 0.29)$ in about 5 steps. With $x^{(0)} = (0.2, -0.3)$, it does not converge.

The secant method can be interpreted as a variation of Newton's method in which we replace the first-order approximation

$$\hat{f}(y) = f(x^{(k)}) + f'(x^{(k)})(y - x^{(k)})$$

with the function

$$\hat{f}(y) = f(x^{(k)}) + \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}(y - x^{(k)}).$$

This is the affine function that agrees with $f(y)$ at $y = x^{(k)}$ and $y = x^{(k-1)}$. We then set $\hat{f}(y)$ equal to zero, solve for y, and take the solution as $x^{(k+1)}$ (figure 4.5).

```
Algorithm 4.4. SECANT METHOD FOR ONE EQUATION WITH ONE VARIABLE.
given two initial points $x$, $x^-$, required tolerance $\epsilon > 0$
repeat
    1. Compute $f(x)$
    2. if $|f(x)| \leq \epsilon$, return $x$.
    3. $g := (f(x) - f(x^-))/(x - x^-)$.
    4. $x^- := x$.
    5. $x := x - f(x)/g$.
until maximum number of iterations is exceeded.
```

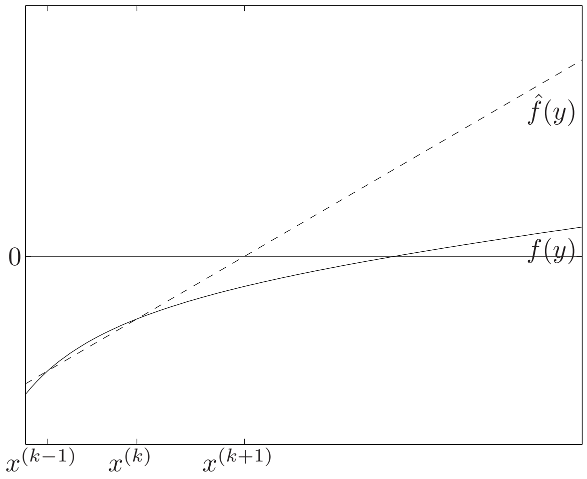

Figure 4.5 One iteration of the secant method. The iterate $x^{(k+1)}$ is the zero-crossing of the affine function through the points $(x^{(k-1)}, f(x^{(k-1)})$ and $(x^{(k)}, f(x^{(k)})$.

The convergence of the secant method is slower than the Newton method, but it does not require derivatives, so the amount of work per iteration is smaller.

Example

Figure 4.6 shows the convergence of the secant method for the same example as in figure 4.2, with starting points $x^{(0)} = 4$, $x^{(-1)} = 4.5$.

4.5 Convergence analysis of Newton's method

Newton's method converges quadratically if $f'(x^\star) \neq 0$, and we start sufficiently close to $x^\star$. More precisely we can state the following result.

Suppose $I = [x^\star - \delta, x^\star + \delta]$, with $\delta > 0$, is an interval around a solution $x^\star$ on which f satisfies the following two properties.

1. There exists a constant $m > 0$ such that $|f'(x)| \geq m$ for all $x \in I$.
2. There exists a constant $L > 0$ such that $|f'(x) - f'(y) \leq L|x - y|$ for all $x, y \in I$.

We will show that if $|x^{(k)} - x^\star| \leq \delta$, then

$$|x^{(k+1)} - x^\star| \leq \frac{L}{2m}|x^{(k)} - x^\star|^2. \tag{4.3}$$

In other words, the inequality (3.4) is satisfied with $c = L/(2m)$, so if $x^{(k)}$ converges to $x^\star$, it converges quadratically.

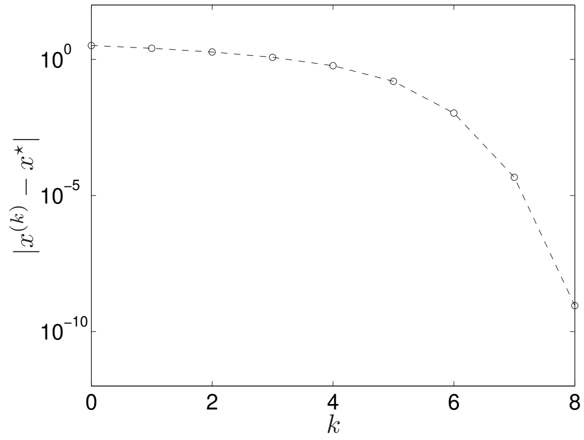

Figure 4.6 Error $|x^{(k)} - x^\star|$ versus k for the secant method applied to the equation $e^x - e^{-x} - 1 = 0$, with starting points $x^{(0)} = 4$, $x^{(-1)} = 4.5$.

The inequality (4.3) is proved as follows. For simplicity we denote $x^{(k)}$ by x and $x^{(k+1)}$ by x^+, *i.e.*,

$$x^+ = x - \frac{f(x)}{f'(x)}.$$

We have

$$\begin{aligned} |x^+ - x^\star| &= |x - \frac{f(x)}{f'(x)} - x^\star| \\ &= \frac{|-f(x) - f'(x)(x^\star - x)|}{|f'(x)|} \\ &= \frac{|f(x^\star) - f(x) - f'(x)(x^\star - x)|}{|f'(x)|}. \end{aligned}$$

(Recall that $f(x^\star) = 0$.) We have $|f'(x)| \geq m$ by the first property, and hence

$$|x^+ - x^\star| \leq \frac{|f(x^\star) - f(x) - f'(x)(x^\star - x)|}{m}. \tag{4.4}$$

We can use the second property to bound the numerator:

$$\begin{aligned} |f(x^\star) - f(x) - f'(x)(x^\star - x)| &= \left| \int_x^{x^\star} (f'(u) - f'(x))\, du \right| \\ &\leq \int_x^{x^\star} |f'(u) - f'(x)|\, du \\ &\leq L \int_x^{x^\star} |u - x|\, du \\ &= \frac{L}{2}|x^\star - x|^2. \end{aligned} \tag{4.5}$$

Putting (4.4) and (4.5) together, we obtain $|x^+ - x| \leq L|x^\star - x|^2/(2m)$, which proves (4.3).

The inequality (4.3) by itself does not guarantee that $|x^{(k)} - x^\star| \to 0$. However, if we assume that $|x^{(0)} - x^\star| \leq \delta$ and that $\delta \leq m/L$, then convergence readily follows. Since $|x^{(0)} - x^\star| \leq \delta$, we can apply the inequality (4.3) to the first iteration, which yields

$$|x^{(1)} - x^\star| \leq \frac{L}{2m}|x^{(0)} - x^\star|^2 \leq \frac{L}{2m}\delta^2 \leq \frac{\delta}{2}.$$

Therefore $|x^{(1)} - x^\star| \leq \delta$, so we can apply the inequality to $k = 1$, and obtain a bound on the error in $x^{(2)}$,

$$|x^{(2)} - x^\star| \leq \frac{L}{2m}|x^{(1)} - x^\star|^2 \leq \frac{L}{2m}\frac{\delta^2}{4} \leq \frac{\delta}{8},$$

and therefore also

$$|x^{(3)} - x^\star| \leq \frac{L}{2m}|x^{(2)} - x^\star|^2 \leq \frac{L}{2m}\frac{\delta^2}{8^2} \leq \frac{\delta}{128}$$

et cetera. Continuing in this fashion, we have

$$|x^{(k+1)} - x^\star| \leq \frac{L}{2m}|x^{(k)} - x^\star|^2 \leq 2\left(\frac{1}{4}\right)^{2^k}\delta,$$

which shows that the error converges to zero very rapidly.

A final note on the practical importance of this (and most other) convergence results. If $f'(x^\star) \neq 0$ and $f'(x)$ is a continuous function, then it is reasonable to assume that the assumptions we made are satisfied for *some* δ, m, and L. In practice, of course, we almost never know δ, m, L, so the convergence result does not provide any practical guidelines that might help us, for example, when selecting a starting point.

The result does provide some interesting qualitative or conceptual information. It establishes convergence of Newton's method, provided we start sufficiently close to a solution. It also explains the very fast local convergence observed in practice. As an interesting detail that we have not observed so far, the proof suggests that quadratic convergence only occurs if $f'(x^\star) \neq 0$. A simple example will confirm this. Suppose we apply Newton's method to the nonlinear equation

$$f(x) = x^2 - a = 0,$$

where a is a nonnegative number. Newton's method uses the iteration

$$\begin{aligned} x^{(k+1)} &= x^{(k)} - \frac{(x^{(k)})^2 - a}{2x^{(k)}} \\ &= \frac{1}{2}(x^{(k)} + \frac{a}{x^{(k)}}), \quad k = 0, 1, 2, \ldots. \end{aligned}$$

Figures 4.7 and 4.8 show the result for $a = 5$ and $a = 0$ respectively, with starting point $x^{(0)} = 5$. Newton's method converges in both cases, but much more slowly when $a = 0$ (and hence $f'(x^\star) = 0$).

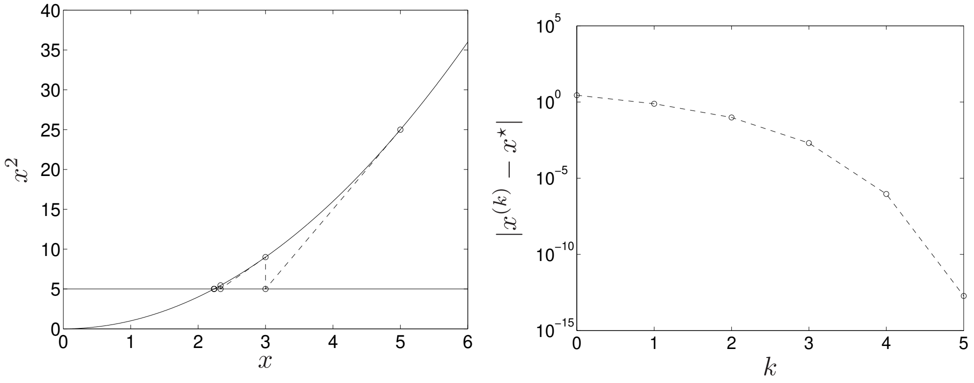

Figure 4.7 The solid line on the left is the function x^2. The circles and dashed lines show the iterates in Newton's method applied to the function $x^2 - 5 = 0$. The right plot shows the error versus k.

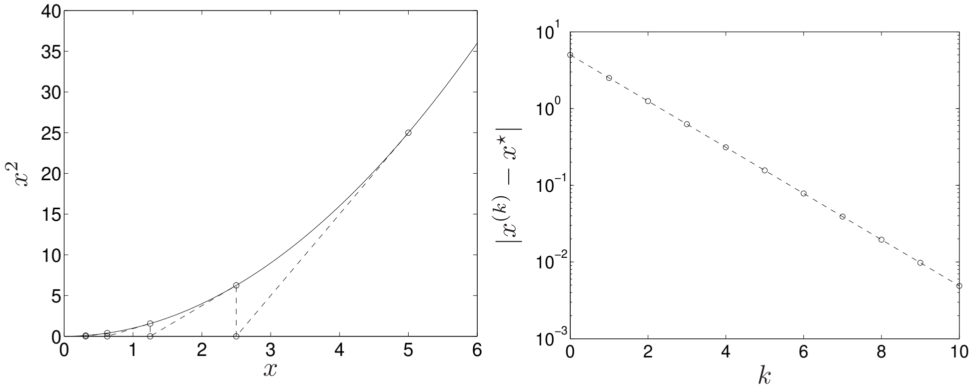

Figure 4.8 The solid line on the left is the function x^2. The circles and dashed lines show the iterates in Newton's method applied to the function $x^2 = 0$. The right plot shows the error versus k.

Sets of equations The quadratic convergence result for Newton's method generalizes to functions of several variables. The precise statement is as follows. Suppose there is a neighborhood

$$I = \{x \mid \|x - x^\star\| \leq \delta\}$$

around a solution $x^\star$ on which f satisfies the following two properties.

1. There exists a constant $m > 0$ such that $\|Df(x)^{-1}\|_2 \leq 1/m$ for all $x \in I$.
2. There exists a constant $L > 0$ such that $\|Df(x) - Df(y)\|_2 \leq L\|x - y\|$ for all $x, y \in I$.

(Note that the norms $\|Df(x)^{-1}\|_2$ and $\|Df(x) - Df(y)\|_2$ are matrix norms, defined in §6.2, and $\|x - y\|$ is a vector norm.) If $\|x^{(k)} - x^\star\| \leq \delta$, then

$$\|x^{(k+1)} - x^\star\| \leq \frac{L}{2m}\|x^{(k)} - x^\star\|^2.$$

Secant method Under similar assumptions as Newton's method (including, in particular, $f'(x^\star) \neq 0$ and a starting point sufficiently close to $x^\star$), the secant method converges superlinearly. We omit the precise statement and the proof.

Chapter 5

Unconstrained minimization

5.1 Introduction

Let $g : \mathbf{R}^n \to \mathbf{R}$ be a scalar-valued function of n variables $x = (x_1, x_2, \ldots, x_n)$. We say $x^\star = (x_1^\star, x_2^\star, \ldots, x_n^\star)$ *minimizes* g if $g(x^\star) \leq g(x)$ for all n-vectors x. We use the notation

$$\text{minimize} \quad g(x)$$

to denote the problem of finding an $x^\star$ that minimizes g. This is called an *unconstrained minimization problem*, with variables $x_1, \ldots, x_n$, and with *objective function* or *cost function* g.

If $x^\star$ minimizes g, then we say $x^\star$ is a *solution* of the minimization problem, or a *minimum* of g. A minimum is also sometimes referred to as a *global* minimum. A vector $x^\star$ is a *local minimum* if there exists an $R > 0$ such that $g(x^\star) \leq g(x)$ for all x with $\|x - x^\star\| \leq R$. In other words, there is a neighborhood around $x^\star$ in which $g(x) \geq g(x^\star)$. In this chapter, minimum means global minimum. The word "global" in "global minimum" is redundant, but is sometimes added for emphasis.

The greatest α such that $\alpha \leq g(x)$ for all x is called the *optimal value* of the minimization problem, and denoted

$$\min g(x)$$

or $\min_x g(x)$. If $x^\star$ is a minimum of g, then $g(x^\star) = \min g(x)$, and we say that the optimal value is *attained* at $x^\star$. It is possible that $\min g(x)$ is finite, but there is no $x^\star$ with $g(x^\star) = \min g(x)$ (see the examples below). In that case the optimal value is not attained. It is also possible that $g(x)$ is unbounded below, in which case we define the optimal value as $\min g(x) = -\infty$.

Examples We illustrate the definitions with a few examples with one variable.

- $g(x) = (x-1)^2$. The optimal value is $\min g(x) = 0$, and is attained at the (global) minimum $x^\star = 1$. There are no other local minima.
- $g(x) = e^x + e^{-x} - 3x^2$, shown in the left-hand plot of figure 5.1. The optimal value is -7.02. There are two (global) minima, at $x^\star = \pm 2.84$. There are no other local minima.

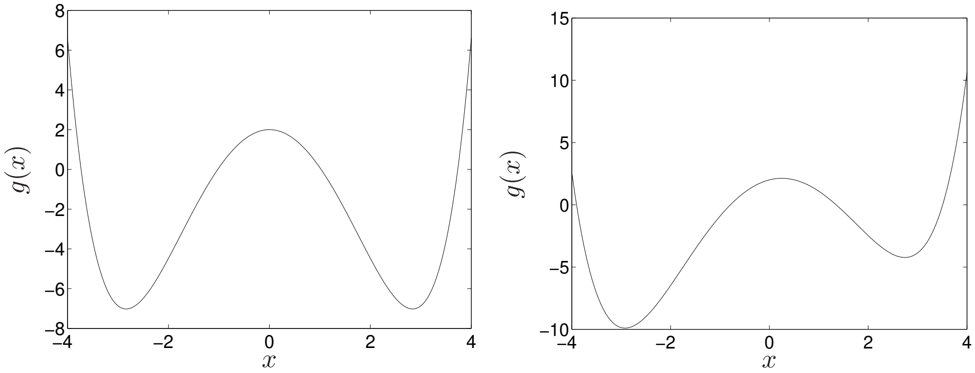

Figure 5.1 Left: the function $g(x) = e^x + e^{-x} - 3x^2$. Right: the function $g(x) = e^x + e^{-x} - 3x^2 + x$.

- $g(x) = e^x + e^{-x} - 3x^2 + x$, shown in figure 5.1 (right). The optimal value is -9.90, attained at the minimum $x^\star = -2.92$. There is another local minimum at $x = 2.74$.

- $g(x) = e^{-x}$. The optimal value is $\min g(x) = 0$, but is not attained. There are no local or global minima.

- $g(x) = -x + e^{-x}$. This function is unbounded below; the optimal value is $\min g(x) = -\infty$. There are no local or global minima.

5.2 Gradient and Hessian

Gradient The *gradient* of a function $g(x)$ of n variables, at $\hat{x}$, is the vector of first partial derivatives evaluated at $\hat{x}$, and is denoted $\nabla g(\hat{x})$:

$$\nabla g(\hat{x}) = \left(\frac{\partial g}{\partial x_1}(\hat{x}),\ \frac{\partial g}{\partial x_2}(\hat{x}),\ \ldots,\ \frac{\partial g}{\partial x_n}(\hat{x}) \right).$$

The gradient is used in the first-order (or affine) approximation of g around $\hat{x}$,

$$\begin{aligned} \hat{g}(x) &= g(\hat{x}) + \sum_{i=1}^{n} \frac{\partial g}{\partial x_i}(\hat{x})(x_i - \hat{x}_i) \\ &= g(\hat{x}) + \nabla g(\hat{x})^T (x - \hat{x}). \end{aligned}$$

If $n = 1$, the gradient is simply the first derivative $g'(\hat{x})$, and the first-order approximation reduces to

$$\hat{g}(x) = g(\hat{x}) + g'(\hat{x})(x - \hat{x}).$$

Hessian The *Hessian* of a function $g(x)$ of n variables, at $\hat{x}$, is the matrix of second partial derivatives evaluated at $\hat{x}$, and is denoted as $\nabla^2 g(\hat{x})$:

$$\nabla^2 g(\hat{x}) = \begin{bmatrix} \dfrac{\partial^2 g}{\partial x_1^2}(\hat{x}) & \dfrac{\partial^2 g}{\partial x_1 \partial x_2}(\hat{x}) & \cdots & \dfrac{\partial^2 g}{\partial x_1 \partial x_n}(\hat{x}) \\ \dfrac{\partial^2 g}{\partial x_2 \partial x_1}(\hat{x}) & \dfrac{\partial^2 g}{\partial x_2^2}(\hat{x}) & \cdots & \dfrac{\partial^2 g}{\partial x_2 \partial x_n}(\hat{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 g}{\partial x_n \partial x_1}(\hat{x}) & \dfrac{\partial^2 g}{\partial x_n \partial x_2}(\hat{x}) & \cdots & \dfrac{\partial^2 g}{\partial x_n^2}(\hat{x}) \end{bmatrix}.$$

This is a symmetric matrix, because

$$\frac{\partial^2 g}{\partial x_i \partial x_j}(\hat{x}) = \frac{\partial^2 g}{\partial x_j \partial x_i}(\hat{x}).$$

The Hessian is related to the second-order (or quadratic) approximation of g around $\hat{x}$, which is defined as

$$\begin{aligned} g_{\mathrm{q}}(x) &= g(\hat{x}) + \sum_{i=1}^{n} \frac{\partial g}{\partial x_i}(\hat{x})(x_i - \hat{x}_i) + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial^2 g}{\partial x_i \partial x_j}(\hat{x})(x_i - \hat{x}_i)(x_j - \hat{x}_j) \\ &= g(\hat{x}) + \nabla g(\hat{x})^T(x - \hat{x}) + \frac{1}{2}(x - \hat{x})^T \nabla^2 g(\hat{x})(x - \hat{x}). \end{aligned}$$

If $n = 1$, the Hessian is the second derivative $g''(\hat{x})$, and the second-order approximation reduces to

$$g_{\mathrm{q}}(x) = g(\hat{x}) + g'(\hat{x})(x - \hat{x}) + \frac{1}{2}g''(\hat{x})(x - \hat{x})^2.$$

As an example, the Hessian of the function

$$g(x_1, x_2) = e^{x_1 + x_2 - 1} + e^{x_1 - x_2 - 1} + e^{-x_1 - 1} \tag{5.1}$$

is

$$\nabla^2 g(x) = \begin{bmatrix} e^{x_1+x_2-1} + e^{x_1-x_2-1} + e^{-x_1-1} & e^{x_1+x_2-1} - e^{x_1-x_2-1} \\ e^{x_1+x_2-1} - e^{x_1-x_2-1} & e^{x_1+x_2-1} + e^{x_1-x_2-1} \end{bmatrix}, \tag{5.2}$$

so the second-order approximation around $\hat{x} = 0$ is

$$g_{\mathrm{q}}(x) = \frac{1}{e}(3 + x_1 + (3/2)x_1^2 + x_2^2).$$

Properties We list here a few properties that often simplify the task of calculating gradients and Hessians. These facts are straightforward (although sometimes tedious) to verify, directly from the definition of gradient and Hessian.

1. *Linear and affine functions.* The gradient and Hessian of $g(x) = a^T x + b$ are

$$\nabla g(x) = a, \qquad \nabla^2 g(x) = 0.$$

2. *Quadratic functions.* The gradient and Hessian of $g(x) = x^T Px + q^T x + r$, where P is a symmetric matrix, are
$$\nabla g(x) = 2Px + q, \qquad \nabla^2 g(x) = 2P.$$

3. *Sum of two functions.* If $g(x) = g_1(x) + g_2(x)$, then
$$\nabla g(x) = \nabla g_1(x) + \nabla g_2(x), \qquad \nabla^2 g(x) = \nabla^2 g_1(x) + \nabla^2 g_2(x).$$

4. *Scalar multiplication.* If $g(x) = \alpha f(x)$ where α is a scalar, then
$$\nabla g(x) = \alpha \nabla f(x), \qquad \nabla^2 g(x) = \alpha \nabla^2 f(x).$$

5. *Composition with affine function.* If $g(x) = f(Cx + d)$ where C is an $m \times n$ matrix, d is an m-vector, and $f(y)$ is a function of m variables, then
$$\nabla g(x) = C^T \nabla f(Cx + d), \qquad \nabla^2 g(x) = C^T \nabla^2 f(Cx + d) C.$$
Note that $f(Cx + d)$ denotes the function $f(y)$, evaluated at $y = Cx + d$. Similarly, $\nabla f(Cx + d)$ is the gradient $\nabla f(y)$, evaluated at $y = Cx + d$, and $\nabla^2 f(Cx + d)$ is the Hessian $\nabla^2 f(y)$, evaluated at $y = Cx + d$.

Examples As a first example, consider the least squares function
$$g(x) = \|Ax - b\|^2.$$

We can find the gradient and Hessian by expanding g in terms of its variables x_i, and then taking the partial derivatives. An easier derivation is from the properties listed above. We can express g as
$$\begin{array}{rcl} g(x) & = & (Ax - b)^T (Ax - b) \\ & = & x^T A^T Ax - b^T Ax - x^T A^T b + b^T b \\ & = & x^T A^T Ax - 2b^T Ax + b^T b. \end{array}$$

This shows that g is a quadratic function: $g(x) = x^T Px + q^T x + r$ with $P = A^T A$, $q = -2A^T b$, $r = b^T b$. From property 2,
$$\nabla g(x) = 2A^T Ax - 2A^T b, \qquad \nabla^2 g(x) = 2A^T A.$$

An alternative derivation is based on property 5. We can express g as $g(x) = f(Cx + d)$ where $C = A$, $d = -b$, and
$$f(y) = \|y\|^2 = \sum_{i=1}^m y_i^2.$$

The gradient and Hessian of f are
$$\nabla f(y) = \left[\begin{array}{c} 2y_1 \\ 2y_2 \\ \vdots \\ 2y_m \end{array}\right] = 2y, \qquad \nabla^2 f(y) = \left[\begin{array}{cccc} 2 & 0 & \cdots & 0 \\ 0 & 2 & \cdots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & 2 \end{array}\right] = 2I.$$

Applying property 5, we find that

$$\begin{aligned}\nabla g(x) &= A^T\nabla f(Ax-b)\\ &= 2A^T(Ax-b),\\ \nabla^2 g(x) &= A^T\nabla^2 f(Ax-b)A\\ &= 2A^TA,\end{aligned}$$

the same expressions as we derived before.

We can use the same method to find the gradient and Hessian of the function in (5.1),

$$g(x_1,x_2)=e^{x_1+x_2-1}+e^{x_1-x_2-1}+e^{-x_1-1}.$$

We can express g as $g(x)=f(Cx+d)$, where $f(y)=e^{y_1}+e^{y_2}+e^{y_3}$, and

$$C=\begin{bmatrix}1 & 1\\ 1 & -1\\ -1 & 0\end{bmatrix},\qquad d=\begin{bmatrix}-1\\ -1\\ -1\end{bmatrix}.$$

The gradient and Hessian of f are

$$\nabla f(y)=\begin{bmatrix}e^{y_1}\\ e^{y_2}\\ e^{y_3}\end{bmatrix},\qquad \nabla^2 f(y)=\begin{bmatrix}e^{y_1} & 0 & 0\\ 0 & e^{y_2} & 0\\ 0 & 0 & e^{y_3}\end{bmatrix},$$

so it follows from property 5 that

$$\nabla g(x)=C^T\nabla f(Cx+d)=\begin{bmatrix}1 & 1 & -1\\ 1 & -1 & 0\end{bmatrix}\begin{bmatrix}e^{x_1+x_2-1}\\ e^{x_1-x_2-1}\\ e^{-x_1-1}\end{bmatrix}$$

and

$$\begin{aligned}&\nabla^2 g(x)=C^T\nabla^2 f(Cx+d)C\\ &=\begin{bmatrix}1 & 1 & -1\\ 1 & -1 & 0\end{bmatrix}\begin{bmatrix}e^{x_1+x_2-1} & 0 & \\ 0 & e^{x_1-x_2-1} & 0\\ 0 & 0 & e^{-x_1-1}\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & -1\\ -1 & 0\end{bmatrix}.\end{aligned}\tag{5.3}$$

5.3 Optimality conditions

Local or global minima of a function g can be characterized in terms of the gradient and Hessian. In this section we state the optimality conditions (without proofs).

Global optimality A function g is *convex* if $\nabla^2 g(x)$ is positive semidefinite everywhere (for all x). If g is convex, then $x^\star$ is a minimum if and only if

$$\nabla g(x^\star)=0.$$

There are no other local minima, *i.e.*, every local minimum is global.

Examples with $n = 1$ A function of one variable is convex if $g''(x) \geq 0$ everywhere. Therefore $x^\star$ is a minimum of a convex function if and only if

$$g'(x^\star) = 0.$$

The functions $g(x) = x^2$ and $g(x) = x^4$ are convex, with second derivatives $g''(x) = 2$ and $g''(x) = 12x^2$, respectively. Therefore we can find the minimum by setting the first derivative equal to zero, which in both cases yields $x^\star = 0$.

The first and second derivatives of the function

$$g(x) = \log(e^x + e^{-x})$$

are

$$g'(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \qquad g''(x) = \frac{4}{(e^x + e^{-x})^2}.$$

The second derivative is nonnegative everywhere, so g is convex, and we can find its minimum by solving $g'(x^\star) = 0$, which gives $x^\star = 0$.

Examples with $n > 1$ A quadratic function $g(x) = x^T P x + q^T x + r$ (with P symmetric) is convex if P is positive semidefinite. (Recall that $\nabla^2 g(x) = 2P$.) Therefore $x^\star$ is a minimum if and only if

$$\nabla g(x^\star) = 2Px^\star + q = 0,$$

which is a set of n linear equations in n variables. If P is positive definite, the equations have a unique solution $x^\star = -(1/2)P^{-1}q$, and can be efficiently solved using the Cholesky factorization.

The least squares function $g(x) = \|Ax - b\|^2$ is convex, because $\nabla^2 g(x) = 2A^T A$, and the matrix $A^T A$ is positive semidefinite. Therefore $x^\star$ is a minimum if and only if

$$\nabla g(x^\star) = 2A^T A x^\star - 2A^T b = 0.$$

We can find $x^\star$ by solving the set of linear equations

$$A^T A x^\star = A^T b,$$

in which we recognize the normal equations associated with the least squares problem.

In these first two examples, we can solve $\nabla g(x^\star) = 0$ by solving a set of linear equations, so we do not need an iterative method to minimize g. In general, however, the optimality condition $\nabla g(x^\star) = 0$ is a set of nonlinear equations in $x^\star$ that has to be solved by an iterative algorithm.

The function g defined in (5.1), for example, is convex, because its Hessian is positive definite everywhere. Although that is not obvious from the expression in (5.2), it follows from (5.3) which shows that

$$\nabla^2 g(x) = C^T D C$$

where D is a diagonal matrix with diagonal elements

$$d_{11} = e^{x_1 + x_1 - 1}, \qquad d_{22} = e^{x_1 + x_1 - 1}, \qquad d_{33} = e^{-x_1 - 1},$$

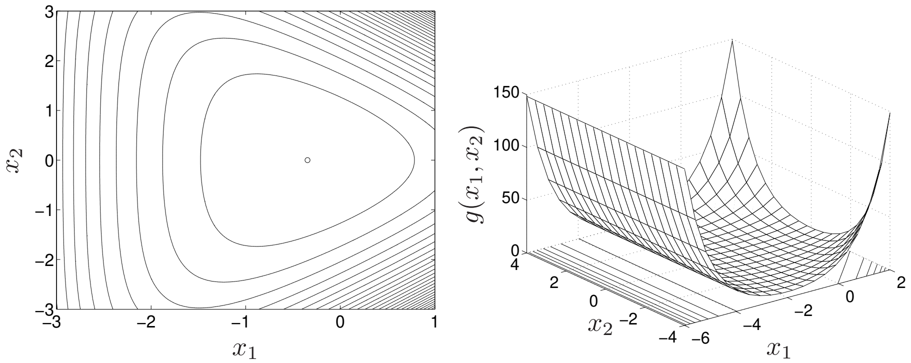

Figure 5.2 Contour lines and graph of the function $g(x) = \exp(x_1 + x_2 - 1) + \exp(x_1 - x_2 - 1) + \exp(-x_1 - 1)$.

and C is a left-invertible 3×2 matrix. It can be shown that $v^T C^T DCv > 0$ for all nonzero v, hence $C^T DC$ is positive definite. It follows that x is a minimum of g if and only if

$$\nabla g(x) = \left[\begin{array}{c} e^{x_1+x_2-1} + e^{x_1-x_2-1} - e^{-x_1-1} \\ e^{x_1+x_2-1} - e^{x_1-x_2-1} \end{array} \right] = 0.$$

This is a set of two nonlinear equations in two variables.

The contour lines and graph of the function g are shown in figure 5.2.

Local optimality It is much harder to characterize optimality if g is not convex (*i.e.*, if there are points where the Hessian is not positive semidefinite). It is not sufficient to set the gradient equal to zero, because such a point might correspond to a local minimum, a local maximum, or a saddle point (see for example, the second function in figure 5.1). However, we can state some simple conditions for *local* optimality.

- *Necessary condition.* If $x^\star$ is locally optimal, then $\nabla g(x^\star) = 0$ and $\nabla^2 g(x^\star)$ is positive semidefinite.

- *Sufficient condition.* If $\nabla g(x^\star) = 0$ and $\nabla^2 g(x^\star)$ is positive definite, then $x^\star$ is locally optimal.

The function $g(x) = x^3$ provides an example that shows that "positive definite" cannot be replaced by "positive semidefinite" in the sufficient condition. At $x = 0$ it satisfies $g'(x) = 3x^2 = 0$ and $g''(x) = 6x = 0$, although $x = 0$ is not a local minimum.

5.4 Newton's method for minimizing a convex function

We first consider the important case when the objective function g is convex. As we have seen in section 5.3, we can find the minimum by solving $\nabla g(x) = 0$. This is a set of n nonlinear equations in n variables that we can solve using any method for nonlinear equations, for example, Newton's method.

For simplicity we assume that $\nabla^2 g(x)$ is positive definite everywhere, which is a little stronger than requiring $\nabla^2 g(x)$ to be positive semidefinite.

Newton's method for solving nonlinear equations, applied to $\nabla g(x) = 0$, is based on the iteration

$$x^{(k+1)} = x^{(k)} - \nabla^2 g(x^{(k)})^{-1} \nabla g(x^{(k)}), \quad k = 0, 1, 2, \ldots.$$

A more detailed description is as follows.

Algorithm 5.1. NEWTON'S METHOD FOR UNCONSTRAINED MINIMIZATION.

```
given initial $x$, tolerance $\epsilon > 0$
repeat
  1. Evaluate $\nabla g(x)$ and $\nabla^2 g(x)$.
  2. if $\|\nabla g(x)\| \leq \epsilon$, return $x$.
  3. Solve $\nabla^2 g(x) v = -\nabla g(x)$.
  4. $x := x + v$.
until a limit on the number of iterations is exceeded
```

Since $\nabla^2 g(x)$ is positive definite, we can use the Cholesky factorization in step 3.

The vector v computed in the kth iteration is called the *Newton step* at $x^{(k)}$:

$$v^{(k)} = -\nabla^2 g(x^{(k)})^{-1} \nabla g(x^{(k)}).$$

The Newton step can be interpreted in several ways.

Interpretation as solution of linearized optimality condition In chapter 4 we have seen that the iterates in Newton's method for solving nonlinear equations can be interpreted as solutions of linearized problems.

If we linearize the optimality condition $\nabla g(x) = 0$ near $\hat{x} = x^{(k)}$ we obtain

$$\nabla g(x) \approx \nabla g(\hat{x}) + \nabla^2 g(\hat{x})(x - \hat{x}) = 0.$$

This is a linear equation in x, with solution

$$x = \hat{x} - \nabla^2 g(\hat{x})^{-1} \nabla g(\hat{x}) = x^{(k)} + v^{(k)}.$$

So the Newton step $v^{(k)}$ is what must be added to $x^{(k)}$ so that the linearized optimality condition holds.

When $n = 1$ this interpretation is particularly simple. The solution of the linearized optimality condition is the zero-crossing of the derivative $g'(x)$, which is monotonically increasing since $g''(x) > 0$. Given our current approximation $x^{(k)}$ of the solution, we form a first-order Taylor approximation of $g'(x)$ at $x^{(k)}$. The zero-crossing of this approximation is then $x^{(k)} + v^{(k)}$. This interpretation is illustrated in figure 5.3.

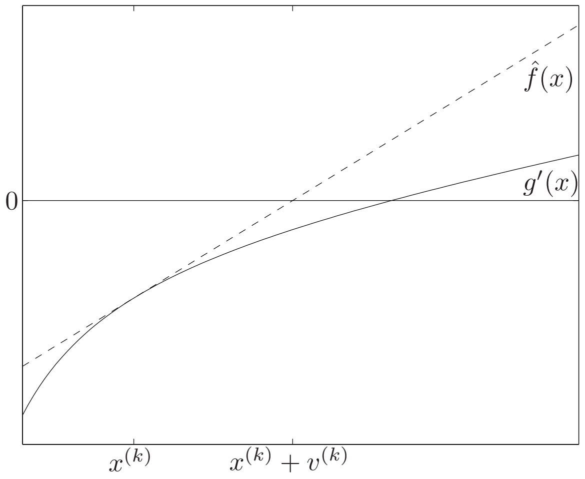

Figure 5.3 The solid curve is the derivative $g'(x)$ of the function g. $\hat{f}(x) = g'(x^{(k)}) + g''(x^{(k)})(x - x^{(k)})$ is the affine approximation of $g'(x)$ at $x^{(k)}$. The Newton step $v^{(k)}$ is the difference between the root of $\hat{f}$ and the point $x^{(k)}$.

Interpretation as minimum of second-order approximation The second-order approximation $g_{\rm q}$ of g near $\hat{x} = x^{(k)}$ is

$$g_{\rm q}(x) = g(\hat{x}) + \nabla g(\hat{x})^T(x - \hat{x}) + \frac{1}{2}(x - \hat{x})^T \nabla^2 g(\hat{x})(x - \hat{x}) \qquad (5.4)$$

which is a convex quadratic function of y. To find the gradient and Hessian of $g_{\rm q}$, we express the function as

$$g_{\rm q}(x) = x^T P x + q^T x + r$$

where

$$P = \frac{1}{2}\nabla^2 g(\hat{x}), \qquad q = \nabla g(\hat{x}) - \nabla^2 g(\hat{x})\hat{x}, \qquad r = g(\hat{x}) - \nabla g(\hat{x})^T \hat{x} + \frac{1}{2}\hat{x}^T \nabla^2 g(\hat{x})\hat{x},$$

and then apply the second property in section 5.2:

$$\nabla^2 g_{\rm q}(x) = 2P = \nabla^2 g(\hat{x})$$

and

$$\nabla g_{\rm q}(x) = 2Px + q = \nabla g(\hat{x}) + \nabla^2 g(\hat{x})(x - \hat{x}).$$

The minimum of $g_{\rm q}$ is

$$x = -\frac{1}{2}P^{-1}q = \hat{x} - \nabla^2 g(\hat{x})^{-1}\nabla g(\hat{x}).$$

Thus, the Newton step $v^{(k)}$ is what should be added to $x^{(k)}$ to minimize the second-order approximation of g at $x^{(k)}$. This is illustrated in figure 5.4.

This interpretation gives us some insight into the Newton step. If the function g is quadratic, then $x^{(k)} + v^{(k)}$ is the exact minimum of g. If the function g is nearly quadratic, $x^{(k)} + v^{(k)}$ should be a very good estimate of the minimum of g.

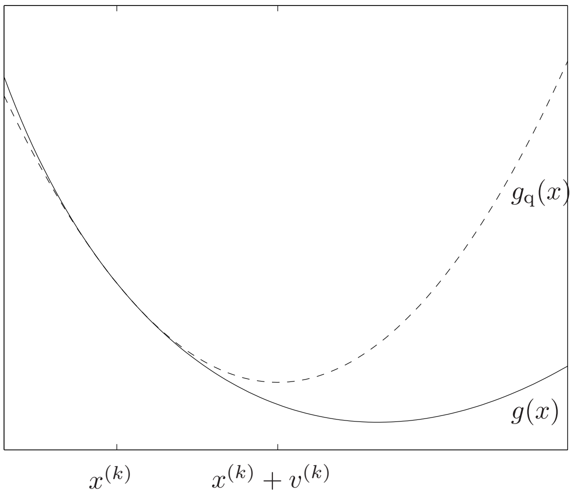

Figure 5.4 The function g (solid line) and its second-order approximation $g_{\rm q}$ at $x^{(k)}$ (dashed). The Newton step $v^{(k)}$ is the difference between the minimum of $g_{\rm q}$ and the point $x^{(k)}$.

Example We apply Newton's method to the convex function

$$g(x) = \log(e^x + e^{-x}).$$

Figure 5.5 shows the iteration when we start at $x^{(0)} = 0.85$. Figure 5.6 shows the iteration when we start at $x^{(0)} = 1.15$.

We notice that Newton's method only converges when started near the solution, as expected based on the general properties of the method. In the next paragraph we describe a simple and easily implemented modification that makes the method globally convergent (*i.e.*, convergent from any starting point).

5.5 Newton's method with line search

The purpose of the line search is to avoid the behavior of figure 5.6, in which the function values *increase* from iteration to iteration. A look at the example shows that there is actually nothing wrong with the direction of the Newton step, since it always points in the direction of decreasing g. The problem is that we step too far in that direction. So the remedy is quite obvious. At each iteration, we first attempt the full Newton step $x^{(k)} + v^{(k)}$, and evaluate g at that point. If the function value $g(x^{(k)} + v^{(k)})$ is higher than $g(x^{(k)})$, we reject the update, and try $x^{(k)} + (1/2)v^{(k)}$, instead. If the function value is still higher than $g(x^{(k)})$, we try $x^{(k)} + (1/4)v^{(k)}$, and so on, until a value of t is found with $g(x^{(k)} + tv^{(k)}) < g(x^{(k)})$. We then take $x^{(k+1)} = x^{(k)} + tv^{(k)}$. In practice, this backtracking idea is often implemented as shown in the following outline.

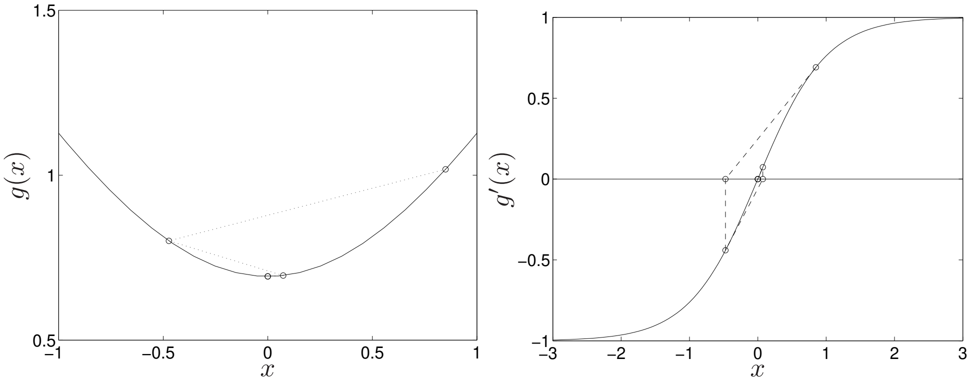

Figure 5.5 The solid line in the left figure is $g(x) = \log(\exp(x) + \exp(-x))$. The circles indicate the function values at the successive iterates in Newton's method, starting at $x^{(0)} = 0.85$. The solid line in the right figure is the derivative $g'(x)$. The dashed lines in the right-hand figure illustrate the first interpretation of Newton's method (linearization of the optimality condition $g'(x) = 0$).

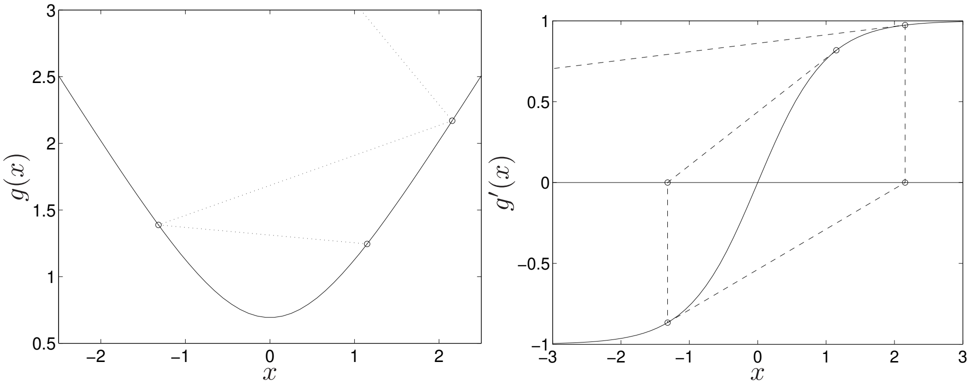

Figure 5.6 The solid line in the left figure is $g(x) = \log(\exp(x) + \exp(-x))$. The circles indicate the function values at the successive iterates in Newton's method, starting at $x^{(0)} = 1.15$. The solid line in the right figure is the derivative $g'(x)$. The dashed lines in the right-hand figure illustrate the first interpretation of Newton's method.

Algorithm 5.2. Newton's method with line search.

```
given initial $x$, tolerance $\epsilon > 0$, parameter $\alpha \in (0, 1/2)$.
repeat
    1. Evaluate $\nabla g(x)$ and $\nabla^2 g(x)$.
    2. if $\|\nabla g(x)\| \leq \epsilon$, return $x$.
    3. Solve $\nabla^2 g(x) v = -\nabla g(x)$.
    4. $t := 1$.
       while $g(x+tv) > g(x) + \alpha t \nabla g(x)^T v$, $t := t/2$.
    5. $x := x + tv$.
until a limit on the number of iterations is exceeded
```

The parameter α is usually chosen quite small (*e.g.*, $\alpha = 0.01$).

The scalar t computed in step 4 is called the *step size*, and the algorithm used to calculate the step size (*i.e.*, step 4) is called the *line search*. During the line search, we examine candidate updates $x + tv$ (for $t = 1, 1/2, 1/4, \ldots$) on the line that passes through x and $x + v$, hence the name line search.

The purpose of the line search is to find a step size t such that $g(x + tv)$ is sufficiently less than $g(x)$. More specifically, the condition of sufficient decrease (used in step 4) is that the step size t is accepted if

$$g(x + tv) \leq g(x) + \alpha t \nabla g(x)^T v. \tag{5.5}$$

To clarify this condition, we consider $g(x + tv)$, where $x = x^{(k)}$, $v = v^{(k)}$, as a function of t. The function $h(t) = g(x + tv)$ is a function of one variable t, and gives the values of g on the line $x+tv$, as a function of the step size t (see figure 5.7). At $t = 0$, we have $h(0) = g(x)$.

A first important observation is that the quantity $\nabla g(x)^T v$ is the derivative of h at $t = 0$. More generally,

$$\begin{aligned} h'(t) &= \frac{\partial g}{\partial x_1}(x+tv)v_1 + \frac{\partial g}{\partial x_2}(x+tv)v_2 + \cdots + \frac{\partial g}{\partial x_n}(x+tv)v_n \\ &= \nabla g(x+tv)^T v, \end{aligned}$$

so at $t = 0$ we have $h'(0) = \nabla g(x)^T v$. (The inner product $\nabla g(x)^T v$ is known as the *directional derivative* of g at x in the direction v.) It immediately follows that $h'(0) < 0$:

$$h'(0) = \nabla g(x)^T v = -v^T \nabla^2 g(x) v < 0,$$

because v is defined as $v = -\nabla^2 g(x)^{-1} \nabla g(x)$, and $\nabla^2 g(x)$ is positive definite.

The linear approximation of $h(t) = g(x + tv)$ at $t = 0$ is

$$h(0) + h'(0)t = g(x) + t\nabla g(x)^T v,$$

so for small enough t we have

$$g(x + tv) \approx g(x) + t\nabla g(x)^T v < g(x) + \alpha t \nabla g(x)^T v.$$

This shows that the backtracking line search eventually terminates. The constant α can be interpreted as the fraction of the decrease in g predicted by linear extrapolation that we will accept.

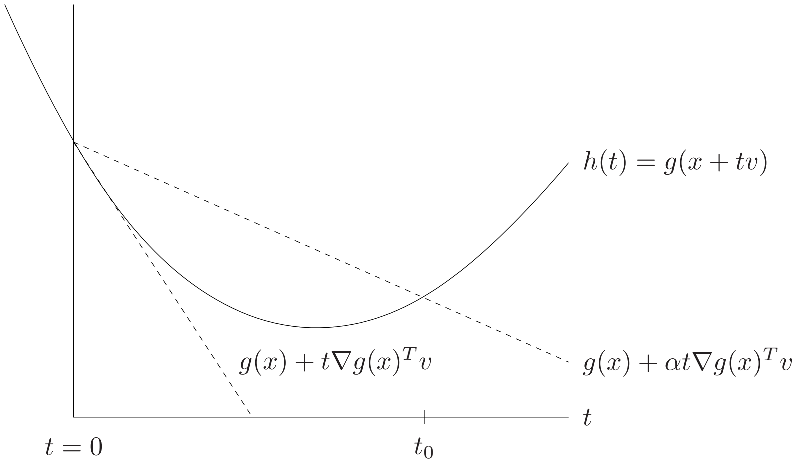

Figure 5.7 Backtracking line search. The curve shows g, restricted to the line over which we search. The lower dashed line shows the linear extrapolation of g, and the upper dashed line has a slope a factor of α smaller. The backtracking condition is that $g(x + tv)$ lies below the upper dashed line, *i.e.*, $0 \le t \le t_0$. The line search starts with $t = 1$, and divides t by 2 until $t \le t_0$.

Examples We start with two small examples. Figure 5.8 shows the iterations in Newton's method with backtracking, applied to the example of figure 5.6, starting from $x^{(0)} = 4$. As expected the convergence problem has been resolved. From the plot of the step sizes we note that the method accepts the full Newton step ($t = 1$) after a few iterations. This means that near the solution the algorithm works like the pure Newton method, which ensures fast (quadratic) convergence.

Figure 5.9 shows the results of Newton's method applied to

$$g(x_1, x_2) = \exp(x_1 + x_2 - 1) + \exp(x_1 - x_2 - 1) + \exp(-x_1 - 1),$$

starting at $x^{(0)} = (-2, 2)$.

To give a larger example, suppose we are given an $m \times n$-matrix A and an m-vector b, and we are interested in minimizing

$$g(x) = \sum_{i=1}^{m} \log(e^{a_i^T x - b_i} + e^{-a_i^T x + b_i}). \tag{5.6}$$

where a_i^T denotes the ith row of A. To implement Newton's method, we need the gradient and Hessian of g. The easiest method is to use the composition property in section 5.2. We express g as $g(x) = f(Ax - b)$, where f is the following function of m variables:

$$f(y) = \sum_{i=1}^{m} \log(e^{y_i} + e^{-y_i}).$$

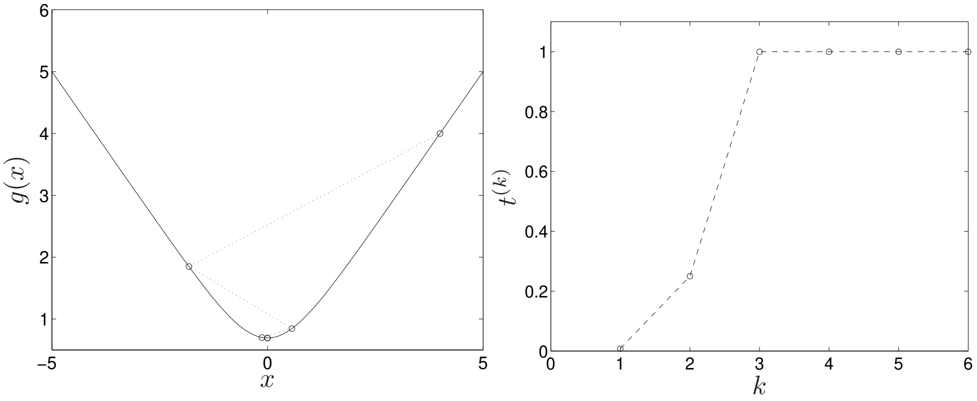

Figure 5.8 The solid line in the figure on the left is $g(x) = \log(\exp(x) + \exp(-x))$. The circles indicate the function values at the successive iterates in Newton's method with backtracking, starting at $x^{(0)} = 4$. The right-hand figure shows the step size $t^{(k)}$ at each iteration. In the first iteration ($k = 0$) the step size is $1/2^7$, *i.e.*, 7 backtracking steps were made. In the second iteration the step size is 0.25 (2 backtracking steps). The other iterations take a full Newton step ($t = 1$).

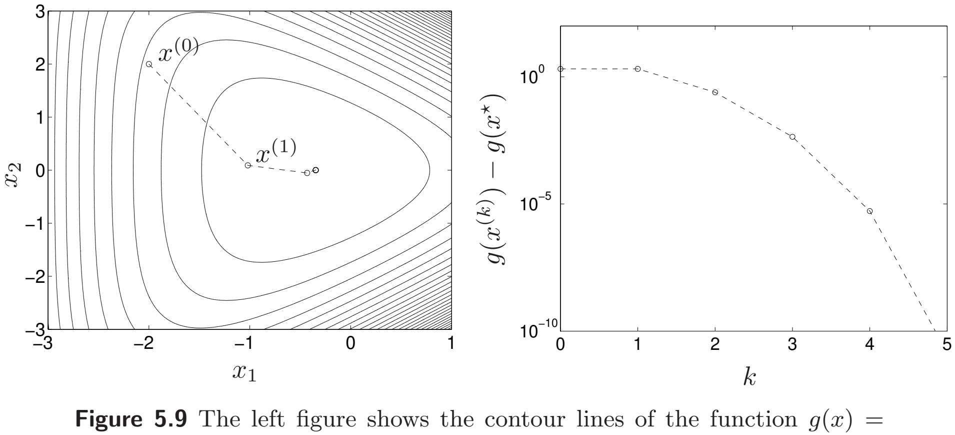

Figure 5.9 The left figure shows the contour lines of the function $g(x) = \exp(x_1 + x_2 - 1) + \exp(x_1 - x_2 - 1) + \exp(-x_1 - 1)$. The circles indicate the iterates of Newton's method, started at $(-2, 2)$. The figure on the right shows $g(x^{(k)}) - g(x^\star)$ versus k.

The partial derivatives of f are given by

$$\frac{\partial f}{\partial y_i}(y) = \frac{e^{y_i} - e^{-y_i}}{e^{y_i} + e^{-y_i}}, \qquad \frac{\partial^2 f}{\partial y_i \partial y_j}(y) = \left\{ \begin{array}{ll} 4/(e^{y_i} + e^{-y_i})^2 & i = j \\ 0 & i \neq j. \end{array} \right.$$

(In other words, $\nabla^2 f(y)$ is diagonal with diagonal elements $4/(\exp(y_i)+\exp(-y_i))^2$.) Given the gradient and Hessian of f, we can find the gradient and Hessian of g:

$$\nabla g(x) = A^T \nabla f(Ax - b), \qquad \nabla^2 g(x) = A^T \nabla^2 f(Ax - b) A.$$

Once we have the correct expressions for the gradient and Hessian, the implementation of Newton's method is straightforward. The MATLAB code is given below.

```
x = ones(n,1);
for k = 1:50
    y = A*x-b;
    val = sum(log(exp(y)+exp(-y)));
    grad = A'*((exp(y)-exp(-y))./(exp(y)+exp(-y)));
    if (norm(grad) < 1e-5), break;  end;
    hess = 4*A'*diag(1./(exp(y)+exp(-y)).^2)*A;
    v = -hess\grad;
    t = 1;
    while ( sum(log(exp(A*(x+t*v)-b)+exp(-A*(x+t*v)+b))) ...
            > val + 0.01*t*grad'*v),  t = 0.5*t;  end;
    x = x+t*v;
end;
```

We start with $x^{(0)} = (1, 1, \ldots, 1)$, set the line search parameter α to $\alpha = 0.01$, and terminate if $\|\nabla f(x)\| \leq 10^{-5}$.

The results, for an example with $m = 500$ and $n = 100$ are shown in figure 5.10. We note that many backtracking steps are needed in the first few iterations, until we reach a neighborhood of the solution in which the pure Newton method converges.

5.6 Newton's method for nonconvex functions

Minimization problems with a nonconvex cost function g presents additional difficulties to Newton's method, even if the only goal is to find a local minimum. If $\nabla^2 g(x^{(k)})$ is not positive definite, then the Newton step

$$v^{(k)} = -\nabla^2 g(x^{(k)})^{-1} \nabla g(x^{(k)}),$$

might not be a descent direction. In other words, it is possible that the function $h(t) = g(x^{(k)} + tv^{(k)})$ that we considered in figure 5.7, has a *positive* slope at $t = 0$. To see this, recall that the slope of h at $t = 0$ is given by

$$h'(0) = \nabla g(x^{(k)})^T v^{(k)},$$

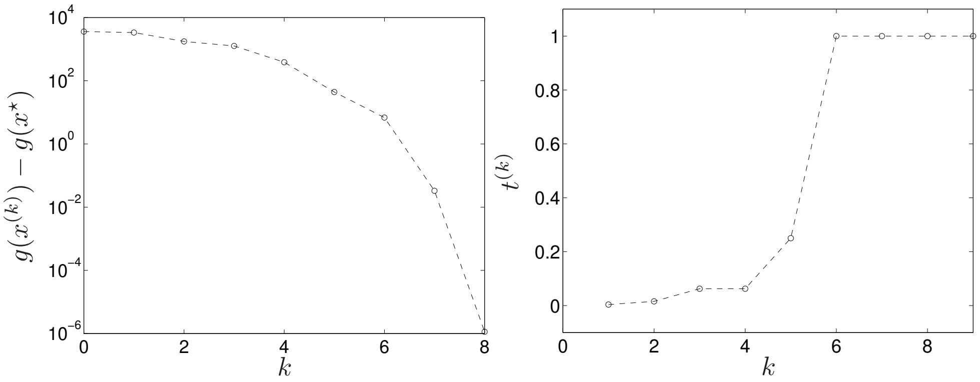

Figure 5.10 Results of Newton's method with backtracking applied to the function (5.6) for an example with $m = 500$, and $n = 100$. The figure on the left is $g(x^{(k)}) - g(x^\star)$ versus k. The convergence accelerates as k increases, and is very fast in the last few iterations. The right-hand figure shows the step size $t^{(k)}$ versus k. The last four iterations use a full Newton step ($t = 1$).

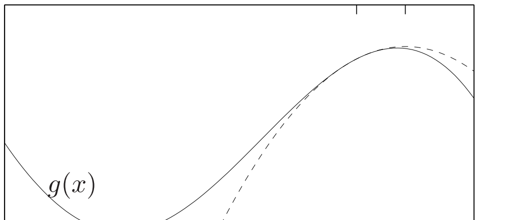

Figure 5.11 A nonconvex function $g(x)$ and the second-order approximation $g_{\mathrm{q}}(x)$ of g around $x^{(k)}$. The Newton step $v^{(k)} = -g'(x^{(k)})/g''(x^{(k)})$ is the difference between the maximum of g_{q} and $x^{(k)}$.

and that the proof that $h'(0) < 0$ in section 5.5 was based on the assumption that $\nabla^2 g(x^{(k)})$ was positive definite. Figure 5.11 shows an example in one dimension.

Practical implementations of Newton's method for nonconvex functions make sure that the direction $v^{(k)}$ used at each iteration is a descent direction, *i.e.*, a direction that satisfies

$$h'(0) = \nabla g(x^{(k)})^T v^{(k)} < 0.$$

Many techniques have been proposed to find such a $v^{(k)}$. The simplest choice is to take $v^{(k)} = -\nabla g(x^{(k)})$ at points where the Hessian is not positive definite. This is a descent direction, because

$$h'(0) = \nabla g(x^{(k)})^T v^{(k)} = -\nabla g(x^{(k)})^T \nabla g(x^{(k)}) = -\|\nabla g(x^{(k)})\|^2.$$

With this addition, Newton's method can be summarized as follows.

Algorithm 5.3. NEWTON'S METHOD FOR NONCONVEX LOCAL MINIMIZATION.

```
given initial $x$, tolerance $\epsilon > 0$, parameter $\alpha \in (0, 1/2)$.
repeat
    1. Evaluate $\nabla g(x)$ and $\nabla^2 g(x)$.
    2. if $\|\nabla g(x)\| \le \epsilon$, return $x$.
    3. if $\nabla^2 g(x)$ is positive definite, solve $\nabla^2 g(x) v = -\nabla g(x)$ for $v$
       else, $v := -\nabla g(x)$.
    4. $t := 1$.
       while $g(x + tv) > g(x) + \alpha t \nabla g(x)^T v$, $t := t/2$.
    5. $x := x + tv$.
until a limit on the number of iterations is exceeded
```

Although the algorithm guarantees that the function values decrease at each iteration, it is still far from perfect. Note for example, that if we start at a point where $\nabla g(x)$ is zero (or very small), the algorithm terminates immediately in step 2, although we might be at a local maximum. Experience also shows that the convergence can be very slow, until we get close to a local minimum where the Hessian is positive definite. Practical implementations of Newton's method for nonconvex minimization use more complicated methods for finding good descent directions when the Hessian is not positive definite, and more sophisticated line searches.

Chapter 6

Condition and stability

The final two chapters are a short introduction to topics related to the accuracy of numerical algorithms. In this chapter we discuss the concepts of problem condition and algorithm stability.

6.1 Problem condition

A mathematical problem is *well conditioned* if small changes in the problem parameters (the problem data) lead to small changes in the solution. A problem is *badly conditioned* or *ill-conditioned* if small changes in the parameters can cause large changes in the solution. In other words, the solution of a badly conditioned problem is very sensitive to changes in the parameters. Note that this is an informal definition. A precise definition requires defining what we mean by large or small errors (*e.g.*, relative or absolute error, choice of norm), and a statement of which of the parameters are subject to error.

In engineering problems, the data are almost always subject to uncertainty, due to measurement errors, imperfect knowledge of the system, modeling approximations, rounding errors in previous calculations, etc., so it is important in practice to have an idea of the condition of a problem.

Example Consider the two equations in two variables

$$\begin{array}{rcl} x_1 + x_2 & = & b_1 \\ (1+10^{-5})x_1 + (1-10^{-5})x_2 & = & b_2. \end{array}$$

It is easily verified that the coefficient matrix

$$A = \left[\begin{array}{cc} 1 & 1 \\ 1+10^{-5} & 1-10^{-5} \end{array} \right]$$

is nonsingular, with inverse

$$A^{-1} = \frac{1}{2} \left[\begin{array}{cc} 1-10^{5} & 10^{5} \\ 1+10^{5} & -10^{5} \end{array} \right],$$

so the solution of $Ax = b$ is

$$x = A^{-1}b = \frac{1}{2}\left[\begin{array}{cc} 1-10^5 & 10^5 \\ 1+10^5 & -10^5 \end{array}\right]\left[\begin{array}{c} b_1 \\ b_2 \end{array}\right] = \frac{1}{2}\left[\begin{array}{c} b_1 - 10^5(b_1 - b_2) \\ b_1 + 10^5(b_1 - b_2) \end{array}\right].$$

Plugging in a few values for b_1 and b_2 around $(1, 1)$ gives the following results.

b_1	b_2	x_1	x_2
1.00	1.00	0.500	0.500
1.01	1.01	0.505	0.505
0.99	1.01	1000.495	−999.505
1.01	0.99	−999.505	1000.495
0.99	0.99	0.495	0.495

We immediately notice that small changes in b sometimes lead to very large changes in the solution x. In most applications that would pose a very serious problem. Suppose for example that the values $b_1 = 1$ and $b_2 = 1$ are obtained from measurements with a precision of 1%, so the actual values of b_1 and b_2 can be anywhere in the interval $[0.99, 1.01]$. The five values of x_1 and x_2 in the table are all in agreement with the measurements, so it would be foolish to accept the value $x_1 = x_2 = 0.5$, obtained from the measured values $b_1 = b_2 = 1$, as the correct solution.

A second observation is that the error in x is not always large. In the second and fifth rows of the table, the error in x is 1%, of the same order as the error in b.

In this example we only changed the values of the right-hand side b, and assumed that the matrix A is exactly known. We can also consider the effect of small changes in the coefficients of A, with similar conclusions. In fact, A is very close to the singular matrix

$$\left[\begin{array}{cc} 1 & 1 \\ 1 & 1 \end{array}\right],$$

so small changes in A might result in an unsolvable set of equations.

The most common method for examining the condition of a problem is precisely what we did in the small example above. We generate a number of possible values of the problem data, solve the problem for each of those values, and compare the results. If the results vary widely, we conclude the problem is ill-conditioned. If the results are all close, we can say with some confidence that the problem is probably well-conditioned. This method is simple and applies to any type of problem, not just linear equations. It can also be misleading. If in the small example above, we had solved the equations for three right-hand sides, $(b_1, b_2) = (1, 1)$, $(b_1, b_2) = (1.01, 1.01)$, $(b_1, b_2) = (0.99, 0.99)$, then we would have incorrectly concluded that the problem is well-conditioned.

A more rigorous method is to mathematically derive bounds on the error in the solution, given a certain error in the problem data. For most numerical problems this is quite difficult, and it is the subject of a field of mathematics called Numerical Analysis. In this course we will give one example of such an error analysis: in section 6.3 we will derive simple and easily computed bounds on the error of the solution of a set of linear equations, given an error in the right-hand side. The analysis will require the notion of matrix norm, which we discuss first.

6.2 Matrix norm

The norm of a matrix serves the same purpose as the norm of a vector. It is a measure of the size or magnitude of the matrix. As for vectors, many possible definitions exist. For example, in analogy with the Euclidean norm of a vector x,

$$\|x\| = (x_1^2 + x_2^2 + \cdots + x_n^2)^{1/2},$$

we can define the norm of an $m \times n$ matrix A as

$$\|A\|_F = (\sum_{i=1}^{m}\sum_{j=1}^{n} A_{ij}^2)^{1/2},$$

i.e., the square root of the sum of the squares of the elements of A. This is called the *Frobenius norm* of A.

In this chapter, we use a different definition of matrix norm (known as *spectral norm* or *2-norm*). The 2-norm of an $m \times n$ matrix A, denoted $\|A\|_2$, is defined as

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}. \tag{6.1}$$

We will include the subscripts in the norm of a matrix to distinguish between the Frobenius norm $\|A\|_F$ and the 2-norm $\|A\|_2$.

Since the ratio $\|Ax\|/\|x\|$ remains the same if we scale x with a nonzero scalar, an equivalent definition is

$$\|A\|_2 = \max_{\|x\|=1} \|Ax\|.$$

To better understand the definition of $\|A\|_2$, it is useful to recall the "operator" or "function" interpretation of a matrix A: we can associate with an $m \times n$ matrix A a linear function

$$f(x) = Ax$$

that maps n-vectors x to m-vectors $y = Ax$. For each nonzero x, we can calculate the norm of x and Ax, and refer to the ratio $\|Ax\|/\|x\|$ as the *gain* or *amplification factor* of the operator f in the direction of x. Of course, the gain generally depends on x, and can be large for some vectors x and small (or zero) for others. The matrix norm, as defined in (6.1), is the maximum achievable gain, over all possible choices of x. Although it is not yet clear how we can actually compute $\|A\|_2$ using this definition, it certainly makes sense as a measure for the magnitude of A. If $\|A\|_2$ is small, say, $\|A\|_2 \ll 1$, then $\|Ax\| \ll \|x\|$ for all $x \neq 0$, which means that the function f strongly attenuates the input vectors x. If $\|A\|_2$ is large, then there exist input vectors x for which the gain $\|Ax\|/\|x\|$ is large.

Simple examples The norm of simple matrices can be calculated directly by applying the definition. For example, if $A = 0$, then $Ax = 0$ for all x, so $\|Ax\|/\|x\| = 0$ for all x, and hence $\|A\| = 0$. If $A = I$, we have $Ax = x$, and hence

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{x \neq 0} \frac{\|x\|}{\|x\|} = 1.$$

As another example, suppose

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

We have $Ax = (x_2, -x_3, x_1)$, hence

$$\|Ax\|_2 = \sqrt{x_2^2 + x_3^2 + x_1^2} = \|x\|,$$

so this matrix also has norm one:

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|} = \max_{x \neq 0} \frac{\|x\|}{\|x\|} = 1.$$

Next, assume that A is an $m \times 1$ matrix,

$$A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}.$$

In this case x is a scalar, so $\|x\| = |x|$ and

$$Ax = \begin{bmatrix} a_1 x \\ a_2 x \\ \vdots \\ a_m x \end{bmatrix}, \qquad \|Ax\| = |x|\sqrt{a_1^2 + a_2^2 + \cdots + a_m^2}.$$

Therefore $\|Ax\|/\|x\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_m^2}$ for all nonzero x, and

$$\|A\|_2 = \sqrt{a_1^2 + a_2^2 + \cdots + a_m^2}.$$

The matrix norm of a matrix with one column is equal to the Euclidean norm of the column vector.

In these four examples, the ratio $\|Ax\|/\|x\|$ is the same for all nonzero x, so maximizing over x is trivial. As an example where the gain varies with x, we consider a diagonal matrix

$$A = \begin{bmatrix} A_{11} & 0 & \cdots & 0 \\ 0 & A_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{nn} \end{bmatrix}.$$

Here the gain is not independent of x. For example, for $x = (1, 0, \ldots, 0)$, we have $Ax = (A_{11}, 0, \ldots, 0)$, so

$$\frac{\|Ax\|}{\|x\|} = \frac{|A_{11}|}{1} = |A_{11}|.$$

If $x = (0, 1, 0, \ldots, 0)$, the gain is $|A_{22}|$, etc. To find the matrix norm, we have to find the value of x that maximizes the gain. For general x, we have $Ax = (A_{11}x_1, A_{22}x_2, \ldots, A_{nn}x_n)$, and therefore

$$\|A\|_2 = \max_{x\neq 0} \frac{\sqrt{A_{11}^2x_1^2 + A_{22}^2x_2^2 + \cdots + A_{nn}^2x_n^2}}{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}}.$$

We will show that this is equal to the maximum of the absolute values of the diagonal elements A_{ii}:

$$\|A\|_2 = \max\{|A_{11}|, |A_{22}|, \ldots, |A_{nn}|\}.$$

Suppose for simplicity that

$$A_{11}^2 \geq A_{22}^2 \geq \cdots \geq A_{nn}^2,$$

so $|A_{11}| = \max_i |A_{ii}|$. Then

$$A_{11}^2x_1^2 + A_{22}^2x_2^2 + \cdots + A_{nn}^2x_n^2 \leq A_{11}^2(x_1^2 + x_2^2 + \cdots + x_n^2)$$

for all $x \neq 0$, and therefore

$$\frac{\sqrt{A_{11}^2x_1^2 + A_{22}^2x_2^2 + \cdots + A_{nn}^2x_n^2}}{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}} \leq |A_{11}|.$$

Moreover for $x = (1, 0, \ldots, 0)$, we have

$$\frac{\sqrt{A_{11}^2x_1^2 + A_{22}^2x_2^2 + \cdots + A_{nn}^2x_n^2}}{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}} = |A_{11}|$$

so it follows that

$$\max_{x\neq 0} \frac{\sqrt{A_{11}^2x_1^2 + A_{22}^2x_2^2 + \cdots + A_{nn}^2x_n^2}}{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}} = |A_{11}| = \max_{i=1,\ldots,n} |A_{ii}|.$$

Properties of the matrix norm The following properties of the matrix norm follow from the definition. We leave the proofs as an exercise.

- *Homogeneity.* $\|\beta A\|_2 = |\beta|\,\|A\|_2$.
- *Triangle inequality.* $\|A + B\|_2 \leq \|A\|_2 + \|B\|_2$.
- *Definiteness.* $\|A\|_2 \geq 0$ for all A and $\|A\|_2 = 0$ if and only if $A = 0$.
- *Matrix–vector product.* $\|Ax\| \leq \|A\|_2\,\|x\|$ for all vectors x (that can be multiplied with A).
- *Matrix product.* $\|AB\|_2 \leq \|A\|_2\,\|B\|_2$ for all matrices B (that can be multiplied with A).
- *Transpose.* $\|A\|_2 = \|A^T\|_2$ (see exercises).

Computing the norm of a matrix The simple examples given above are meant to illustrate the definition of matrix norm, and not to suggest a practical method for calculating the matrix norm. In fact, except for simple matrices, it is very difficult to see which vector x maximizes $\|Ax\|/\|x\|$ and it is usually impossible to find the norm by inspection or a simple calculation.

In practice, however, there exist efficient and reliable *numerical* methods for calculating the norm of a matrix. In MATLAB the command is `norm(A)`. Algorithms for computing the matrix norm are based on techniques that are not covered in this course. For our purposes it is sufficient to know that the norm of a matrix is readily computed.

6.3 Condition number

Suppose we are given a set of linear equations $Ax = b$, with A nonsingular and of order n. The solution x exists and is unique, and can be expressed as $x = A^{-1}b$. Now suppose we replace b with $b + \Delta b$. The new solution is

$$x + \Delta x = A^{-1}(b + \Delta b) = A^{-1}b + A^{-1}\Delta b = x + A^{-1}\Delta b,$$

so $\Delta x = A^{-1}\Delta b$. We are interested in deriving bounds on Δx, given bounds on Δb.

We discuss two types of bounds: one relating the *absolute errors* $\|\Delta x\|$ and $\|\Delta b\|$ (measured in Euclidean norm), the second relating the *relative errors* $\|\Delta x\|/\|x\|$ and $\|\Delta b\|/\|b\|$.

Absolute error bounds Using the fifth property of matrix norms on page 61, we find the following bound on $\|\Delta x\| = \|A^{-1}\Delta b\|$:

$$\|\Delta x\| \leq \|A^{-1}\|_2 \, \|\Delta b\|. \tag{6.2}$$

This means that if $\|A^{-1}\|_2$ is small, then small changes in the right-hand side b always result in small changes in x. On the other hand if $\|A^{-1}\|_2$ is large, then small changes in b may result in large changes in x.

Relative error bounds We find a bound on the relative error by combining (6.2) and $\|b\| \leq \|A\|_2 \, \|x\|$ (which follows from $b = Ax$ and property 5):

$$\frac{\|\Delta x\|}{\|x\|} \leq \|A\|_2 \, \|A^{-1}\|_2 \frac{\|\Delta b\|}{\|b\|}. \tag{6.3}$$

The product $\|A\|_2 \, \|A^{-1}\|_2$ is called the *condition number* of A, and is denoted $\kappa(A)$:

$$\kappa(A) = \|A\|_2 \, \|A^{-1}\|_2.$$

Using this notation we can write the inequality (6.3) as

$$\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A) \frac{\|\Delta b\|}{\|b\|}.$$

It follows from the sixth property of matrix norms on page 61 that

$$\kappa(A) = \|A\|_2 \|A^{-1}\|_2 \geq \|AA^{-1}\|_2 = \|I\|_2 = 1$$

for all nonsingular A. A small condition number (*i.e.*, $\kappa(A)$ close to one), means that the relative error in x is not much larger than the relative error in b. We say that the matrix A is *well-conditioned* if its condition number is small. A large condition number means that the relative error in x may be much larger than the relative error in b. If $\kappa(A)$ is large, we say that A is *ill-conditioned*.

Computing the condition number The MATLAB command `cond(A)` computes the condition number of a matrix A (using a method that we will not cover). The commands `rcond(A)` and `condest(A)` provide very fast *estimates* of the condition number or its inverse. Fast condition number estimators are useful for large matrices, when calculating the exact condition number is too expensive.

6.4 Algorithm stability

Numerical algorithms almost never compute the exact solution of a mathematical problem, but only a very good approximation. The most important source of error is rounding error, introduced by the finite precision used by computers. The result of an algorithm may also be inaccurate because the method is based on discretization (*e.g.*, in numerical integration, or when solving differential equations), or truncation (*e.g.*, evaluating a function based on the Taylor series).

An algorithm is *numerically stable* if small errors introduced during the calculations lead to small errors in the result. It is *unstable* if small errors during the calculation can lead to very large errors in the result. Again, this is an informal definition, but sufficient for our purposes.

We have already encountered two examples of instability. In section 1.6 we noted that solving linear equations via the LU factorization without row permutations is unstable. We have also seen that the QR factorization method for solving least-squares problems is more stable than the Cholesky factorization method. We will give other examples of instability in section 6.5.

Note the difference between condition and stability. Condition is a property of a *problem*, while stability is a property of an *algorithm*. If a problem is badly conditioned and the parameters are subject to error, then the solution will be inaccurate, regardless of how it is computed. If an algorithm is unstable, then the result is inaccurate because the algorithm introduces “unnecessarily” large errors.

In practice, of course, we should always try to use stable algorithms, while the condition of a problem is not always under our control.

6.5 Cancellation

Instability in an algorithm is often (but not always) caused by an effect called *cancellation.* Cancellation occurs when two numbers are subtracted that are almost equal, and one of the numbers or both are subject to error (for example, due to rounding error in previous calculations).

Suppose

$$\hat{x} = x + \Delta x, \qquad \hat{y} = y + \Delta y$$

are approximations of two numbers x, y, with absolute errors $|\Delta x|$ and $|\Delta y|$, respectively. The relative error in the difference $\hat{x} - \hat{y}$ is

$$\frac{|(\hat{x}-\hat{y})-(x-y)|}{|x-y|} = \frac{|\Delta x - \Delta y|}{|x-y|} \leq \frac{|\Delta x| + |\Delta y|}{|x-y|}.$$

(The upper bound is achieved when Δx and Δy have opposite signs.) We see that if $x - y$ is small, then the relative error in $\hat{x} - \hat{y}$ can be very large, and much larger than the relative errors in $\hat{x}$ and $\hat{y}$. The result is that the relative errors $|\Delta x|/|x|$, $|\Delta y|/|y|$ are magnified enormously.

For example, suppose $x = 1$, $y = 1{+}10^{-5}$, and x and y have been calculated with an accuracy of about 10 significant digits, *i.e.*, $|\Delta x|/|x| \approx 10^{-10}$ and $|\Delta y|/|y| \approx 10^{-10}$. The error in the result is

$$\frac{|(\hat{x}-\hat{y})-(x-y)|}{|x-y|} \leq \frac{|\Delta x| + |\Delta y|}{|x-y|} \approx \frac{2 \cdot 10^{-10}}{|x-y|} = 2 \cdot 10^{-5}.$$

The result may only have about 5 correct digits.

Example The most straightforward method for computing the two roots of the quadratic equation

$$ax^2 + bx + c = 0$$

(with $a \neq 0$) is to evaluate the expressions

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

This method is unstable if $b^2 \gg |4ac|$. If $b > 0$, there is a danger of cancellation in the expression for x_1. If $b < 0$, cancellation may occur in the expression for x_2.

For example, suppose $a = c = 1$, $b = 10^5 + 10^{-5}$. The exact roots are given by

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = -10^{-5}, \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} = -10^5,$$

We evaluate these expressions in MATLAB, rounding the square roots to 6 correct digits using the `chop` function:

```
>> a = 1;  b = 1e5 + 1e-5;  c = 1;
>> x1 = (-b + chop(sqrt(b^2 - 4*a*c), 6)) / (2*a)
ans =
   -5.0000e-6
>> x2 = (-b - chop(sqrt(b^2 - 4*a*c), 6)) / (2*a)
ans =
   -1.0000e+05
```

The relative error in x_1 is 50%, and is due to cancellation.

We can formulate an algorithm that is more stable if $b^2 \gg |4ac|$ as follows. First suppose $b > 0$, so we have cancellation in the expression for x_1. In this case we can calculate x_2 accurately. The expression for x_1 can be reformulated as

$$\begin{aligned} x_1 &= \frac{\left(-b+\sqrt{b^2-4ac}\right)\left(-b-\sqrt{b^2-4ac}\right)}{(2a)\left(-b-\sqrt{b^2-4ac}\right)} \\ &= \frac{b^2-b^2+4ac}{(2a)(-b-\sqrt{b^2-4ac})} \\ &= \frac{c}{ax_2}. \end{aligned}$$

Similarly, if $b > 0$, we can use the expression $x_2 = c/(ax_1)$ to compute x_2, given x_1. The modified algorithm that avoids cancellation is therefore:

- if $b \leq 0$, calculate

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}, \qquad x_2 = \frac{c}{ax_1}$$

- if $b > 0$, calculate

$$x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}, \qquad x_1 = \frac{c}{ax_2}.$$

For the example, we get

```
>> a = 1;  b = 1e5 + 1e-5;  c = 1;
>> x2 = (-b - chop(sqrt(b^2 - 4*a*c), 6)) / (2*a)
ans =
   -1.0000e+05
>> x1 = c / (a*x2)
ans =
   -1.0000e-05
```

Chapter 7

Floating-point numbers

7.1 IEEE floating-point numbers

Binary floating-point numbers Modern computers use a binary floating-point format to represent real numbers. We use the notation

$$x = \pm(.d_1 d_2 \ldots d_n)_2 \cdot 2^e \tag{7.1}$$

to represent a real number x in binary floating-point notation. The first part, $.d_1 d_2 \ldots d_n$, is called the *mantissa*, and d_i is called the ith bit of the mantissa. The first bit d_1 is always equal to 1; the other $n-1$ bits can be 1 or 0. The number of bits in the mantissa, n, is called the *mantissa length*. The exponent e is an integer that can take any value between some minimum $e_{\min}$ and a maximum $e_{\max}$.

The notation (7.1) represents the number

$$x = \pm(d_1 2^{-1} + d_2 2^{-2} + \cdots + d_n 2^{-n}) \cdot 2^e.$$

For example, the number 12.625 can be written as

$$12.625 = (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 0 \cdot 2^{-8}) \cdot 2^4$$

and therefore its binary representation (with mantissa length 8) is

$$12.625 = +(.11001010)_2 \cdot 2^4.$$

Example A binary floating-point number system is specified by three numbers: the mantissa length n, the maximum exponent $e_{\max}$, and the minimum exponent $e_{\min}$. As a simple example, suppose $n = 3$, $e_{\min} = -1$ and $e_{\max} = 2$. Figure 7.1 shows all possible positive numbers in this number system.

We can make several interesting observations. First, it is clear that there are only finitely many numbers (16 positive and 16 negative numbers). The smallest number is

$$+(.100)_2 \cdot 2^{-1} = 0.25,$$

and the largest number is

$$+(.111)_2 \cdot 2^2 = (2^{-1} + 2^{-2} + 2^{-3}) \cdot 2^2 = 3.5.$$

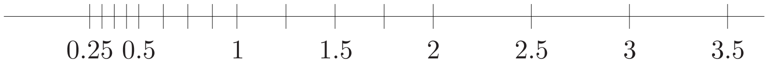

Figure 7.1 Positive numbers in a binary floating-point system with $n = 3$, $e_{\min} = -1$, $e_{\max} = 2$.

Second, the spacing between the numbers is not constant. There are four numbers with exponent $e = -1$:

$$+(.100)_2 \cdot 2^{-1} = 0.25, \quad +(.101)_2 \cdot 2^{-1} = 0.3125,$$
$$+(.110)_2 \cdot 2^{-1} = 0.375, \quad +(.111)_2 \cdot 2^{-1} = 0.4375,$$

and the difference between these numbers is $2^{-4} = 0.0625$. The next four numbers are

$$+(.100)_2 \cdot 2^{0} = 0.5, \quad +(.101)_2 \cdot 2^{0} = 0.625,$$
$$+(.110)_2 \cdot 2^{0} = 0.75, \quad +(.111)_2 \cdot 2^{0} = 0.875,$$

with a spacing of $2^{-3} = 0.125$. These are followed by the four numbers 1, 1.25, 1.5 and 1.75 (with a spacing of $2^{-2} = 0.25$) and the four numbers 2, 2.5, 3 and 3.5 (with a spacing of $2^{-1} = 0.5$).

Finally, we note that the number 0 cannot be represented, because the first bit d_1 is normalized to be 1. In practical implementations, the number 0 is represented by giving the exponent e a special value (see below).

The IEEE standard Almost all computers implement the so-called IEEE floating-point standard (published in 1985). The standard defines two types of floating-point numbers: single and double. Since numerical computations almost always use the double format, we will restrict the discussion to IEEE double precision numbers.

The IEEE double format is a binary floating-point system with

$$n = 53, \qquad e_{\min} = -1021, \qquad e_{\max} = 1024.$$

To represent an IEEE double precision number we need 64 bits: one sign bit, 52 bits for the mantissa (recall that the first bit is always one, so we do not have to store it), and 11 bits to represent the exponent. Note that the bit string for the exponent can actually take $2^{11} = 2048$ different values, while only 2046 values are needed to represent all integers between -1021 and 1024. The two remaining values are given a special meaning. One value is used to represent *subnormal* numbers, *i.e.*, small numbers with first bit $d_1 = 0$, including the number 0. The other value represents $\pm\infty$ or NaN, depending on the value of the mantissa. The value $\pm\infty$ indicates overflow; NaN (not a number) indicates arithmetic error.

7.2 Machine precision

The mantissa length n is by far the most important of the three numbers n, $e_{\min}$, $e_{\max}$, that specify a floating-point system. It is related to the *machine precision*, which is defined as

$$\epsilon_{\mathrm{M}} = 2^{-n}.$$

In practice (for IEEE double-precision numbers), $n = 53$ and

$$\epsilon_{\mathrm{M}} = 2^{-53} \approx 1.1102 \cdot 10^{-16}.$$

The machine precision can be interpreted in several ways. We can first note that $1 + 2\epsilon_{\mathrm{M}}$ is the smallest floating-point number greater than 1: the number 1 is represented as

$$1 = +(.100 \cdots 00)_2 \cdot 2^1,$$

so the next higher floating-point number is

$$+(.100 \cdots 01)_2 \cdot 2^1 = (2^{-1} + 2^{-n}) \cdot 2^1 = 1 + 2\epsilon_{\mathrm{M}}.$$

More generally, $2\epsilon_{\mathrm{M}}$ is the difference between consecutive floating-point numbers in the interval $[1, 2]$. In the interval $[2, 4]$, the spacing between floating-point numbers increases:

$$2 = +(.100 \cdots 00)_2 \cdot 2^2,$$

so the next number is

$$+(.100 \cdots 01)_2 \cdot 2^2 = (2^{-1} + 2^{-n}) \cdot 2^2 = 2 + 4\epsilon_{\mathrm{M}}.$$

Similarly, the distance between consecutive numbers in $[4, 8]$ is $8\epsilon_{\mathrm{M}}$, and so on.

7.3 Rounding

If we want to represent a number x that is not a floating-point number, we have to round it to the nearest floating-point number. We will denote by $\mathrm{fl}(x)$ the floating-point number closest to x, and refer to $\mathrm{fl}(x)$ as the floating-point representation of x. When there is a tie, *i.e.*, when x is exactly in the middle between two consecutive floating-point numbers, the floating-point number with least significant bit 0 (*i.e.*, $d_n = 0$) is chosen.

For example, as we have seen, the smallest floating-point number greater than 1 is $1 + 2\epsilon_{\mathrm{M}}$. Therefore, $\mathrm{fl}(x) = 1$ for $1 \leq x < 1 + \epsilon_{\mathrm{M}}$, and $\mathrm{fl}(x) = 1 + 2\epsilon_{\mathrm{M}}$ for $1 + \epsilon_{\mathrm{M}} < x \leq 1 + 2\epsilon_{\mathrm{M}}$. The number $x = 1 + \epsilon_{\mathrm{M}}$ is the midpoint of the interval $[1, 1 + 2\epsilon_{\mathrm{M}}]$. In this case we choose $\mathrm{fl}(1 + \epsilon_{\mathrm{M}}) = 1$, because

$$1 = +(.10 \ldots 00) \cdot 2^1, \qquad 1 + 2\epsilon_{\mathrm{M}} = +(.10 \ldots 01) \cdot 2^1,$$

and the tie-breaking rule says that we choose the number with least significant bit 0. This provides a second interpretation of the machine precision: all numbers $x \in [1, 1 + \epsilon_{\mathrm{M}}]$ are indistinguishable from 1.

The machine precision also provides a bound on the rounding error. For example, if $1 \leq x \leq 2$, the maximum rounding error is ϵ_{M}:

$$|\,\mathrm{fl}(x) - x| \leq \epsilon_{\mathrm{M}}$$

for $x \in [1,2]$. For $x \in [2,4]$, the maximum rounding error is $|\,\mathrm{fl}(x) - x| \leq 2\epsilon_\mathrm{M}$, et cetera. It can also be shown that

$$\frac{|\,\mathrm{fl}(x) - x|}{x} \leq \epsilon_\mathrm{M}$$

for all x. This bound gives a third interpretation of the machine precision: ϵ_M is an upper bound on the relative error that results from rounding a real number to the nearest floating-point number. In IEEE double precision arithmetic, this means that the relative error due to rounding is about $1.11 \cdot 10^{-16}$, *i.e.*, the precision is roughly equivalent to 15 or 16 significant digits in a decimal representation.